Wiretap your SoC

Why scattering Verification IPs throughout your design is a smart thing to do

Avidan Efody,
Verification Architect

Agenda

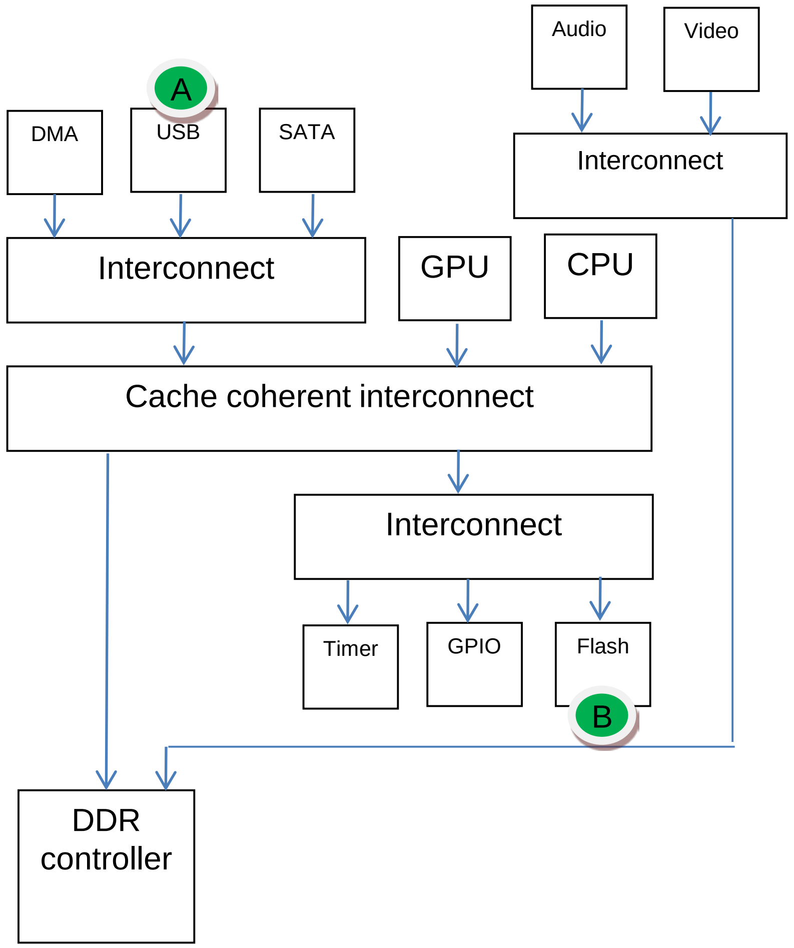
- Introduction
  - Common SoC verification challenges
  - How mass VIP deployment can help
- Problems with mass VIP deployment
- Mass VIP deployment solution requirements
- Mass VIP deployment solution overview
- Summary

Common SoC verification challenges

- System level debug
  - Follow transactions through the system
- Integration validation
  - IP to IP connectivity
  - Address map correctness
  - Supported protocol options
- Performance analysis
  - Validate bandwidth and latency requirements
  - Debug bottlenecks

