Applying Design Patterns to Maximize Verification Reuse at Block, Subsystem and System-on-Chip Level

Paul Kaunds | Revati Bothe | Jesvin Johnson

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

Agenda

- Design Patterns
- Features of a Good Software Design
- Good Software Design Principles
- SOLID Principle
- Design Pattern Examples
- Design Pattern Categories
- Design Pattern & UVM
- Design Pattern UVM Applications
- Metric Driven Verification @ Block, Subsystem & SoC-level
  - Verification Planning
  - Verification Environment Development
  - Stimulus Development
  - Execution
  - Coverage Closure
- Case Study
- Summary

Design Patterns

- What is the Design Pattern?
  - Design Patterns are typical solutions to commonly occurring problems in the software design
  - Like pre-crafted blueprints that can be customized to solve re-occurring design problems
  - Like a high-level description of a solution
    - Can predict results
    - Can inform about its features
    - Independent of exact order of implementation

Design Patterns

- What is the Design Pattern?
  - A pattern is a proven solution to a problem in a context.
  - Christopher Alexander says each pattern is a three-part rule which expresses a relation between a certain context, a problem, and a solution.
  - Design patterns represent a solutions to problems that arise when developing UVC within a particular context.
  - Patterns = problems.solution pairs in a context

Design Patterns

- What is the Design Pattern?
  - A Design pattern is a recurring solution to a standard problem, in a context.
  - Design patterns are Thought Processes.
  - Design Pattern is like Dress Patterns
  - Jim Coplein, a software engineer: “I like to relate this definition to dress patterns…

Design Pattern

- How are Design Patterns described?
  - Intent
    - Briefly describes the Problem & Solution
  - Motivation
    - Further details the Problem & Solution that the pattern makes possible
  - Structure
    - Inter-relation between different classes to show each part of the pattern

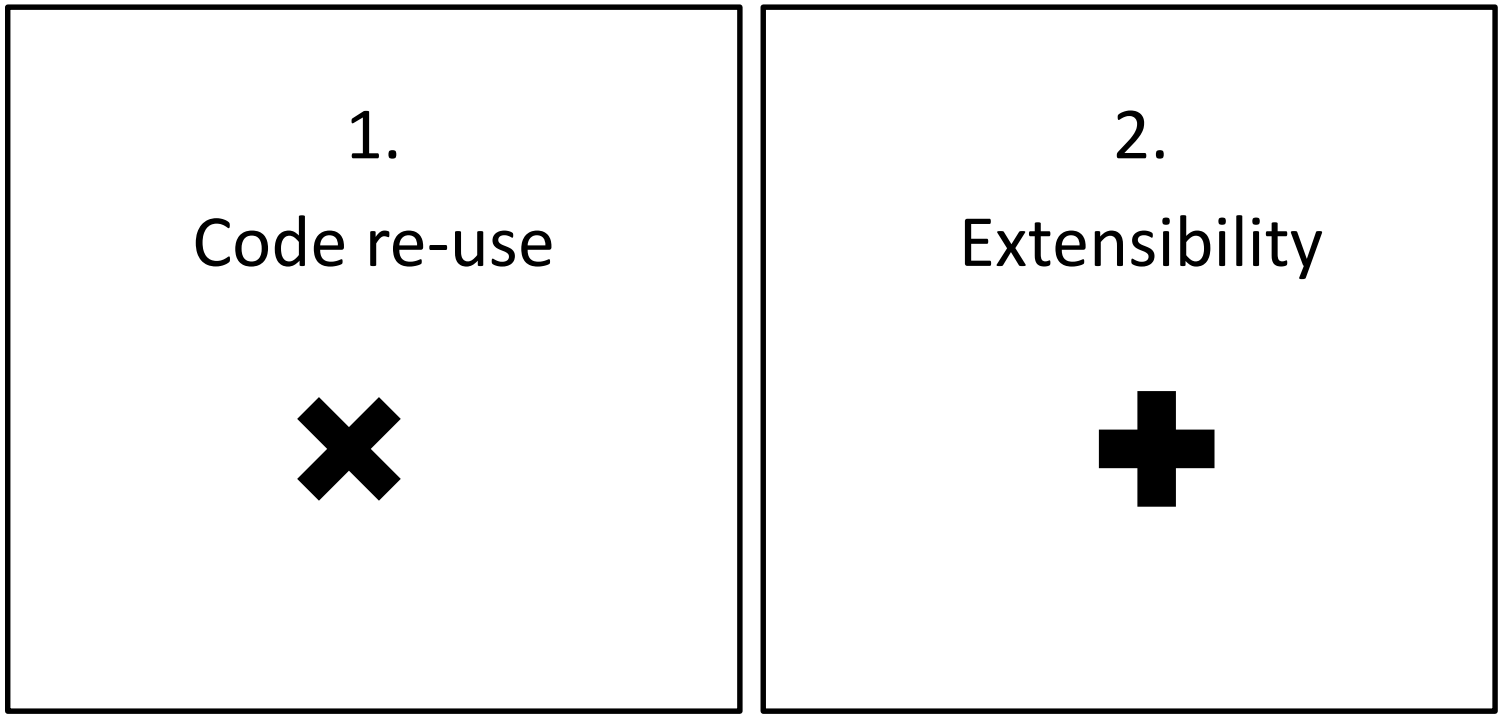
Design Pattern

- What Design Pattern is NOT about
  - Not a specific piece of code that can be copied into our code
  - Not like off-the-shelf tasks/functions/libraries.
  - Not an algorithm.
    - Has clearly defined steps

Design Pattern

- Categories of Design Patterns
  - Creational Pattern
    - Handles object creation mechanism
    - Increases flexibility & reuse of existing code
  - Structural Pattern
    - Assembly of the objects & classes in a larger structure
    - Keeps the structure flexible and efficient
  - Behavioral Pattern
    - Effective communication between objects
    - Assignment of responsibilities between objects

Features of a Good Software Design



Good Software Design Principles

- Encapsulate what varies
- Program to an interface not an implementation
- Favours composition over inheritance

SOLID Principle

- **S**ingle responsibility principle

  A class should have one, and only one, reason to change.

- **O**pen/closed principle

  "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

- **L**iskov substitution principle

  Objects of a superclass shall be replaceable with objects of its subclasses without breaking the application.

SOLID Principle

- **I**nterface segregation principle

  Clients should not be forced to depend upon interfaces that they do not use

- **D**ependency inversion principle

  It consists of two parts:

  1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  2. Abstractions should not depend on details. Details should depend on abstractions.

Examples of Design Patterns

- Singleton Pattern –
  - Restricts the instantiation of a class to one object
- Factory Pattern –
  - Provides an interface for creating families of related or dependent objects and specifies a policy for how it creates
- Observer Pattern –
  - When one object changes state, all its subscribers are notified & updated automatically

Design Patterns Categories

Creational	Structural	Behavioral
Factory Method	Adapter	Chain of responsibility
Abstract Factory	Bridge	Command
Builder	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

