Verification Methodology

Version 00.01

Revision History

Date	Author	Description
14 April 17	Anton Tschank	00.01: Early draft

Table of Contents

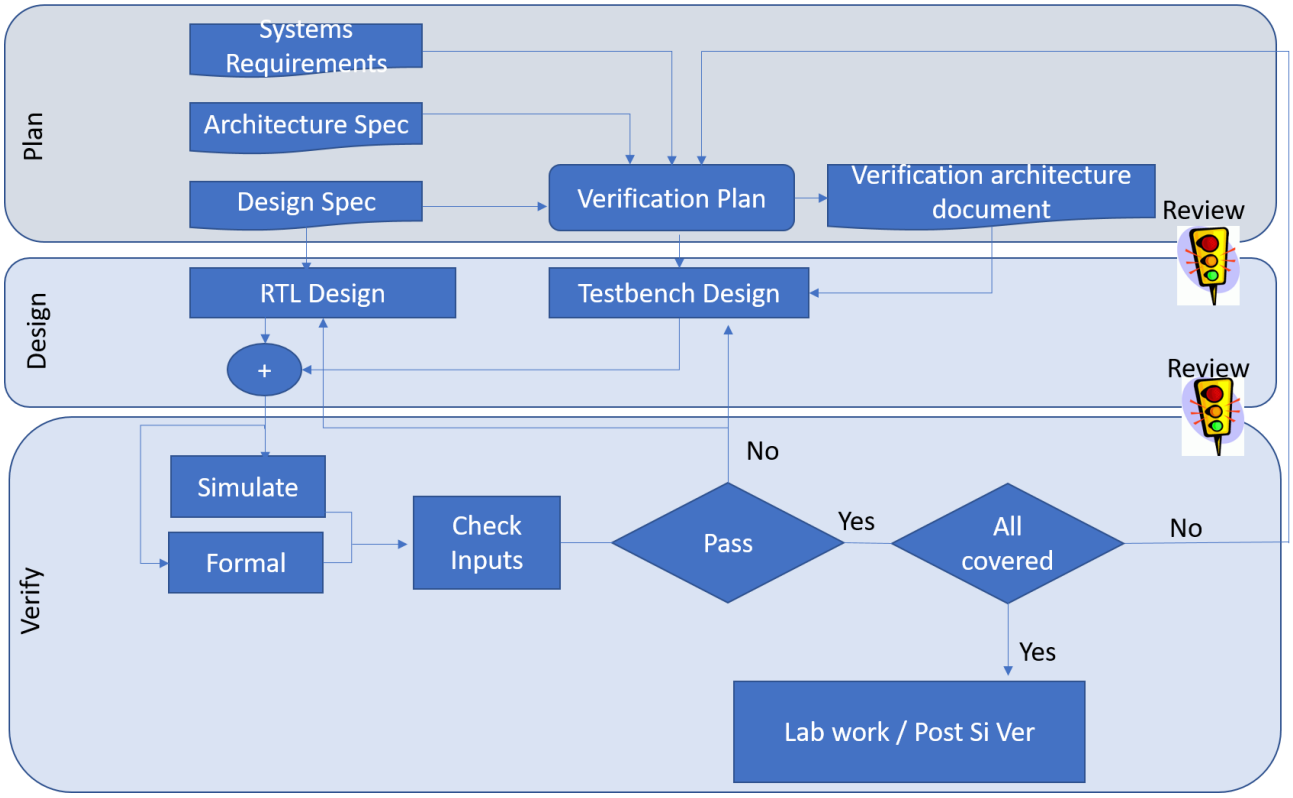
1 PURPOSE

Good quality verification of any design size is a very daunting process that requires good planning, good methodology, architecting and execution.

With design complexity ever increasing and project schedules getting tighter, it is important to have a strong verification methodology which will contribute to high quality design with first silicon success. A good methodology will aid in creating re-usable environments and enforce accountability thorough verification planning and metric driven verification.

This document is therefore there to help and guide you in successfully strategizing verification for your project.

2 VERIFICATION STRATEGY



2.1 Verification Plan

A verification plan is a living document used that helps ensure that the functionalities of the design is fully verified.

Verification planning is key to having a successful verification process. As an old saying goes, if you fail to plan, you plan to fail.

Creating a good verification plan requires close collaboration from all the key stakeholders i.e. architect, project lead, designer and verification engineer etc.

Verification planning involves identifying the features of the design that needs to be verified, prioritizing those features, measuring progress, and adjusting the allocation of verification resources so that verification closure can be reached on the required timescale.

The plan must be able to correlate design requirements, functional specification, designers' intent and testbench implementation. In the absence of using vendor supplied planning tools such as Cadence's vmanager, the plan should be created using a suitable format that is easy to review such as a spreadsheet. In addition to specifying exactly what needs to be verified, it should also contain other sections such as references to specifications, priority, coverage goal, co-features, interface features e.t.c.

Coverage for each feature should be annotated back to the plan after every simulation in order to easily and efficiently measure progress and create a clear path to verification closure.

Since a lot of function coverage code is well defined and structured, the verification plan could be used with complimentary scripting to automatically generate the coverage points. This ensures that the plan and the

testbench are always in sync with each other.

	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	Item Description					Coverage point descriptions														Results
2	Item Details	Spec Reference	Priority	Verification details	Test/seq/rand	Coverpoint Comments	Coverage Base Class	Hierarchy Override	Non default covergroup	Label	iff	Coverpoint	Cross	bins	ignore_bins	illegal_bins	wildcard_bins	at_least	Bind	func coverage %
3	Verify the correct behaviour of the PLL. Check that the correct output clocks (3GHz or 1.5 GHz) are generated using the specified reference clock (25, 40, 125 or 156.25 MHz)	5.01	High	Apply valid refclk, configure MPY and measure the output frequency when locked	tb_rand_pll test							refclk_period, pll_period, mpy, en								
4	Check the correct behaviour of PLL Lock	2.1	High	When locked, lock should be deasserted then re-lock when refclk, enable or mpy settings are changed	tb_rand_pll_lock test							mpy, refclk_period, pll_lock, pll_en								