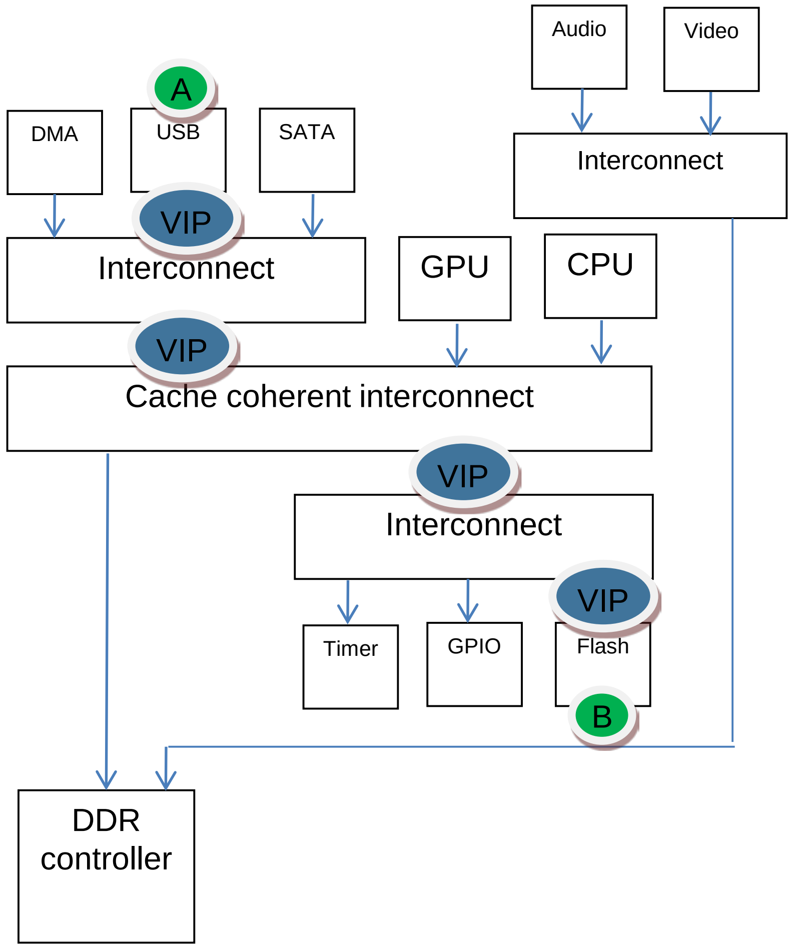
System level debug

- A typical scenario:
  - A read-modify-write test from point A to point B fails
  - Where did the data get corrupted?
- Requires a transaction to be tracked across bridges/interconnects
  - If done at signal level can be very time consuming
  - VIPs simplify the task by raising the abstraction level
- The more VIPs placed along the way the easier



System level debug

- A typical scenario:
  - A read-modify-write test from point A to point B fails
  - Where did the data get corrupted?
- Requires a transaction to be tracked across bridges/interconnects
  - If done at signal level can be very time consuming
  - VIPs simplify the task by raising the abstraction level
- The more VIPs placed along the way the easier



Integration validation

- Make sure all IPs are properly connected
  - Iterate over all possible paths, address segments, protocol options
  - Failures could happen anywhere
- VIPs are required on almost any path

Performance analysis

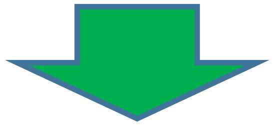
- Make sure SoC meets IPs bandwidth and latency requirements
  - Bandwidth/latency need to be measured from various sources
  - Bottlenecks along the way need to be detected
  - The more sampling points along the way, the better
- VIPs could be used to provide bandwidth/latency information on standard interfaces

Problems with mass VIP deployment

- Creation effort
  - Too much code to write and maintain
  - Must be kept aligned with RTL changes
- Integration effort & risk
  - Adding VIPs to existing testbench can cause new regression failures
  - A VIP added to improve debug can make testbench non-operational
- Performance penalty
  - VIPs implement complex state machines, coverage, assertions
  - They do have a cost in performance

Creation effort

- Each new VIP added to the testbench requires
  - Signal connection code
  - Additional instantiations
  - Additional configuration
- A repetitive code that needs to be maintained



- Requirement:
  - Code to connect VIPs and to configure them should be auto generated
  - Best source for generation is RTL itself
    - Always exists
    - No need to maintain another format

Problems with mass VIP deployment

Integration effort & risk

- Adding VIPs in might trigger various errors
  - Compilation/elaboration errors
  - OVM/UVM/VMM/xVM errors
  - VIP false alarms firing
- Might make a regression non-operational

- Requirement:
  - All code to instantiate VIPs should be external to testbench
  - It should be possible to run without any additional VIPs
    - User could make the choice at run time