- Visitor

Design Pattern & UVM

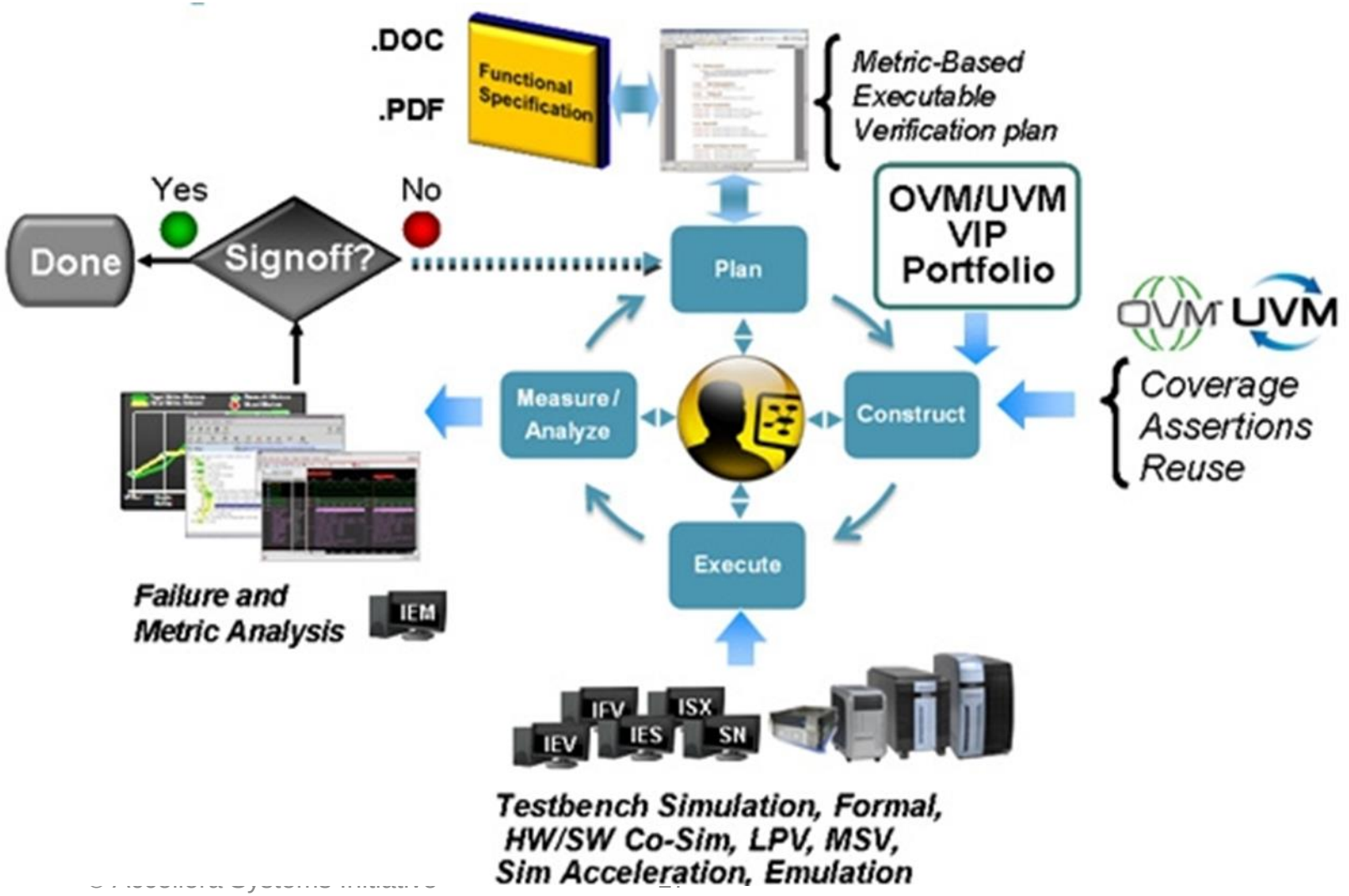
- UVM (Universal Verification Methodology) / OVM (Open Verification Methodology) is based on CRV (Constrained Random Verification) approach.

- UVM applies Software Best Practices like –
  - SOLID principles
  - OOPs features
  - Various combinations of Design Patterns
    - To provide a workable solution for common problems (blueprint)

Design Pattern UVM Applications

Design Patterns	UVM use cases
Factory Method	Factory mechanism
Abstract Factory	Polymorphic Interface
Singleton	UVM pool, UVM resource pool
Composite	UVM Component Hierarchy; UVM Sequence Library
Façade	TLM ports
Command	UVM sequence item
Adapter	UVM reg adapter
Bridge	UVM driver and UVM sequencer
Observer	UVM subscriber
Template Method	UVM transaction
Strategy	UVM sequence, UVM driver
Mediator	UVM virtual sequencer

Metric Driven Verification (MDV) @ Block, Subsystem & SoC-level



Metric Driven Verification Phases

- MDV phases
  - Verification planning
  - Verification environment development
  - Stimulus development
  - Execution
  - Coverage closure

Verification Planning (1/5)

- Strategy development for DUT verification.
  - Verification approaches like Constrained random/Directed testcases/Mix of both.
  - Testbench architecture
  - Vertical/Horizontal/Diagonal re-use
    - Re-use of external and internal VIPs/UVCs/Sequence library/Tests
  - Block/Sub-system/Chip-level approaches

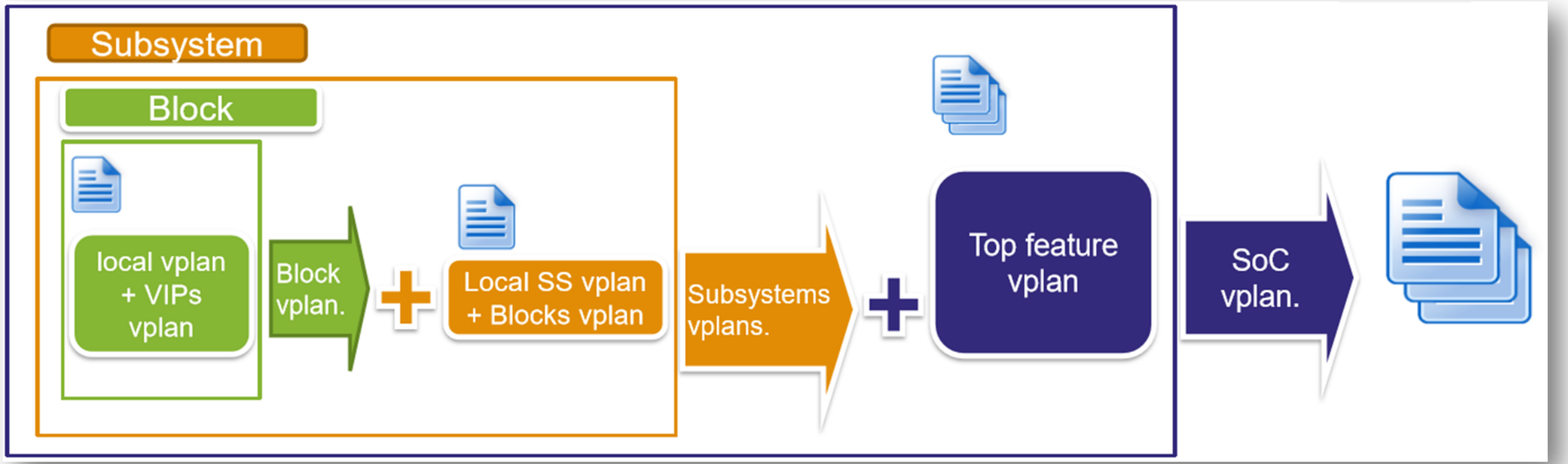
Verification Planning (2/5)

- Verification Plan Development
  - ALPHA
    - Basic integration tests, covering clocks, resets, registers and memory interfaces + reviewed Verification plan
  - BETA
    - All major functionality tested > 90%, Verification reports delivered as proof
    - All integration tests passing at sub-system level should be delivered
    - I/O of sub-system Coverage exclusion files provided with reasons of exclusion
  - FINAL
    - Fully verified Block/Subsystem/SoC with verification plan reports to support
    - Exceptions for code coverage documented with reasons + checklist

Verification Planning (3/5)

- HVP Development

SoC



Verification Planning (4/5)

- Testbench Architecture Development
  - Vertical, Horizontal & Diagonal Reuse
    - **Horizontal reuse** – one SOC to derivatives

    Horizontal typically means using a verification component in a different system or project but at roughly the same level of abstraction and with the same functional role

    - **Vertical reuse** – IP to SOC

    Vertical reuse means using a verification component in a different hierarchy level, usually with an implied change of role

    - **Diagonal reuse** – various level of abstractions

    (Simulation, Emulation, FPGA, Post Silicon)

Verification Planning (5/5)

- Testbench Architecture Development
  - Block, Sub-system & SoC Level Testbench
    - The environment comprised of:
      - VIPs (External/Internal),
      - SV test components,
      - SV Assertions,
      - UVM Components,
      - C test-based infrastructure to accomplish the Verification goal.
    - The generic TB components:
      - BFMs for the bus interfaces like AXI4 interfaces,
      - Can be configurable as masters or slaves.