2.1.1 Key guidelines

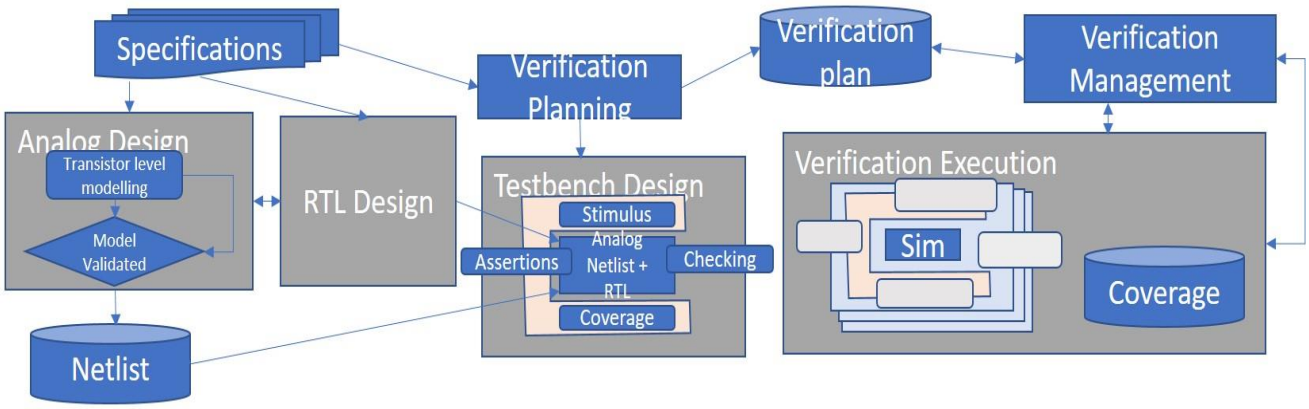
- Have an initial brainstorm with all the stakeholders capturing features.
- Focus on features and on the “what to verify”. The “how” can be answered later.
- Try to account for any and all CDC paths.
- Identify critical design elements through the data/control path.
- The plan shall have priority fields.
- The plan shall have references to specification documents.
- Each item should have a goal metric.
- Verification intent for each item should be covered.
- The plan should be structured properly and allow to be reviewed easily by stakeholders.
- Fields that aid in testbench automation should be portioned such that it details can be hidden to allow easier review.
- Use a structured a top-down process and add necessary filter mechanism.
- Metrics data from regressions should be annotated back to the design.
- The plan should contain a field to show current implementation status of each item
- Elaborate metrics (‘how’ to measure success) where possible.
- Where there a co-features or block interactions, these should also be clearly captured.
- Try to be as complete as possible.
- The plan should have a low barrier to automation and integration. i.e. structure in a way to maximize efficiency though automation in coverage/test generation from the plan and annotating results back.
- Regular reviews by all stakeholders should be made on every milestones and measure taken to ensure the plan is updated on every specification change.

2.2 Metric Driven Verification

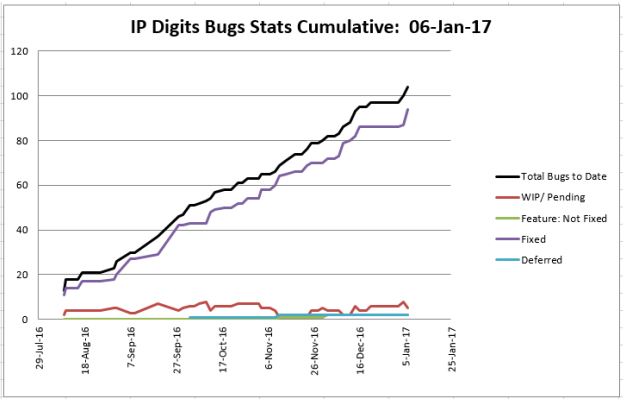
Metric-driven verification requires a significant change in mindset and practice when compared to directed testing.

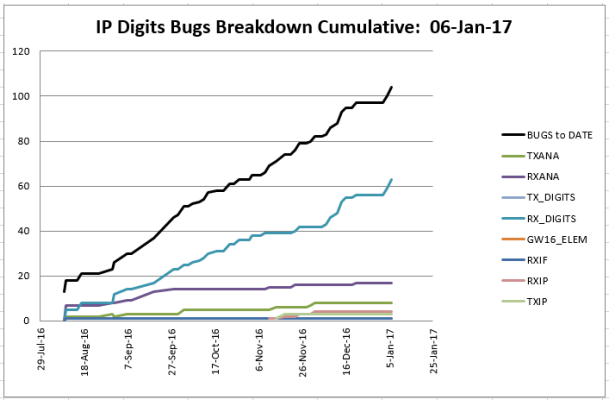
Instead of writing directed tests to exercise specific features, the features to be tested are fully implemented in the coverage model instead and the tests are there only to steer the constrained random stimulus generation toward filling any coverage holes.

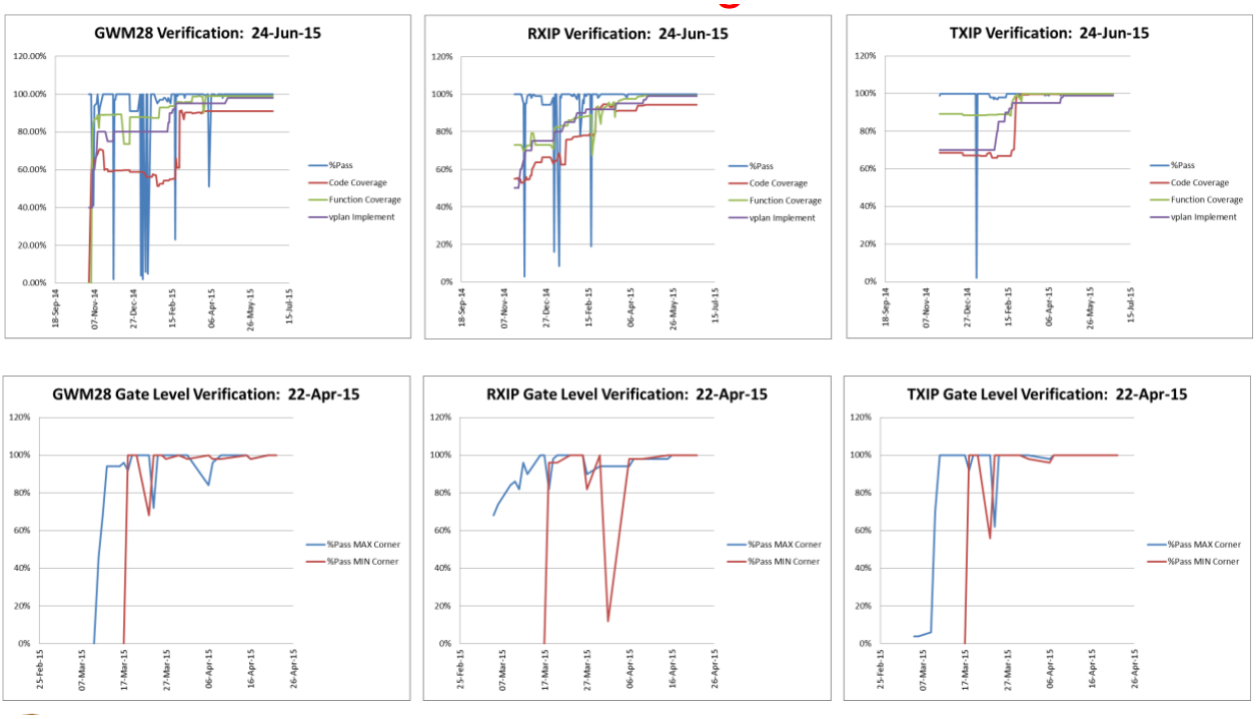
Using coverage to measure successful execution of the verification plan ensures a tight association between the original specification and the output from the functional verification process. This association improves requirements traceability and makes it easier to respond to specification changes.