Performance penalty

- Performance penalty
  - Instantiating VIPs takes a high toll on performance
- But…
  - Additional debug information always costs something in performance
    - Common examples are RTL signal visibility, or message verbosity
- Requirement
  - Allow users to trade off some performance for VIP debug information
    - During regression – maximize performance
    - During debug – trade off some performance for visibility on interesting interfaces
  - => It should be possible to turn VIPs on and off according to the task at hand
    - Preferably at run-time and without any re-compilation/elaboration

Mass VIP deployment solution requirements

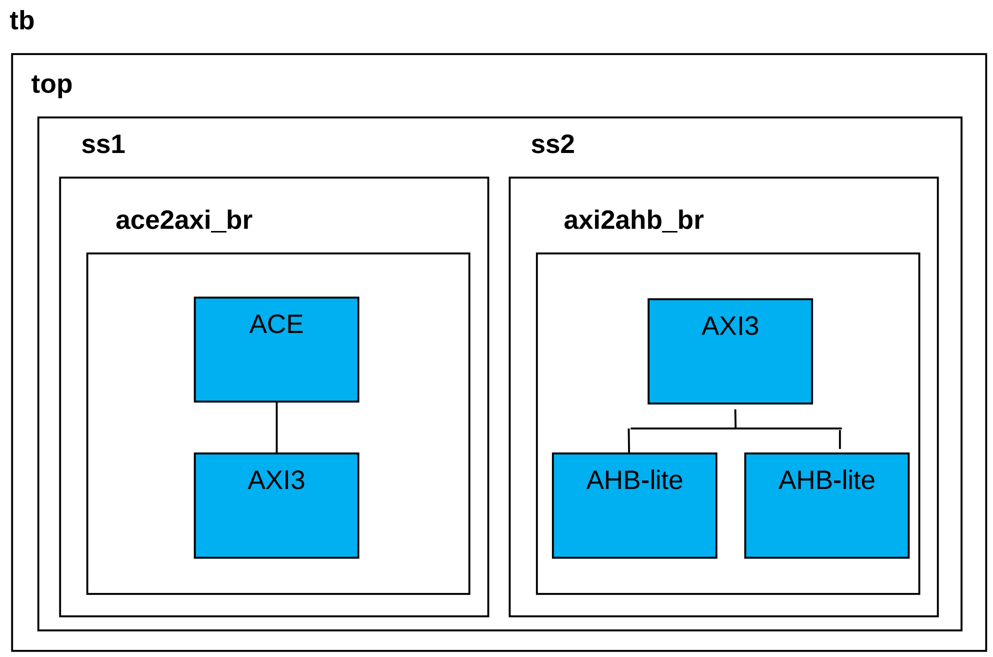
#	Requirement	Reason
1	It should be possible to generate almost all solution code automatically. Source for generation should be RTL itself.	Avoid creation and maintenance effort
2	It should be possible to have the solution code run with any testbench/DUT without modifications to testbench/DUT code. User could always choose to switch back to original version without and VIPs instantiated.	Avoid integration effort and risk
3	It should be possible to disable all or a subset of the VIPs, preferably at fast turn-around time	Control performance penalty introduced by VIPs
4	It should be possible to easily add support for analysis capabilities such as transaction linkers or scoreboards	

Mass VIP deployment solution assumptions

- The proposed solution makes no assumptions about language/methodolgy
  - Can be any language/methodology
    - VHDL, Verilog, SystemVerilog or e
    - OVM, UVM, VMM, eRM or proprietery
- The proposed solution assumes standard interface signals stick to some naming convention
  - Required for automatic code generation
  - More details in relevant section below
- The proposed solution assumes that a VIP has an API to turn it off
  - If that is not the case, a VIP might be turned off by holding it in reset