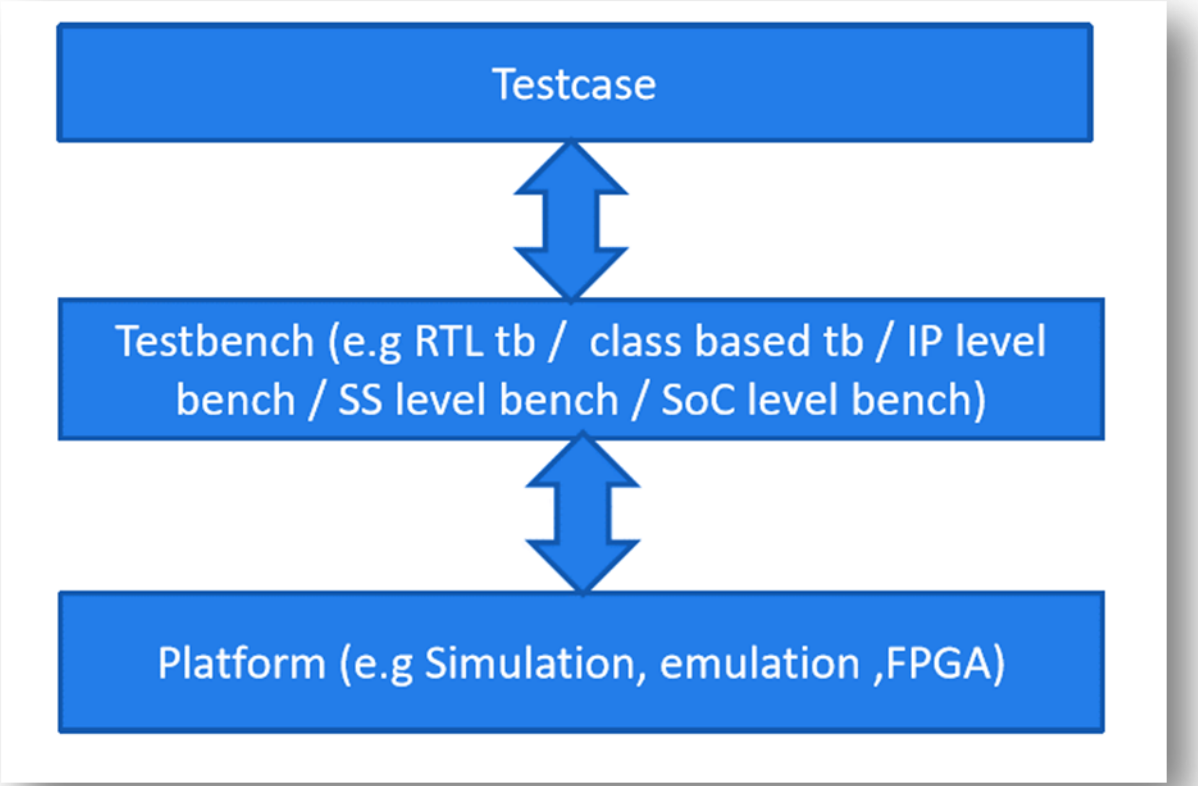
Verification Environment Development (1/9)

- Bottom-up development strategic view (Vertical re-use approach)
  - Block level -> Sub-system level -> Top-level

- Key elements
  - Testcase reusability
  - Effective classification of functions modularity
  - Generic Testbench components application
  - UVCs to support performance related parameters
  - Sign-off metrics implementation

Verification Environment Development (2/9)

- Testcase re-usability



Verification Environment Development (3/9)

- Effective classification of functions modularity
  - Good software design principles/Design Patterns application
  - Immutable functions (fx__ functions)
    - Main thread that runs the simulation – fixed in nature
    - These functions/tasks do not change for different Testbench and/or platforms.
      - Clock setup
      - IP configuration
      - Functional testing
      - Reporting

Verification Environment Development (4/9)

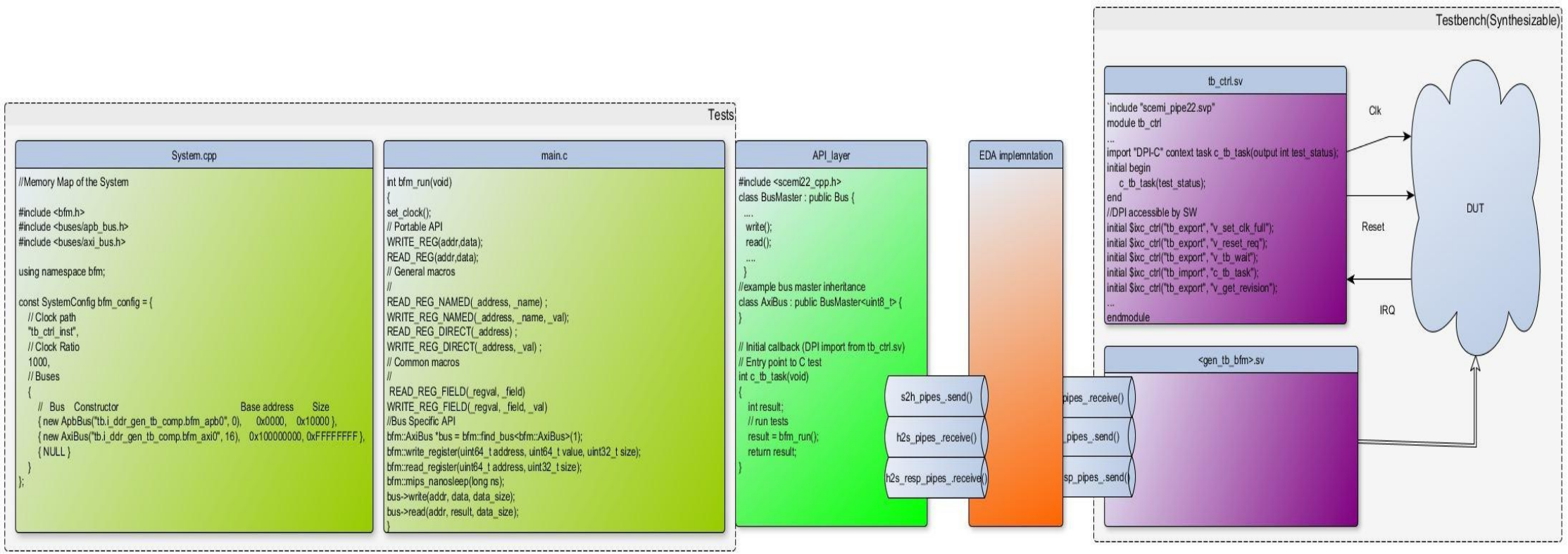
- Effective classification of functions modularity
  - Mutable functions (if__ functions)
    - These are also called interface functions
    - Helps to develop the immutable functions
    - Can be redefined based on the selected TB and/or platform
      - These interface functions helps desired behavior
        - » Clock set up task –
          - IP level – toggling a signal
          - SoC level – configuring the PLL
        - » TB Driver/Monitor calls to global or common functions
          - REG_WRITE
          - REG_READ

Verification Environment Development (5/9)

- Generic Testbench components application
  - SCEMI transactors
    - Supports multiple platforms
      - Simulation
      - Emulation
      - FPGA
  - Can be implanted early in the project cycle.
  - Maximum stimulus code re-use.
  - Generic TB components (different protocol variants) – AXI4/IMG etc.
    - Master
    - Slave (with memory support)

Generic transactor testbench structure

Generic transactor testbench structure



Generic transactor testbench structure

- Figure depicts a typical generic testbench architecture testbench architecture illustrating the flow of transactions from testcase (C/C++ / SV) to Scemi pipes and ultimately to a synthesizable transactor (purple) which fetches transaction from Scemi pipe and drives the test environment .