Remember that verification can only show presence of bugs and not absence of any bug. Therefore, in addition to using coverage, we need to use other metrics in order to have an even larger confidence that the design is of the best quality before tape-out. There are many other metrics that could be used, such as bugs rate, pass/fail rate, historical data, number of simulation cycle etc.

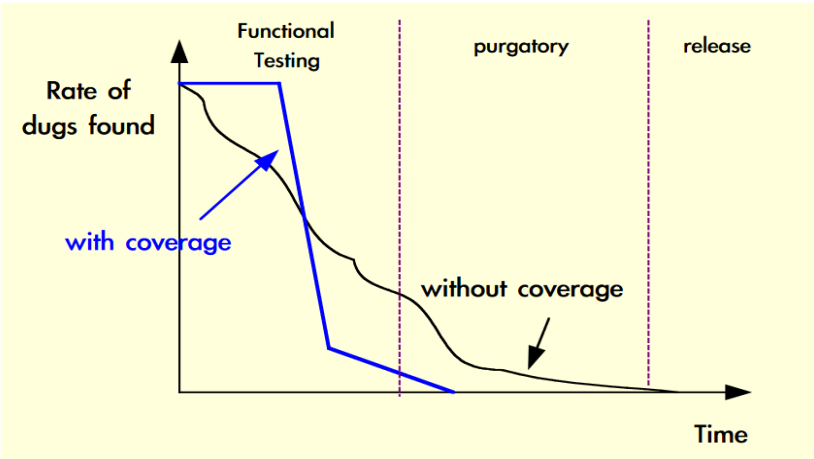






2.3 Coverage

Having a verification environment without coverage is like a pilot flying blind. This is especially important when using constraint random approach as you need to measure progress and the feature/state space that you have already covered.



Functional coverage and code coverage measure different things. Code coverage measures the execution of the actual RTL code (which must therefore exist before code coverage can be run at all). The collection of code coverage information, including statement and branch coverage, state coverage, and state transition coverage, is largely automatic. Code coverage can be considered as a quantitative measure of DUT code execution. Functional coverage, on the other hand, attempts to measure whether the features described in the verification plan have actually been executed by the DUT. The features to be measured have to be "brainstormed" from the specification and implementation of the design to create the verification plan, and

so functional coverage can be considered as a qualitative measure of DUT code execution. The number of features to be *hit* by functional coverage depends upon the diligence and thoroughness of the humans who draw up the verification plan.

2.3.1 Code Coverage

Code coverage is a metric measures the structural aspect of the code but not necessarily functionality. These can be toggle activity, statements, branches, expression, Finite State machines etc.

There are many types of code coverage as listed below.

- **Line Coverage** – Checks that each line of the code has been exercised
- **Branch Coverage** – Examines branches of each conditional statements and counts true/false conditions that have been hit.
- **Expression Coverage** – Monitors logical expressions and indicates whether all the possible logical expressions have been exercised.
- **Toggle Coverage** – Monitors signal logic values transitions. i.e. a signal transitions from 0->1->-0 or 1->0->1;
- **FSM Coverage** – Checks whether all the state transitions in a finite state machine occur.

Code coverage is normally available automatically through the tool with negligible effort need to set-up. It is an important metric as it helps in determining:

- **Dead-code –** Areas of code which are unreachable due to either bug in design, redundant code or legacy code.
- **Quality of function coverage –** It can help determining the quality of function coverage. E.g. when function coverage is 100% but code coverage is low, it may imply that we have untested features.
- **Optimizing code –** Can help in determining badly written code (FSM, expressions etc.)

Target

Due to many factors such as complexities of design, legacy code, static signals etc., it may be very difficult to achieve a 100% code coverage. However, with the help of filtering/waivers, you should still aim at achieving 100% coverage. This is what I refer to as "**explained coverage**". These waivers/filters must be reviewed periodically and compulsory during sign-off.

2.3.2 Function Coverage

Each feature on the verification plan is implemented by creating a function coverage point to collect coverage information. This is normally referred to as the coverage model. Typically, the coverage should be collected at the checker or scoreboard after the checker passes to ensure that there are no false passes. Every test that fails should have its coverage information discarded.

Ruthless prioritization is also needed to ensure you meet business needs. Focus first on critical functionality before closing low hanging fruits.

Best practice

- Focus on functionality.

- Make sure your sampling at the correct point. Preferably in the scoreboard after checking or in monitors after checkers. Never trigger coverage before checker passes!
- Name coverage and crosses based on intent.
- Wrap covergroups in classes to enable re-usability and portability.
- Sample coverage by calling sample() method explicitly rather than event triggered.
- Make your coverage readable and reviewable. Keep in mind reviewers may not have any knowledge of SV or UVM!
- Group multiple conditions appropriately.
- When re-using coverage, ensure that it is still accurate!
- Avoid unrealistically large bins or crosses. Use sensible bin ranges to reduce volume.
- Beware of trying to re-implement toggle coverage. This is already available for free from the tool!
- Include sequences and scenarios.
- Trap illegal states but use illegal bins carefully. When used correctly can be useful for debug.

Target

Before sign-off, function coverage must be 100%.

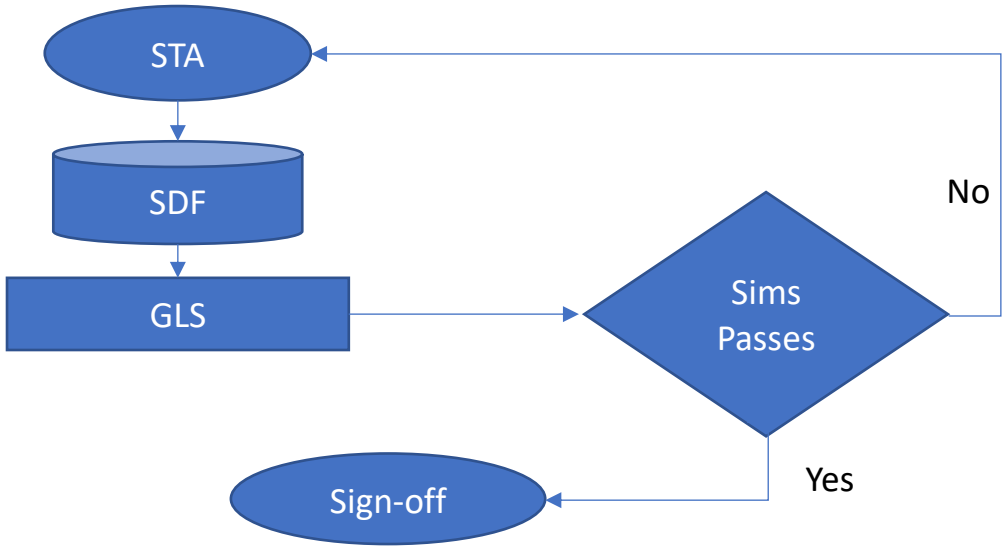
2.4 Gate Level Verification

Gate-level simulation (GLS) with annotated timing (SDF) must be run on at least two corners (max/min delays).