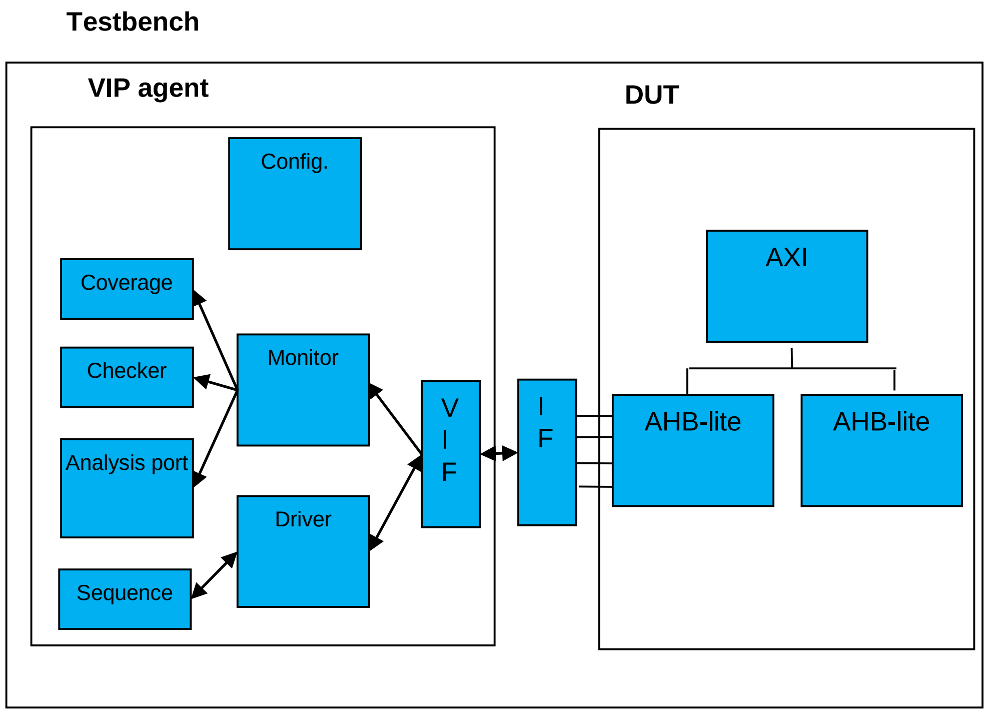
Example description (1)

- Example code is shown in SV/UVM
- Example code refers to the following SoC
  - tb – might contain any existing testbench code in any language/methodology
  - top – an RTL hierarchy with two AMBA interconnects



Example description (2)

- Undelaying VIP assumed to have a standard SV/UVM structure
  - Made of a uvm_component hierarchy and an SV interface
  - SV interface passed to uvm_component hierarchy via UVM's config_db
  - A configuration object controls the VIP's behavior



Standard interface detection (1)

- Requirement #1:

1	**It should be possible to generate almost all solution code automatically. Source for generation should be RTL itself.**	**Avoid creation and maintenance effort**

- How can we find where are the standard interfaces in RTL???
  - Standard interfaces always define standard signal names
    - (i.e. ARREADY on AXI)
  - We assume users keep those names as a base
    - Then add postfix/prefix
    - Or change upper/lower case
    - (i.e. ARREADY_B or c_arready)
  - This assumption is correct for all off-the-shelf on chip interconnect IPs
    - And probably for most proprietary RTL
    - Otherwise, it can be very confusing to figure out which signal is which

Mass VIP deployment solution

Standard interfaces detection (2)

- Example flow for detecting AXI interfaces in RTL
  - Can be implemented in DPI, PLI or proprietary simulator commands
  - RHS shows the information we get for the example design

Look for signals called [prefix]ARVALID[postfix] or [prefix]arvalid[postfix] in the design.	tb.top.ss1.ace2axi_br.ARVALID_A tb.top.ss1.ace2axi_br.ARVALID_B tb.top.ss2.axi2ahb_br.ARVALID
↓	

For each of the signals found, extract prefix/postfix/case parameters. These will be used to search for other interface signals.	prefix =“”, postfix = “_A”, uppercase
	prefix =“”, postfix = “_B”, uppercase
	prefix = “”, postfix = “”, lowercase

↓	
Check that all obligatory signals exist, and find out which optional signals exist	
↓	
Extract the RTL path for the interface and register it	tb.top.ss1.ace2axi_br tb.top.ss1.ace2axi_br tb.top.ss2.axi2ahb_br
↓	
Check the bus widths for any signals with variable bus width such as ARADDR and register the information	For all: address width = 32, data width = 64