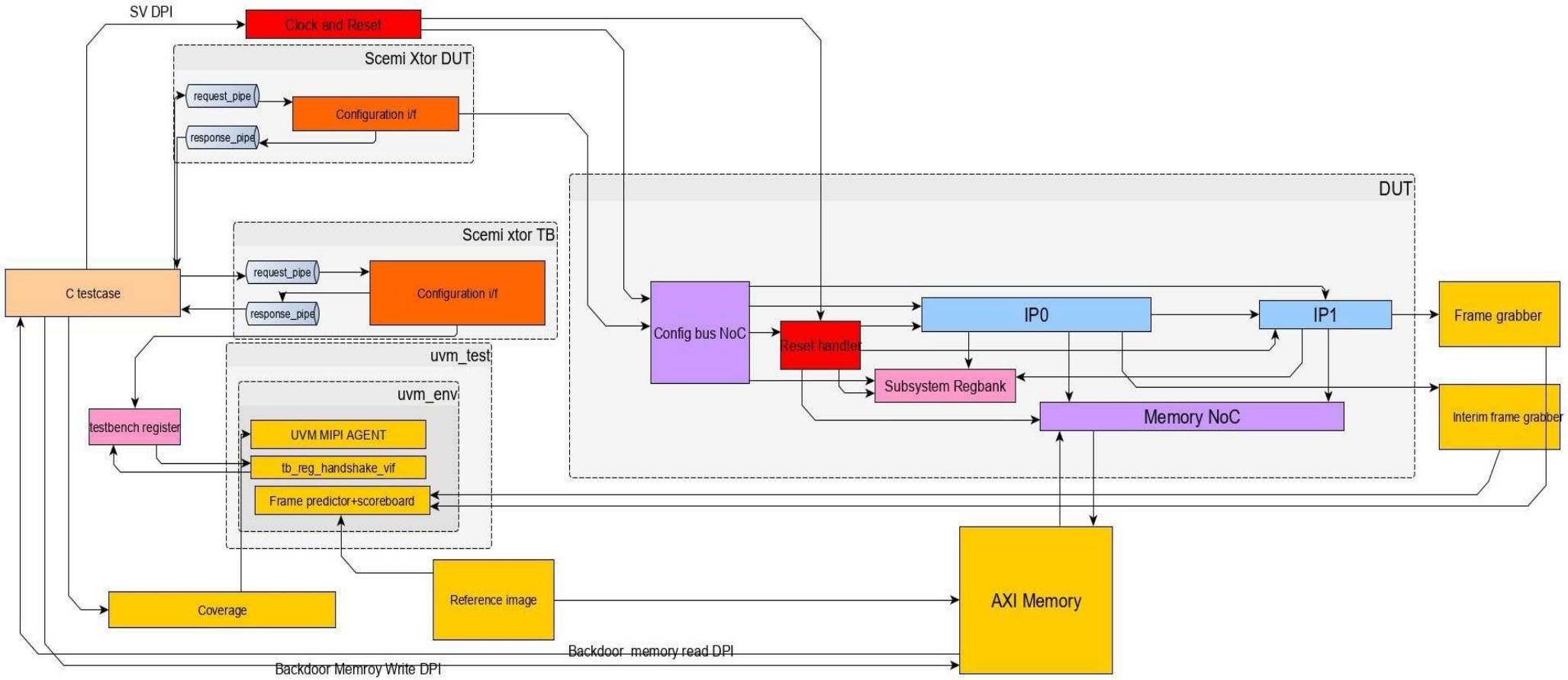
- These scemi transactor based testbench contains an instance of a testbench control module which imports a DPI hdl2c() which is invoked within an initial block , similar to having a run_test() for accessing UVM test form testbench. The hdl2c() call is blocking and will transfers the execution thread control from HDL testbench side to C and executes the test sequence. C to HDL synchronizations are controlled via DPI or polling for certain status flags from a testbench register.

Generic transactor testbench structure

- Immutable functions are set of process like for e.g. programming an IP for a certain mode of operation and this behavior will not change regardless of the platform or whether testing at IP level or system level.

- A Variable function or hook function can morph its behavior based on platform or testbench. For example as a preamble to programming the IP one may choose to enable the clocks and bring the system out of reset, this process may vary depending on the testbench as at IP level this can be a signal connected straight to the I/Os of the IP, although at the subsystem level there might be a clock gate and some external clock controller register needs configuring to enable the clock to IP . Exposing these hook function will provide the flexibility to adapt the action for a given platform.

Generic re-usable testbench



Generic re-usable testbench

- Key:
  - Light Blue: IPs
  - Red: Clock and Reset Blocks
  - Orange: Configuration Interfaces (Synthesizable SCEMI BFMs) Yellow: Testbench elements
  - Pink: Register Banks
  - Peach: C testcases
  - Purple: interconnect

- Figure depicts block diagram of Subsystem verification architecture. The IP in case of this particular Image Processing Subsystem would be delivering the IP level tests and these tests will be re-used at subsystem and SoC level with some modifications, like commenting the commands which are related to IP's internal data generator since we will be using the external imager in form of UVC/models. We also commented the 'test models' related commands because we will not be using the IP delivered test models at the Subsystem, We had to add some additional commands required to access the testbench registers used for the synchronization between the software and the external imager sequence (uvm) in case of Subsystem and SoC Level.

- Additional verification code was to configure various subcomponents and backdoor access to IPs eg descriptors. A C framework was used for the backdoor access in the testbench. Also, C testcases are written in a way that the CPU can run them, however during early integration testing, not all testcases have the CPU live. C part was written with consideration of generic test bench approach so that same tests can be used for FPGA/Emulator with little modifications.