As wells as boosting confidence regarding the physical implementation of the design, GLS helps to verify that the design works dynamically which static timing analysis tools (STA) would not accurately establish.

This that GLS typically catches are:

- X optimism in RTL.
- Constraints errors.
- False paths errors – Paths which have been wrongly constrained as false paths but actually affect mission mode.
- Wrongly multi-cycled paths. - paths that have been multi-cycled but actually should not have been.
- Glitches and synchronization issues.
- Reset sequences
- Scan chain hookup issues – Normally scan chain are not inserted in the RTL, so if they were issues during insertion that breaks mission mode, the can be caught with GLS.



Since gate level sims are very time consuming, they only need to be run on a subset of tests. These tests must include:

- Power-up and reset sequence test
- Default data path test
- Different data rates
- Slow speed test.
- Boundary scan test.

2.5 Reference model integration

Where an architectural model is available, preferably C, C++ or SystemC. Every effort should be made to try and integrate this into the testbench environment, preferably in the scoreboard. This will help in further cross checking architectural requirement versus design implementation as well as aid in debugging.

2.5.1 Guidelines when integrating

- Use DPI rather than PLI
    - Much simpler and faster!
- Encapsulate model.
    - Have wrapper functions that easily map to Verilog functions.
- Define extern import/export header tasks/functions in a separate file rather than embedding it in the source code.
- Use svArray methods when dealing with vectors.
- Pass large arrays by pointer.
- Make sure the context is clearly defined where necessary.

3 COMMON FEATURES VERIFIED

- Datapath
- Datarates
- Pattern checkers and verifiers
- Boundary scan (JTAG)
- CDR/CDA
- Loopback
- Calibration loops
- Eyescope.
- Offset Correction
- PLL

TODO

4 WORKING ENVIRONMENT

4.1 Tools

Tool	
Simulation	Cadence Incisive, Synopsys VCS
Formal	Jasper, IFV
Scripting	Perl, Python
Vplan	Vmanager, spreadsheet
Regression	Home grown perl tool, vmanager
CI	Jenkins
Bug tracking	Redmine

TODO

4.2 Version Control

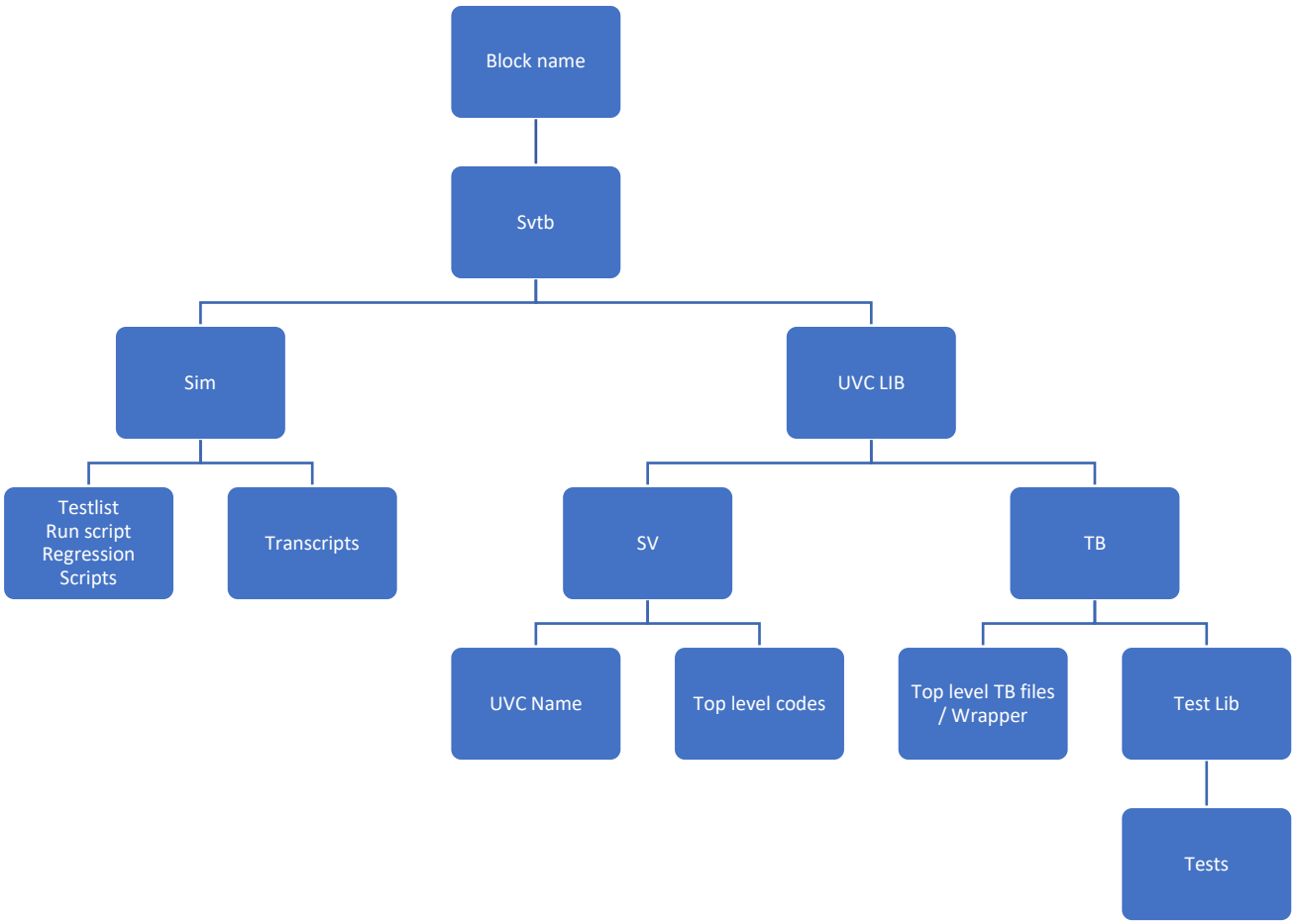
- All testbench needs to be under the same version control as the project.
- Need to ensure that latest updates are regularly checked in.
- Prior to checkin, s mini-regression must be run to ensure nothing is broken.

- Regression should be run using a regression only workspace (clean non-user workspace) which should always be updated to the latest label prior to kicking of a regression.