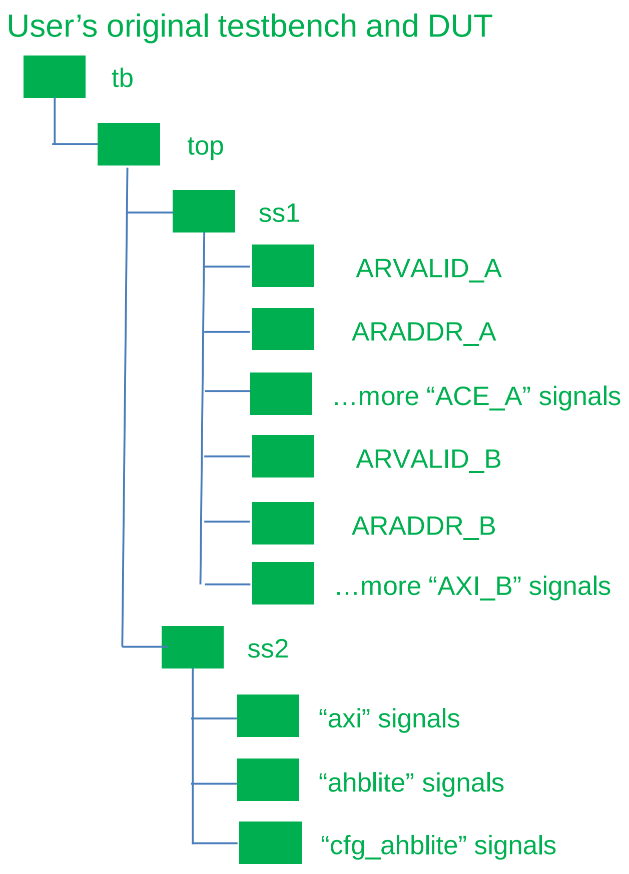
VIP instantiation (1)

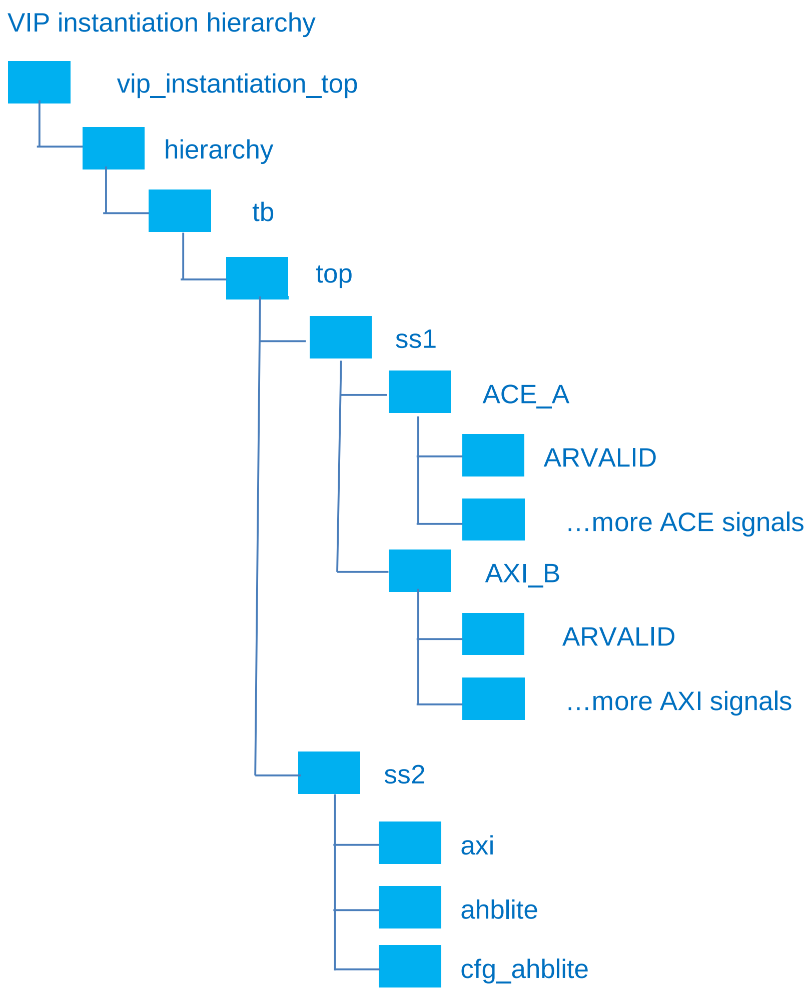
- A VIP should be instantiated per detected interface. This includes:
  - A SystemVerilog interface
    - To connect to the signals
  - An agent uvm_component
- Instance names should:
  - Be created automatically
  - Not collide with each other
  - Be intuitive for users
- The best way is if a VIP instance name is identical to the name of the interface to which it is connected
  - Which can be captured by the following convention
    - [rtl_path],[postfix],[prefix],[case] all obtained from auto-detection