Generic re-usable testbench

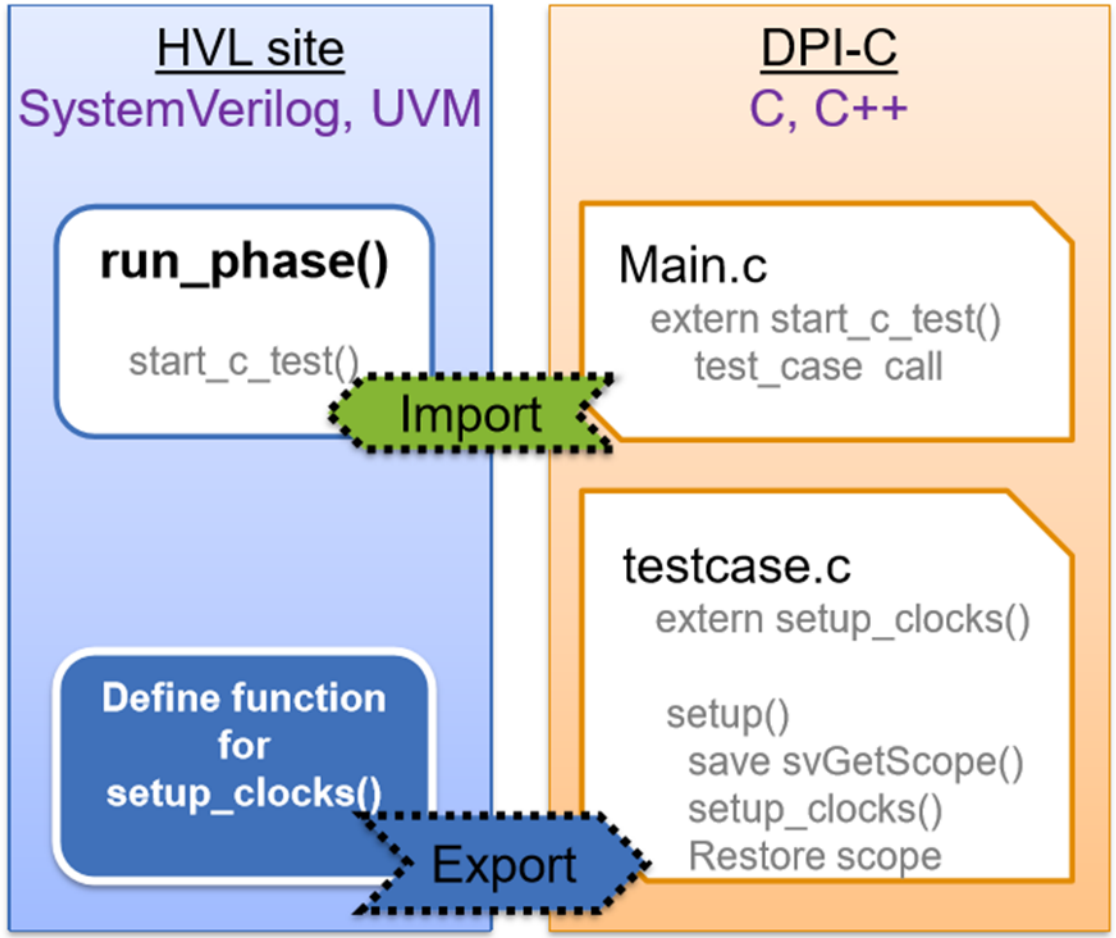
- In order to provide a common verification environment across different SoC, subsystems and different platforms some generic reusable testbench components have been developed The standard generic components are synthesizable BFMs for AXI4 interfaces, Company Standard Interfaces.

- These BFMs are bus masters and can drive standard slave interfaces that are compliant with AXI4 and Company Standard Interface. Additional reusable components developed were for clock and reset generation (tb_ctrl),

- In order to re-use tests and test sequences, the Subsystem test sequence are layered where the platform, the testbench and the subsystem specific code are structured in layers. This allows seamless porting to different platforms like an FPGA or emulator, ports to a different testbench as well, without expecting any changes to be made to the tests

Verification Environment Development (6/9)

- UVM <-> C communication
  - SystemVerilog is the master of the test case, but the C performs all the scenario steps/behavior.
  - C test needs also SV task mainly for clocking timing, reset interface control and all register access transactions driven by an SV BFM/transactor.
  - Another mechanism is using some GPIO ports on the SoC connected to "registers" in the testbench to monitor SoC white box signal through signal mirroring.



Verification Environment Development (7/9)

- UVCs to support performance & connectivity
  - SystemVerilog assertions
    - Can be used to predict maximum latency
    - Can be used to verify the system clock frequencies/duty cycles.
    - Reset tree connectivity

Verification Environment Development (8/9)

- Sign-off metrics implementation
  - Assertion coverage
  - Group coverage
  - Sub-plan coverage
  - VSIF vs HVP mapping
  - Code coverage (Implicit)

Verification Environment Development (9/9)

- SoC Level Specific components
  - TB interconnect
  - API based TB regbank generation
  - Tb_MFIO configuration via APIs
  - DDR BFM model
  - Memory backdoor APIs
  - Dummy view support
  - Efuse control
  - Bootstrap control

Stimulus Development (1/3)

- Choice of Stimulus
  - C based test for
    - maximum vertical re-use
      - IP
      - Subsystem
      - SoC level
    - Across platforms
      - Simulation
      - Emulation
      - FPGA
  - UVM based fully random testcases ideal for extensive IP verification.

Stimulus Development (2/3)

- Guidelines
  - Register macros
    - Address must come from header-file defines.
  - Self-checking mechanism.
    - Data integrity checks with the test, assertion or coverage.
  - No use of hard code values.
  - Addresses should be passed to register access functions as base_address + any cumulative offsets.
  - Each testcase checks all the features that it has been mapped to in the HVP.
  - Correct choice of if__ and fx__ functions.
  - No printf calls in code without being wrapped in an if__ function.

Stimulus Development (2/3)

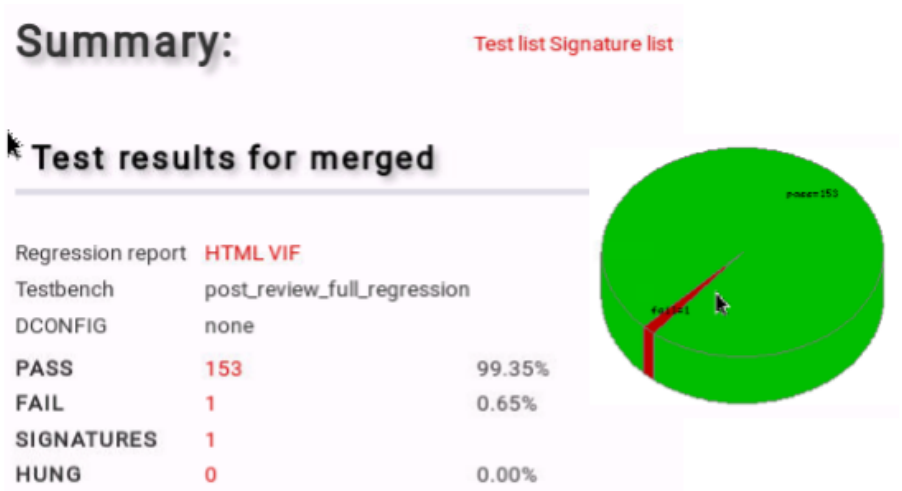
- Promotability mechanism
  - Test promotion
    - Promoted
      - Reusable integration test (toggling boundaries, Does not necessarily covering key features).
      - A promoted test should have.
        - » The attribute "top_level=1" in the Subsystem vplan.
        - » The correct "milestone = alpha/beta or final" attribute in its vplan.
    - Not Promoted
      - Exhaustive IP level test (The functional testcases exploring the features of the IP and corner cases).
      - Interoperability tests of IPs in the Subsystem.

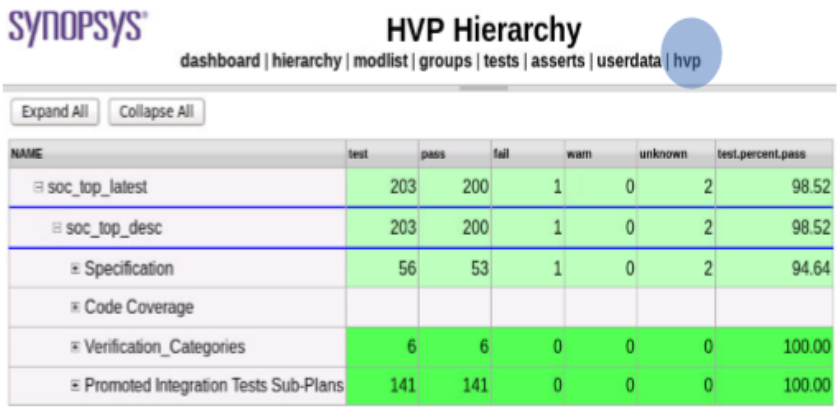
Execution

- Running the tests in order to cover defined scenarios for different phases –
  - ALPHA
  - BETA
  - BETA+V
  - FINAL

Execution

- Regression is achieved using Jenkins
  - Weekly runs for Subsystems and top
  - Visibility for all the stakeholders
  - Automatic coverage merge.
  - Automatic reporting (a coverage dashboard mapped to the verification plan)