4.3 Location

Testbench environment should have a directory structure similar to the one below. Tests and top level wrappers should be under "tb" whilst UVCs and other TB testbench files for the environment should reside under the "sv" directory.

Simulations should be run under a completely separate directory hierarchy "sim". This ensures that the core testbench environment remains tidy and could easily be ported to other blocks or projects.



4.3.1 Directory Structure Summary

block_name

- ➢ svtb
  - ➢ uvc_lib
    - ➢ tb
      - Contains the top level UVM testbench class (i.e. where all the other UVCs are instantiated and configured). Name is typically called **<block_name>_tb.sv**

- Also has the top-level module which instantiated the DUT and the UVM testbench. Typically called **<block_name>_top.sv**
- Testcases also resides here in a sub-directory called **test_lib**

➢ sv
- Contains all other sub-uvcs (interfaces) in sub directories
- Also contains driver/monitors/sequences etc. for the current block uvc

➢ sim
➢ A single test (batch or interactive mode) is executed from here. Waveforms and Simulation files will be dumped in this directory. Script to run is **./runme.pl -t <testcase>**. Add -h for extra options on the script.
➢ Regressions are also launched from here using the script **./regress_me.pl -t <testlist>.** Add -h for extra options. Simulations are run in ../sim_<testcase>_<seed> directory and transcripts/log files will be saved in **./sim/transcripts** directory.

5 BLOCK / CLUSTERS VER METHODOLOGY

5.1 UVM Based Constrained Random SV

The methodology of choice is UVM. This works pretty well with the ethos of Metric driven verification. The main reasons for using UVM are:

- Mature methodology and open industry standard
- Supported by all major vendors
- Ensures maximum re-use
  - Solid base classes such as Factory overrides, register model methodology.
  - Object oriented.
- Good for constrained random environments
- Easy to port lower level environment to top level without much effort
- System verilog is a super set of verilog so designers can easily understand and help in TB development

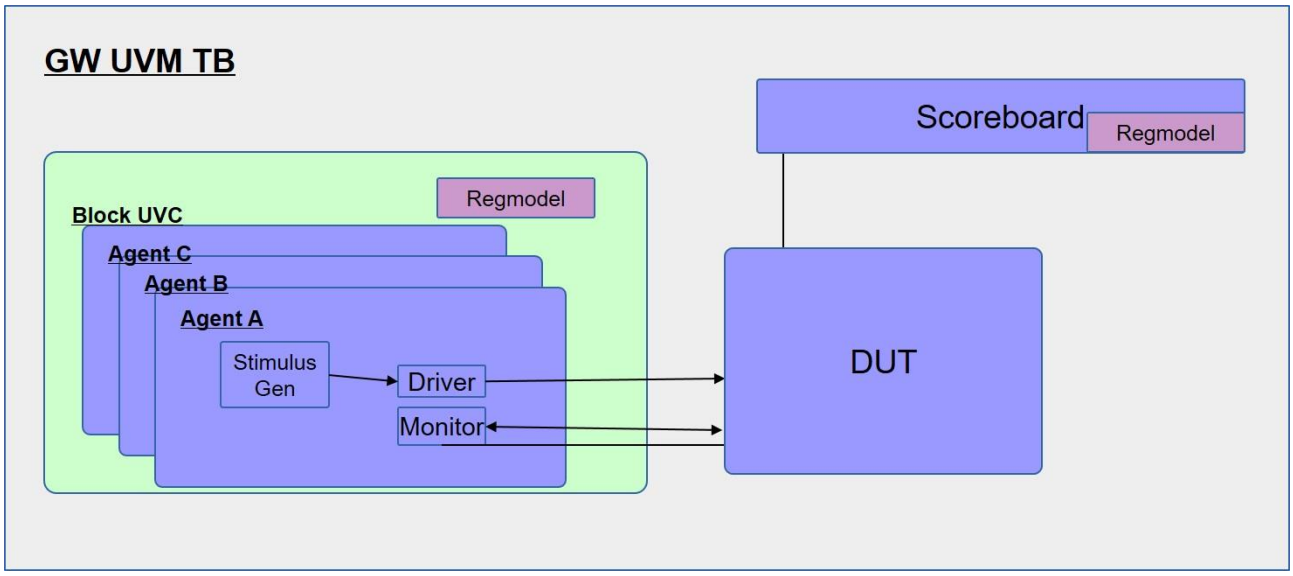
5.2 Top-up/Bottom-up hybrid Approach

Recommended approach for a new project will be a hybrid of both top-up and bottom up by default unless there is a clear advantage on doing it otherwise. This means in parallel, overall architecture of the TB is built with top-level in mind and a skeleton framework is put in place whilst at the same time focusing on verifying low level blocks which have already been implemented or are under development.

Main advantages are:

- Supports vertical re-use (and actually some horizontal) right from the onset. This will help in top-level IP and SOC verification.
- Integration time for top-level will be less as updates are continuously made as soon as a lower level block is available as opposed to waiting for all the sub-blocks to be completed.
- High level protocol issues are identified earlier in the process.

- Locality – Problem is narrowed down in a small area when doing block level.
- Debug and catching bugs is easier and faster due to dealing with a smaller scope.
- You do not need to wait for the full design to be complete before starting verification
- When doing top-level verification, you'll have a higher confidence of the design being functional due to "bug-free" sub-blocks"
- Lower level environments are be re-used at the top level thus improving productivity and efficiency.



5.3 Main Blocks/Clusters

For a bi-directional SerDes like IP must have include:

- RXIP
- TXIP
- IP TO-LEVEL
- 2 DUT ENVIRONMENT

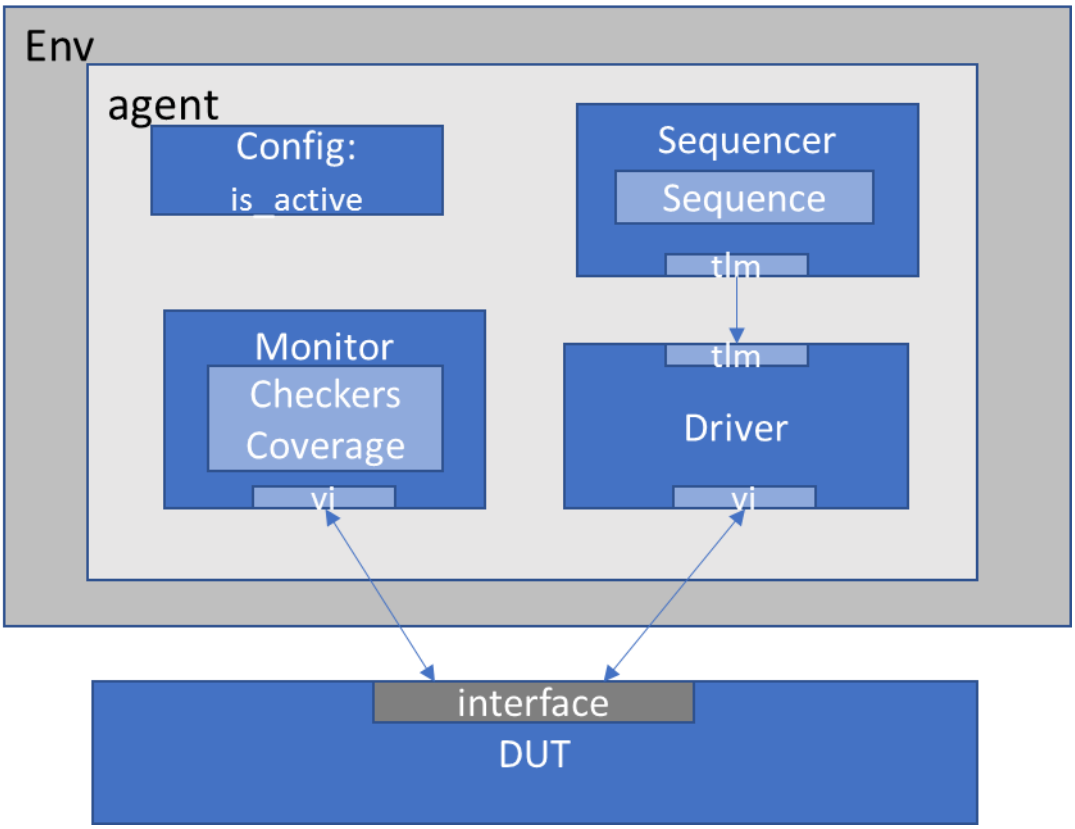
Within each cluster, it is also recommended to have multiple UVCs (UVM Verification Component), one for each interface. This helps in ensuring portability to different level of hierarchy as well as maximum reuse for other designs that they may share similar interfaces.

Each UVC (Agent) should have all the following UVM component/Object:

- Interface (with parameters where bus widths are programmable etc.)
- Transaction class (sequence_item)
- Driver
- Monitor
- Sequencer
- Base Sequence

- Config class (if programmability is required or more than one setup is available)

Each UVC should be able to be configured as ACTIVE (i.e. bus driver present) or PASSIVE (no driver or sequences just monitors), so as to allow them to be used in higher level of integration such as IP level or Chip level.



In the block / cluster, each UVC should be instantiated and connected from a uvm_env which will then be in turn be called/instantiated from a uvm_tb.

In addition to the requirements of the UVCs above, each block/cluster environment should contain a scoreboard, a config class and register model if present.

5.4 Register / memory map verification

- UVM reg shall be used.
- UVM register model should be automatically generated from register map documentation and control files.
    - Currently genregmodel.py
- For scalability and portability, backdoor accesses should be limited.
    - When used, the paths should be declared in a central place.
- Encapsulate functions using wrapper functions which allows easier access.

- See common_pkg.sv in gwmq16 svtb_shared.

5.5 Testing Digital, Analog and IOs

- Testbench should be fully self checking.
- TODO

5.6 Whitebox Testing

White-box testing can be a powerful way to test functionality in depth. However, it has some major draw backs such as:

- May mask functional issues and features that are not fully implemented.
- Requires constant updating of the test when design changes.
- Not very re-usable

By default, avoid using white-box testing unless absolutely necessary. If there is genuine observability missing a recommendation should be made to get these added in the design. If there's a strong need to observe or stimulate internal signals, then best use a **grey-box** approach. i.e. Use back-box for everything that you can and only use internal signals when needed.

When white-box testing is being used, limit this on a test rather than the whole TB. Makes re-usability and portability easier.

5.7 Coding Guidelines

SystemVerilog is a very large and complex language. If you couple this with the sheer size and complexities of UVM, then this can create a big challenge for people adopting the methodology as there are often numerous ways of achieving the same thing. If not careful, depending on the path you may have taken, you may end up with code which is not re-usable or which does not scale easily.

It is therefore vitally important to have a set of coding guidelines to follow in order to avoid these pitfalls as well as making the code have the same structure on different blocks or projects especially given that often various people would be developing different areas of the testbench.

Here are some basic principles that you should follow:

5.7.1 SystemVerilog DOs

- Use a consistent coding style.
- Define one class per file.
- Use a descriptive typedef for variables.
- Use an end label for methods, classes and packages
- Use `includes to compile classes into packages
- Define classes within packages
- Only `include a file in one package
- Import packages to reference their contents
- Check that $cast() calls complete successfully. Use if/else rather than asserts

- Check that randomize() calls complete successfully. Use if/else rather than assert.
- Use if rather than assert to check the status of method calls.
- Wrap covergroups in class objects.
- Only sample covergroups using the sample() method.
- Label covergroup coverpoints and crosses.

5.7.2 SystemVerilog DON'Ts

- Avoid `including the same class in multiple locations
- Avoid placing code in $unit or $root.
- Avoid using associative arrays with a wildcard index.
- Avoid using #0 delays.
- Don't rely on static initialization order.

5.7.3 UVM DOs

- Do not hard code paths.
- Define classes within packages.
- Define one class per file.
- Use factory registration macros.
- Use message macros as opposed to calling uvm_report_* tasks directly.
- Manually implement do_copy(), do_compare(), etc.
- Use sequence.start(sequencer)
- Use start_item() and finish_item() for sequence items
- Use the uvm_config_db API to get and set variables and interfaces etc.
- Use a configuration class for each agent
- Use phase objection mechanism
- Use the phase_ready_to_end() func
- Use the run_phase() in transactors
- Use connect_phase() for all your connections or handle assignments.
- Use the reset/configure/main/shutdown phases in tests.

5.7.4 UVM DON'Ts

- Avoid `including a class in multiple locations.
- Avoid constructor arguments other than name and parent .
- Avoid field automation macros were not necessary. – performance impact!
- Avoid uvm_comparer policy class.
- Avoid the sequence list and default sequence.
- Avoid the sequence macros (`uvm_do).
- Avoid pre_body() and post_body() in a sequence.
- Avoid explicitly consuming time in sequences.
- Avoid set/get_config_string/_int/_object(). Use get#(type) instead
- Avoid the uvm_resource_db API. Use uvm_config_db instead.
- Avoid callbacks
- Avoid user defined phases unless absolutely necessary!!
- Avoid phase jumping and phase domains (for now)
- Avoid raising and lowering objections for every transaction.

6 CLOCK DOMAIN

As SOCs are becoming larger and more complex, it is now very common to have more than 2 clock domains particular in hi-speed communication designs. Cross Domain Crossing verification is therefore becoming more and more challenging and an important part of verification.