[rtl_path].[prefix]_[protocol+case]_[postfix]

Mass VIP deployment solution

VIP instantiation (2)





VIP instantiation (3)

- The separate hierarchy removes the need for integration, hence meeting requirement #2:

2	**It should be possible to have the solution code run with any testbench/DUT without modifications to testbench/DUT code. User could always choose to switch back to original version without and VIPs instantiated.**	**Avoid integration effort and risk**

- Sample code for the VIP instantiation hierarchy is shown in paper
  - It can be automated based on information extracted from the design

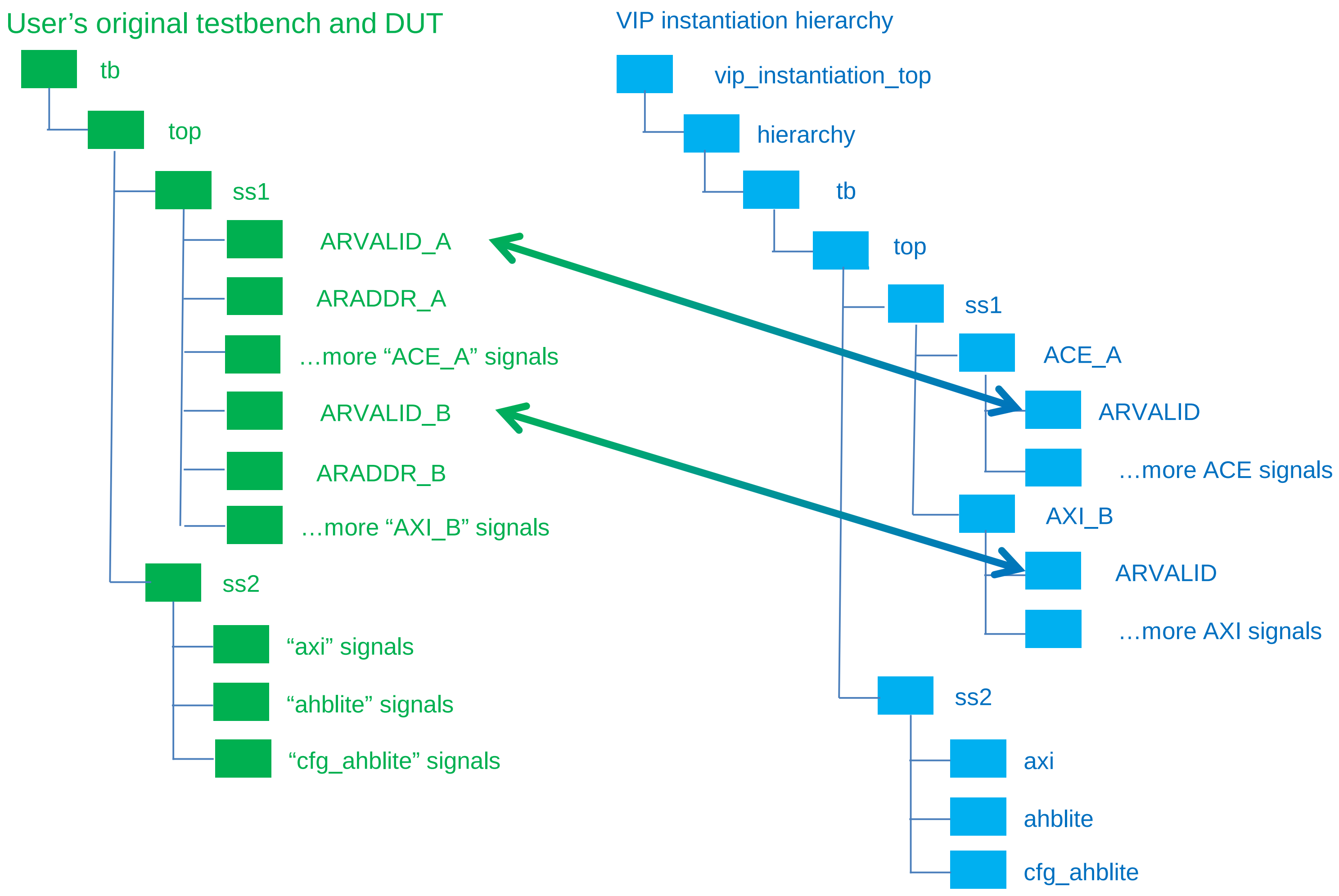
Signals connection (1)