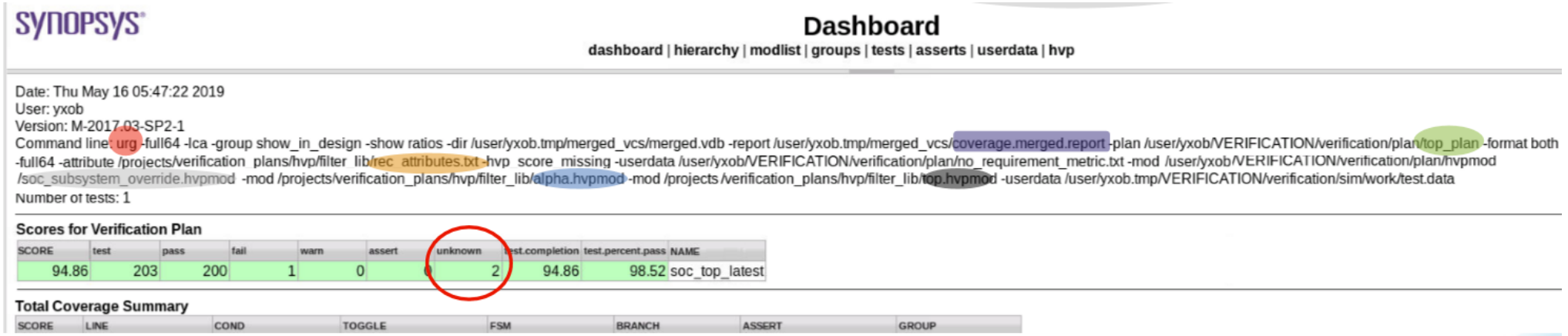




NAME	test	pass	fail	warn	unknown	test.percent.pass
soc_top_latest	203	200	1	0	2	98.52
soc_top_desc	203	200	1	0	2	98.52
Specification	56	53	1	0	2	94.64
Code Coverage						
Verification_Categories	6	6	0	0	0	100.00
Promoted Integration Tests Sub-Plans	141	141	0	0	0	100.00

Coverage Closure

- Analysis and Tuning
  - The RTL code coverage get instrumented using a configuration file.
  - The verification plan is merged with the coverage database for annotation.
  - Several filter can be applied to fine tune the coverage results based on
    - The project milestone (alpha, beta, final),
    - The promoted test from subsystem to top (top_level attribute is used).


SYNOPSYS

Dashboard

dashboard | hierarchy | modlist | groups | tests | asserts | userdata | hvp

Date: Thu May 16 05:47:22 2019
User: yxob
Version: M-2017.03-SP2-1
Command line: urg -full64 -lca -group show_in_design -show ratios -dir /user/yxob.tmp/merged_vcs/merged.vdb -report /user/yxob.tmp/merged_vcs/coverage.merged.report -plan /user/yxob/VERIFICATION/verification/plan/top_plan -format both -full64 -attribute /projects/verification_plans/hvp/filter_lib/rec_attributes.txt -hvp score_missing -userdata /user/yxob/VERIFICATION/verification/plan/no_requirement_metric.txt -mod /user/yxob/VERIFICATION/verification/plan/hvpmod /soc_subsystem_override.hvpmod -mod /projects/verification_plans/hvp/filter_lib/alpha.hvpmod -mod /projects/verification_plans/hvp/filter_lib/top.hvpmod -userdata /user/yxob.tmp/VERIFICATION/verification/sim/work/test.data
Number of tests: 1

Scores for Verification Plan

SCORE	test	pass	fail	warn	assert	unknown	test.completion	test.percent.pass	NAME
94.86	203	200	1	0	0	2	94.86	98.52	soc_top_latest

Total Coverage Summary

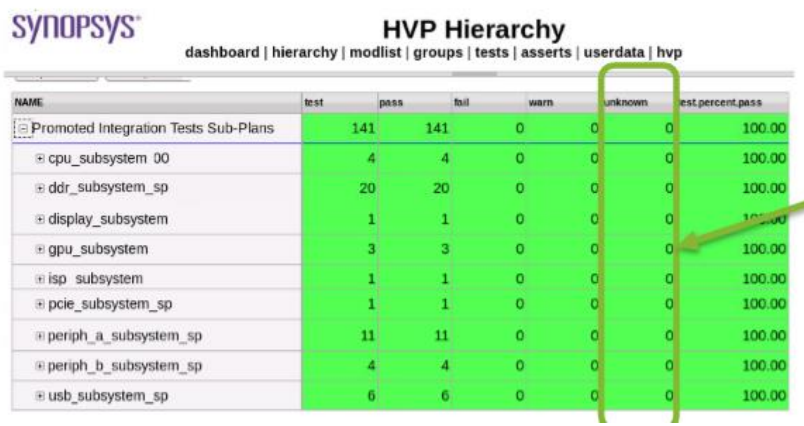
SCORE	LINE	COND	TOGGLE	FSM	BRANCH	ASSERT	GROUP

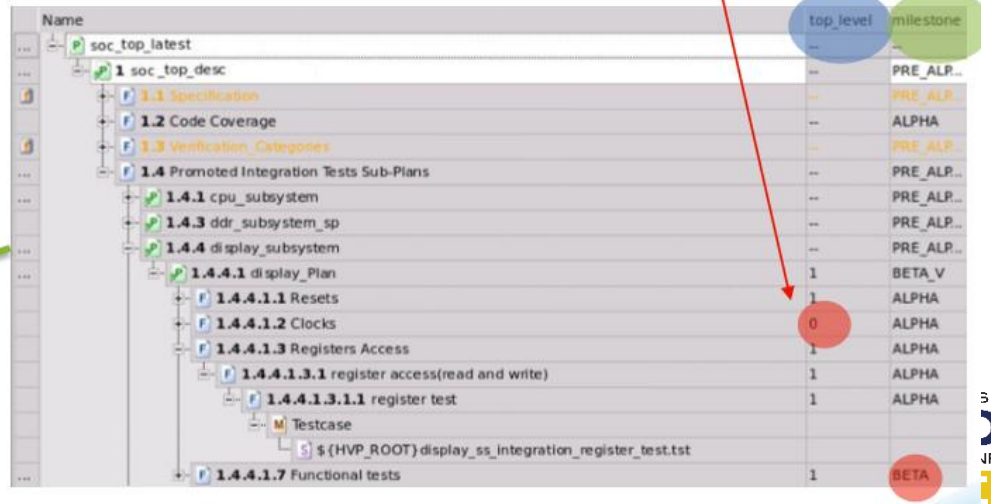
Coverage Closure

- Coverage tuning – override
  - As part of the coverage analysis, verification plan override file can be implemented and reviewed later:
    - It's as a kind of exclusion file.
    - Helps to make the dashboard clean up without waiting for a new subsystem release.

```
override display_override;

//DISPLAY//
soc_top_latest.soc_top_desc."Promoted Integration Tests Sub-Plans".display_subsystem.display_verif_plan.Clocks."Frequencies ".top_level = 0;
soc_top_latest.soc_top_desc."Promoted Integration Tests Sub-Plans".display_subsystem.display_verif_plan.Resets.pdci_reset.top_level = 0;
soc_top_latest.soc_top_desc."Promoted Integration Tests Sub-Plans".display_subsystem.display_verif_plan.Resets."Asynchronous active low primary reset".top_level =
```