6.1 CDC Planning

The first step as with everything is planning. CDC planning goes beyond the traditional test planning and should follow the following guidelines:

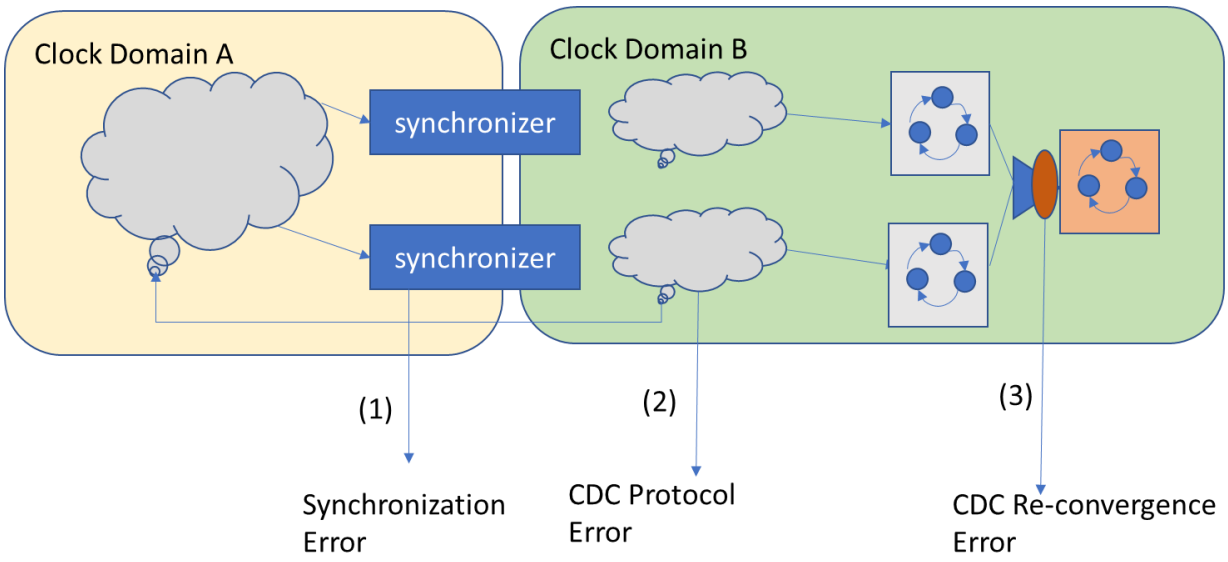
- Identify all the candidates for CDC verification.
- Document all the requirement for each CDC item.
- Formalize known exceptions.
- Define CDC coverage goal.
- Define a strategy for each CDC requirement.

6.2 CDC Strategy

In order to make CDC verification metric driven and scalable. The recommended strategy is to do the CDC verification into three steps:

- Static analysis to identify synchronization errors.
    - Annotate RTL with suitable BMMs.
- Protocol assertion generation and coverage to ensure that data is being sent and received correctly.
- Convergence verification with metastability injection using a suitable BMM (Behavioral Metastability Model)

In addition, CDC should be run hierarchically. i.e. you first run static CDC on an individual block to ensure it is CDC compliant. You should then have control files that characterize each block. After this CDC can then be run at the next level of hierarchy. This will typically find all the CDC re-convergence errors.



6.3 Method of Checking Data Across Domains

6.3.1 Behavioral modelling

In order to capture metastability issues during CDC, it is vitally important to have a robust mechanism that should be able to inject metastable state in the block. To do this, you should have a suitable (Behavioral Metastability Model) that could be annotated on to the RTL when running CDC checks.

Accurate metastability modelling should include:

- Single -Cycle delay effect.
  - Single cycle delay may be introduced when a metastable is captured.
- Bleed-Through effect
  - Output can introduce single cycle speedup when a metastable 1 is captured
- Individual bit modelling.
  - Each CDC should be modelled individually.
- Realistic setup/hold.
- Coverage reporting.

Methods of modelling can be:

- Simple clock jitter modelling
  - Drawback: Unsynchronized path problem. Will inject metastability even to paths not intended.
- 3-DFF Model
  - Randomly selects between 1 and 3 DFFS
  - Drawback: Can inaccurately suppress pulses. Values can be skewed by 2 cycles.

- 2 DFF with input delay
    - Replace 2DFF synchronizer with cells that randomly delay inputs
    - Drawback: No bleed through effect. does not cover unsynchronized path.

For synchronizers, apart from assertions and other checkers, it is recommended to also have a model that would delay the input randomly between 1 and 3 clock cycles and ensure there is no metastability.

6.3.2 Formal

With the continual development in formal methods, most vendors have a formal app that may help is significantly verifying CDC paths without spending too much resources. However, depending on the interfaces at chip level, there could be a significant initial effort needed to set this up.

7 FULL CHIP/IP VERIFICATION

Full chip / IP verification should be aimed at covering the following:

- Interaction of sub blocks or sub systems.
- Work with realistic clock domains
- Connectivity checking. – All sub blocks/systems are connected and communicating correctly.
- Correct overall application.
- Overall performance of the system.
- Reset and initialization protocols.

7.1 Interfaces

At chip level verification, as you know that the individual blocks have been thoroughly and robustly verified, verification at this level therefore should primarily focus on verifying the interfaces and interaction between blocks.

Interface verification is thus normally the starting point of chip level verification.

Inter-block interfaces normally have defined structures. Some could be datapath oriented, i.e. address and data busses connecting blocks whilst others could be more control path oriented such as request/acknowledgement protocol. and could be either point-to-point or on-chip busses.

Due to the regular structures of the interface it makes it easier to create an even higher level of abstraction and start testing these interfaces at a transactional level which will give you even more throughput in terms of simulation cycles. By transactional based, the idea is that there are only a few permitted sequences of control/data signals that change from one transaction to another.

7.1.1 Transaction verification process.

Start by listing all transactions that occur at each interface and start to systematically test each one of them. Once this is complete, all that remains to be done verify the behaviors of the blocks when using various data values in the transaction. To do this, you should instantiate the RTL/behavioral model of the block and drive some transactions using the testbench. The testbench should be self-checking and also have additional bus monitors that would inspect the transactions directly. Having the monitors not helps improve observability but can help in improving controllability thus making it possible to generate

transactions that are precise to the order that is needed. This allows you to use simple BFMs as opposed to having the full functional model in place which may be slow and difficult to understand.

7.1.2 Behavioral process.

Once all the transactions have been verified, it is also important to check that the block functions correctly using real data that would be used in actual operation. At chip-level, generating complete set o data is normally not possible due to complexities and sheer size. Remember that the goal is not to re-verify robustness of the block but to check higher level protocols.

The approach to efficiently check the behavior would be use Bus Functional Models (BFMs) for all blocks apart from the actual block under test which will be RTL instead. Automatic self-checking o these blocks is non-trivial and requires a level of knowledge of the sub-block and as such tehe checkers and BFMs should be implemented by re-using the block level verification environments in PASSIVE mode rather than having to re-implement them.