- Given the proposed VIP instantiation hierarchy, connecting the VIP signals can be done using the code below

```
module signal_connections();
// derived from [additional_signals] specified explicitly by the user
assign vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.ACLK = tb.top.ss2.axi2ahb_br.sys_clk;
assign vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.ARESETn = ~tb.top.ss2.axi2ahb_br.sys_reset;

// both LHS and RHS are derived from [rtl_path]
assign vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.AWVALID = tb.top.ss2.axi2ahb_br.awvalid;
assign vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.AWADDR = tb.top.ss2.axi2ahb_br.awaddr;
assign vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.AWLEN = tb.top.ss2.axi2ahb_br.awlen;
assign vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.AWSIZE = tb.top.ss2.axi2ahb_br.awsize;
assign vip_instantiation_top.hierarchy.tb.top.ss2.axi2ahb_br.axi.AWBURST = tb.top.ss2.axi2ahb_br.awburst;
// more signal assignments...
endmodule
```

Signals connection (2)



VIP configuration (1)

- Requirement #3:

3	**It should be possible to disable all or a subset of the VIPs, preferably at fast turn-around time**	**Control performance penalty introduced by VIPs**

- VIP agent uvm_components are instantiated in a hierarchy that replicates RTL structure
  - Just like the SystemVerilog interfaces
- It is possible to configure them from the command line or from a test using their full names
  - Which follow the same convention as the SystemVerilog interfaces

VIP configuration (2)

- Configuration fields are placed in the component that instantiates the VIP agent

```
class container_component_5 extends uvm_component;
  `uvm_component_utils(container_component_5)

  on_off_t cfg_ahblite_on_off = Off;
   ahblite_agent cfg_ahblite_agent;

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    begin
      reg signed [4095:0] tmp;

      uvm_config_db #(reg signed [4095:0])::get(this, "",  "cfg_ahblite_on_off", tmp);
      cfg_ahblite_on_off = on_off_t'(tmp);

      // create the agent and configure it
    end
  endfunction

  //code for more interfaces skipped here…
Endclass
```

VIP configuration (3)

- And are then configured from a command line
  - (Or from a test)

```
+uvm_set_config_int=uvm_test_top.vip_instantiation_test.hierarchy.tb.top.ss2.axi2ahb_br,cfg_ahblite_on_off,1
```

Summary

- VIPs are a very useful tool during SoC verification
  - They improve debug
  - They provide checking & coverage
  - And data that could be used for performance analysis
- But deploying them in large scale is often costly
  - Creation and maintenance effort
  - Integration effort and risk
  - Performance penalty
- We tried:
  - To raise awareness to the benefits of VIP mass deployment
  - And, to provide a simple methodology that would address the problems above
- Hope we did well ☺