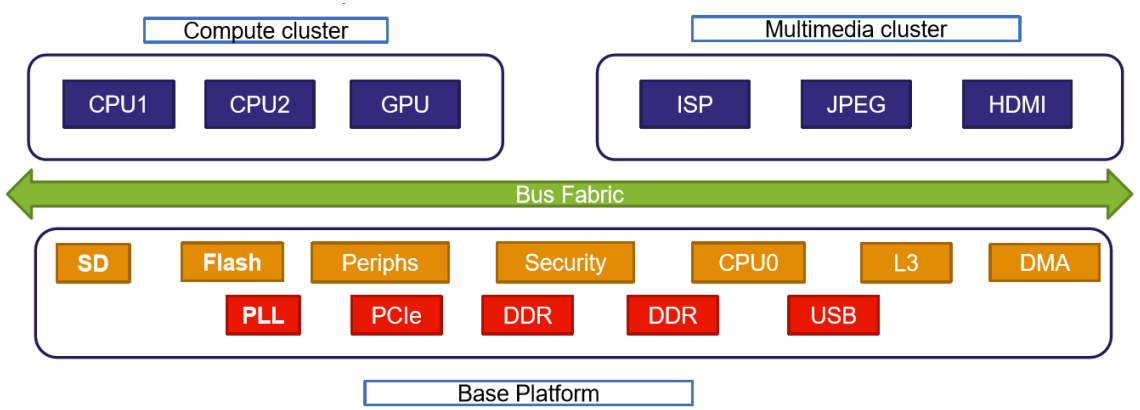




Case Study – Solaris (1/7)

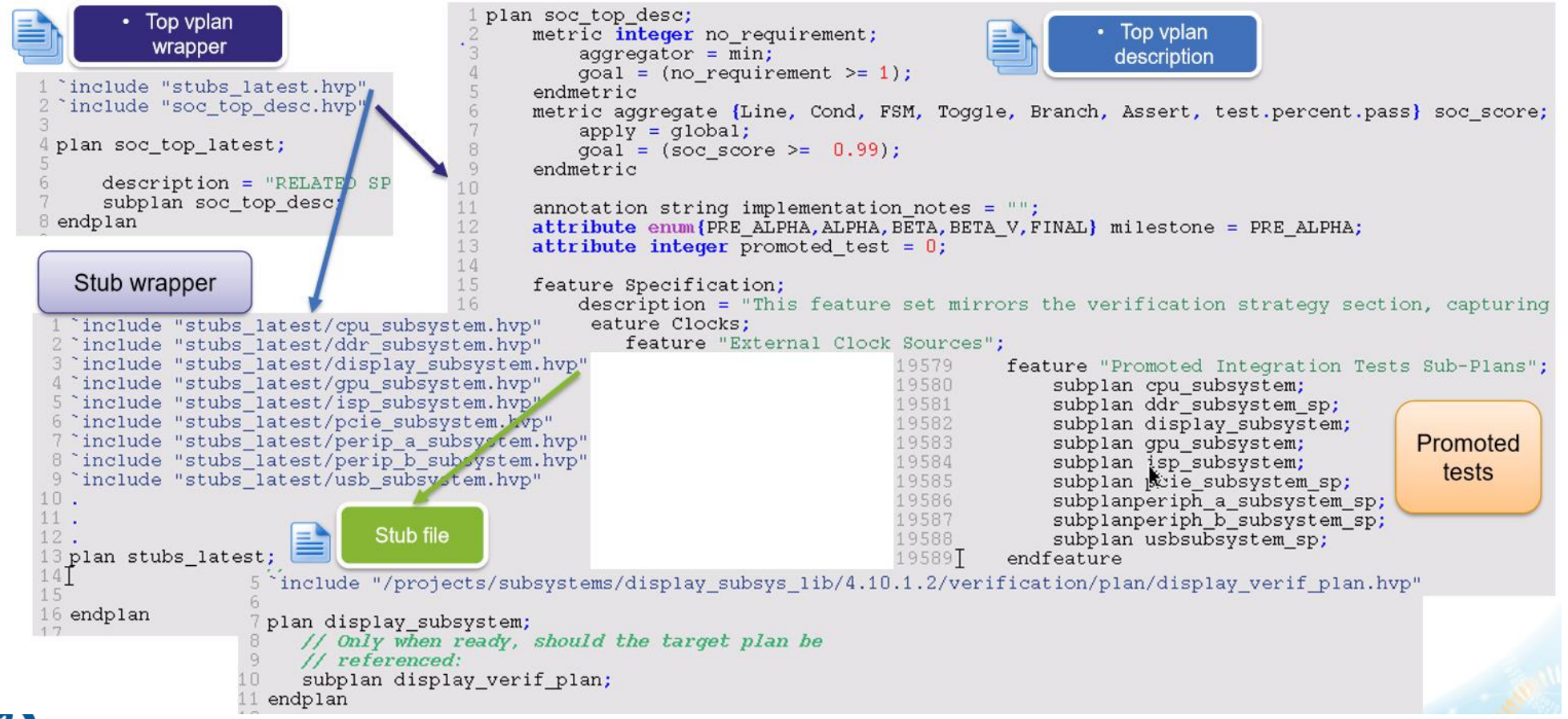
- Compute cluster
  - 2 CPU with Quad-core 64-bit processors
  - High end GPU
  - Multi-core DSP
- Multimedia cluster
  - ISP, H.265/H.264/JPEG/Multi standard encode/decode
- Base Platform
  - Booting/Housekeeping CPU
  - High speed NoC
  - Hi-speed/Low-speed peripherals
  - Security subsystem
- Total 29 different complex subsystems

- SoC High Level Block Diagram



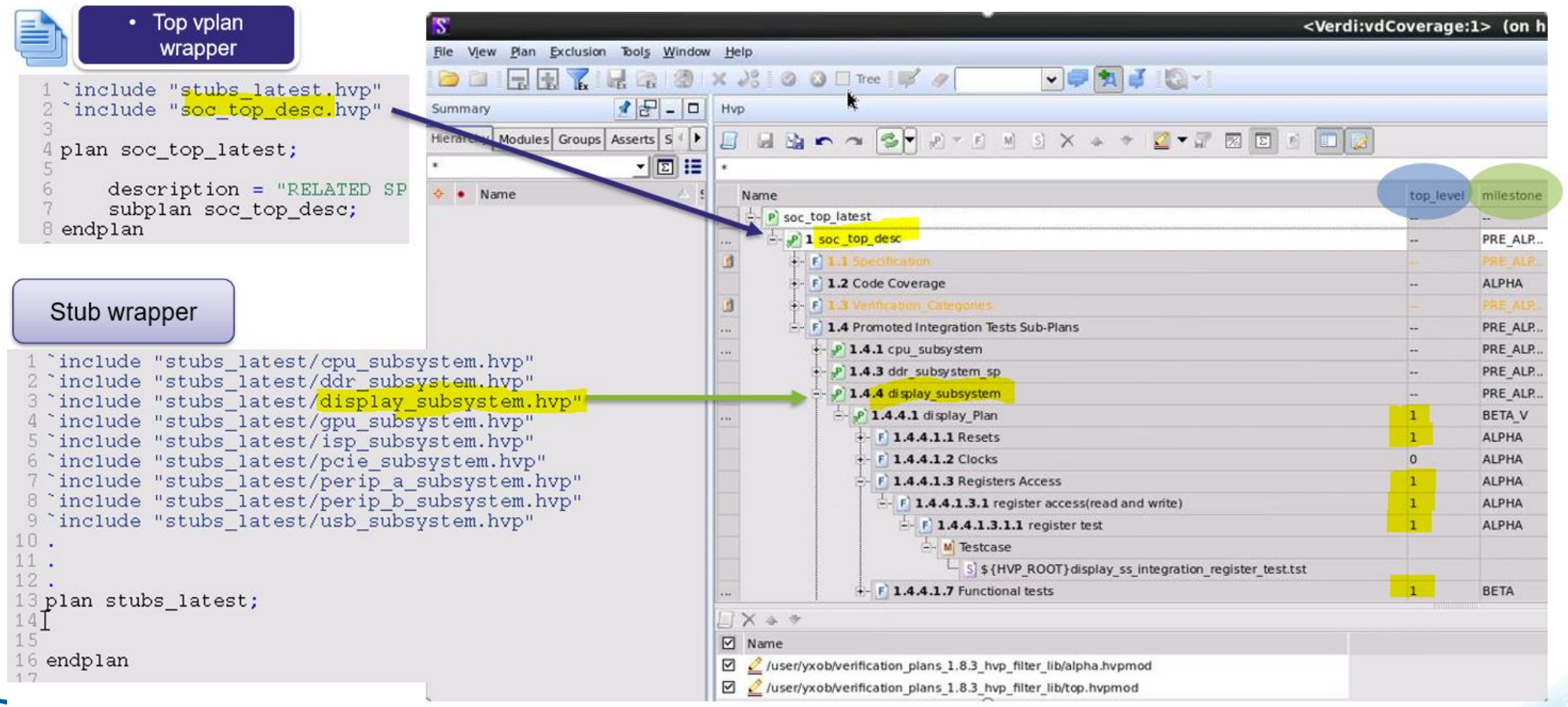
Case Study – Solaris (2/7)