7.1.3 Verification IP and Formal verification apps.

As mentioned earlier, the main aim o chip level verification is to check or interconnectivity between sub-blocks or IP. A lot of development have been done in the formal verification tools provided by most vendors which may have off the shelf “Apps” such as connectivity app and register access app. Using these if available, could significantly help saving time used at chip level varication. Apart, from the formal apps, there may be already a verification IP (VIP) available, especially for standard protocols such as AMBA or PCIE etc. and you should consider using these as a more efficient approach.

7.1.4 Summary Interface verification guidelines.

- List down all inter-block interfaces and identify interfaces that could be shared.
- Re-use verification environment which were used at lower levels in passive mode.
- Use a higher level of abstraction.
- Use simplified models
    - BFM, bus monitors and checkers
- Verify whole system using functional models and test it would be used in real world as opposed to any order (this was done in block level verification!)
- Use formal apps and VIPs if available
- Run real world applications. This should boost your confidence in the chip working in real Si.

7.2 Configuration and Address Space

TODO

- Use RAL/UVM regs
- Formal Register verification APP

The register verification should also make sure the correct register access happens for all types. These must include:

- **Reset Checks** – Check all values after reset matches specifications.

- **Read/Write Checks** – Check read value of the register matches the last value written.
- **Read only check** – Check read-only registers are not writable. Cheick read register after write.
- **W1C** – check that writing one to the register clears the registers on the next read.
- **Read modify write**

7.3 Initialisation (Power-up, reset and initialization sequences)

TODO

-
-
- Check on reset, registers with reset have values that matches specification.
- Check that all registers without registers never fan-out to the outputs
- Check that registers without resets do not fan-out to critical logic such as FSMs.

8 MODELLING

Analog blocks should be modelled using VeriliogD by default. If more accuracy is needed then consider using VerilogAMS using Wreal. If even more accuracy is needed, Verilog AMS should be considered before starting a full blown VerilogA.

8.1 Level of modelling

In order to keeps the models simple and accurate, the models should be modelled only in the level of hierarchy where there are transistors or other analog components. A request can be made to the analog team to bury a device one level of hierarchy if it will make the modelling easier at that block.

Modelling at the lowest level possible would allow you to netlist and preserve most of the analog hierarchies and thus can catch connectivity issues such as bus swapping or missing biases.

8.2 Model Validation

- The designer of the block should review the models.
- Compare waveforms between schematic and model.
- Have automated mechanism to compare outputs within a certain accuracy threshold.

On critical blocks where there's close interaction loops between analog and digital, co-simulations should be run.

9 REGRESSION

Having an automated regression system is quite important in a metric-driven verification system. In addition, it also improves efficiency, productivity and quality.

At a minimum, the regression system should contain the following features:

- The regression system should be able to launch multiple jobs in batch mode, and allow mechanism to queue pending jobs.
- Should be able to analyze logs and send pass/fail summary.
- Ability to merge coverage results and generate necessary hooks.
- Ability to re-run ailing tests with the same setup and seed.
- Ability to accept multiple test suit lists.
- Should be easy to integrate it on a continuous integration tool such as Jenkins.
- Ability to annotate results back to the verification plan.

On every verification environment, it is recommended to have at least 3 separate regression lists and have the ability to run these individually. These are:

- **Mini Regression** – A short list that is very fast to run and has a test that could show if default design is broken. This regression should be run periodically after every design check-in. Will help in identifying design or testbench check-ins that may have broken the design.
- **Nightly Regression** – A regression list or tests that can be run and completed overnight. This should be the main testlist.
- **Log Regression** – This is very useful and is intended to run tests which typically take very long to complete. They need to be in a separate regression in order not to hold up other tests which could run far quicker.

A good regression system should also be able to tabulate results (pass/fail, coverage etc.) and automatically send email to the relevant stakeholders.

All regressions must be run from a clean work area and not the same one used to implement testbench updates or design changes. This will aid in identifying files that have not been checked in properly but at the same time also allow regressions to be run to completion without being broken by updates being made to “live data”. In addition to these, it also serves as an important audit and correlation point.

TODO: Add flow diagram

10 TEST PLAN

- Constrained random tests.

  - Have testscase per feature or subset of features.
  - Need also a fully constrained random test that would target a random feature
- Directed tests.
  - Only on a need basis.
  - Cover coverage holes.
  - Address a legacy bug or specific bug
- Power-up and reset tests are a must.
- Have a simple test that brings up IP in most simple way in mission mode
- Gate level must be run on a subset of tests. Must include:
  - Reset and powerup
  - Mission mode default test.
  - Some selected critical calibration loops.
  - Slow speed.
  - A random constrained test.

11 MILESTONES AND REVIEW STAGES

Milestone	Milestone Goal
Requirements and specifications review	Requirements specifications are complete and approved for starting design development process.
Initial Verification Plan review	First pass verification plan is complete and annotated back to specification document. Reviewed by key stakeholders. Architect, designers and verification engineer etc.
Preliminary testbench architecture review	Testbench architecture in place and reviewed fit for project.
Functional coverage implementation review	Coverage implementation reviewed to make sure it covers design needs
Preliminary design review	Architectural design in place and hierarchies defined
Preliminary coverage review	Coverage status reviewed and holes identified. Prioritise on what to tackle next. Target 50% for both function coverage and code coverage.
Final design review	Final designs are fully implemented. RTL/Schematics locked for verification.

Final verification plan review	Final testplan reviewed. Test procedures approved. Can proceed to verification/coverage closure.
Final coverage review	Coverage reviewed. Waivers approved. Targets: 100% function coverage. 100% explained code coverage.
Gate level review	Gate level test list reviewed and approved. Critical corners run. Waived violations approved.
Verification Sign-off review.	Verification plan Coverage Bugs rate Gate level sims Waivers

12 RISKS

A risk assessment must be completed to assess the possible consequences of a change so that proper action can be taken to eliminate or mitigate the risk. This will also help in identifying the level of verification required.

Each risk needs to be well defined and should identify what might go wrong, the likelihood of it going wrong and the consequence. The probability of identifying when things go wrong should also be made.

Some of the risks to be considered are:

- Systematic and architectural bugs.
- Bugs rate.
- Pending bugs.
- Schedule / resource
- Design lateness.
- Coverage closure.

13 SIGN-OFF

Coverage

Bugs rate

Gate level sim

Uncovered tests

Vplan implementation

Testplans review

TODO

d. Hi speed IO's Training

e. Test Bench's

f. Checks for 'X' (X propagation)

7. Modeling

a. System C for performance and architecture

b. C++ for features modes

8. Testing using FPGA (emulator if needed).

9. IP's verification

11.

a. Design guide lines

i. Coding method and guide lines

b. HR needed and function

i. Design blocks

ii. Integration

iii. IO's

iv. DFT

v. Clock & Reset

12. Main Mile Stones
13. Review points and method
14. Risks
15. Special Notes (if needed)