- Hierarchical Verification Plan



Case Study – Solaris (3/7)

- Test promotion from sub-system to top-level

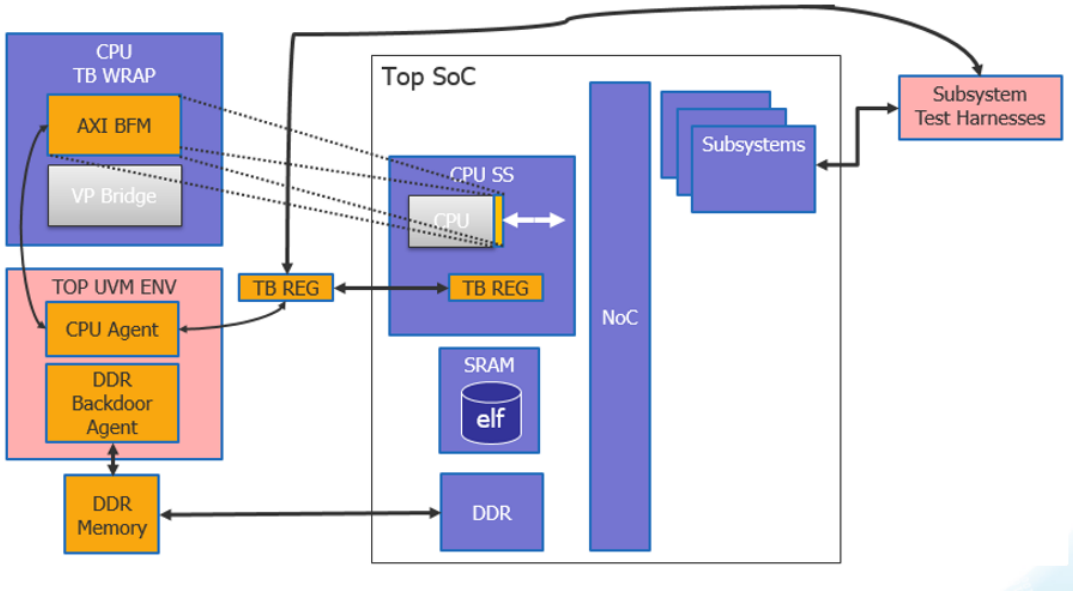


Case Study – Solaris (4/7)

- SoC Verification
  - Simulation, Emulation & FPGA
  - CPU modes
    - RTL mode – Booting scenarios
    - BFM mode – ALPHA milestone
    - VP mode – BETA milestone
  - STUB modeling – Dummy views
  - Sub-system Test harnesses
  - Regression
    - VSIF test list structure – classified based on the simulation time.
    - Jenkins – weekly, nightly

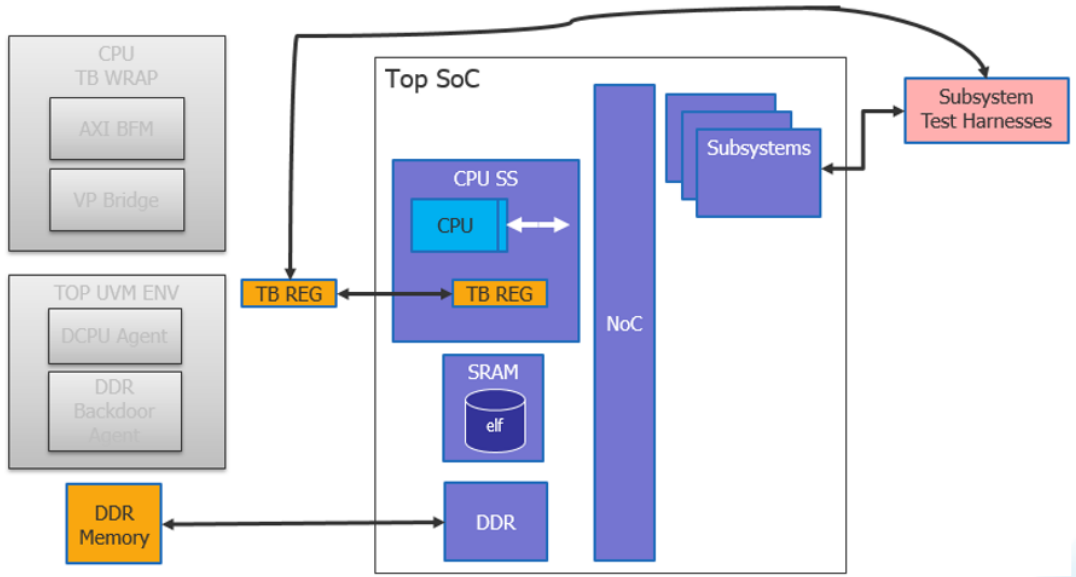
Case Study – Solaris (5/7)

- CPU mode – BFM
  - The (RTL) CPU shall be held, permanently, in reset; instead an AXI VIP shall drive the main memory interface of the CPU.
  - The UVM CPU sequencer shall be active , which shall call write/read functions to drive transfers on the CPU memory bus interface.



Case Study – Solaris (6/7)

- CPU mode - RTL
  - The (RTL) CPU shall drive the transfers.
  - The C-based test cases shall be compiled, and the resultant elf file shall be loaded into the SoC ROM/SRAM.



Case Study – Solaris (7/7)

- In-progress Regression result

VSIF: solaris_top_rtl_freeze.vsif - Finished

Start Date:	Fri Aug 9 17:26:11 BST 2019
Report Date:	Mon Aug 12 10:11:33 BST 2019
Report User:	xbw
Work:	/projects/solaris/verification/jenkins/workspace/HwVsif_Solaris_top_full_vcs_TESTS_full/regression_results/vsif_magic/current
VSIF:	solaris_top_rtl_freeze.vsif - processed
UGE Session:	laris_top_full_vcs_T05 772239
Domains:	merged

Summary: Test list Signature list

Test results for merged

Regression report HTML VIF

Testbench	solaris_top_rtl_freeze	
DCONFIG	none	
PASS	196	48.64%
FAIL	206	51.12%
SIGNATURES	20	
HUNG	1	0.25%
RUNNING	0	0.00%

Summary

- We discussed:
  - **Design Patterns**
    - What design patterns are and what not?
    - Different types & application
    - Design patterns role in advanced functional verification arena
  - **MDV flow and it's different associated phases**
    - Block, Sub-system and SoC level hierarchical approaches
    - Vertical, horizontal and diagonal re-usability
  - **Sondrel case study of a handling a big SoC**
    - 2 CPU (quad core 64-bit each), CPU for housekeeping, Multicore DSP, 29 subsystems
    - Test harness reusability from subsystem to SoC level
    - Different SoC Verification modes i.e. RTL, BFM, VP to support different project milestones.
      - Stimulus reusability across subsystem, top-level (simulation/emulation)
      - VSIF regression results/Sign-off

Questions

Thank You