AXI4 VIP

This VIP was created by Yosef Belyatsky @https://github.com/SkyVerify

For any comments, bugs, issues or questions feel free to contact via yosefbel92@gmail.com

1. Overview

AXI4 VIP with parametrized bus widths, standalone memory model and active slave support for back-to-back connectivity.

This VIP was made with emphasis on flexibility and reusability, it's not a DUT specific VIP but generic, created to support all AXI DUTs with small to no changes in code.

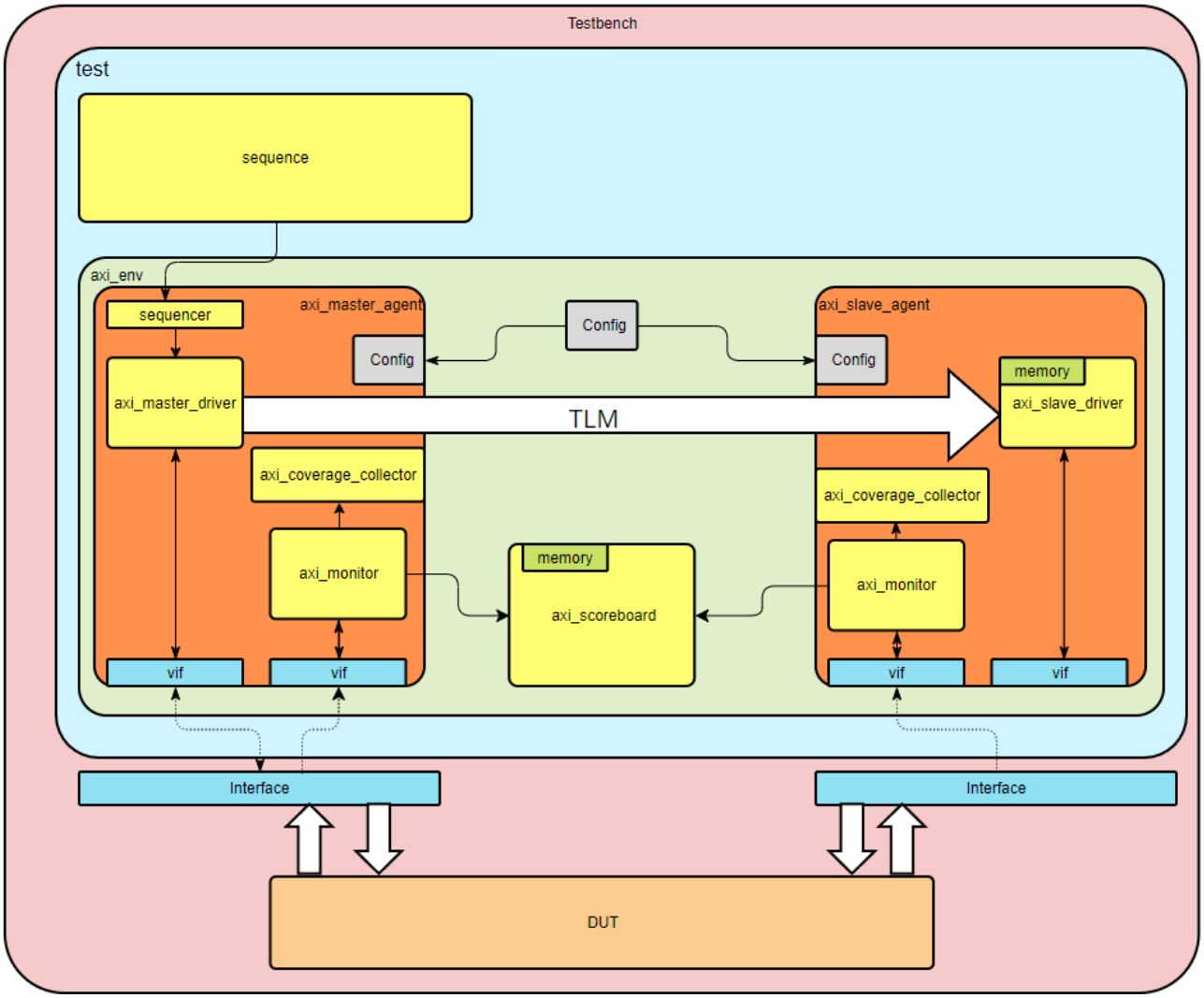

Fig. 1 – testbench architecture; master and slave drivers extend from mutual driver; axi_slave_driver and TLM port are present if slave is UVM_ACTIVE.

1.1 Supported features

- All types of bursts, lengths, sizes, narrow or unaligned transfers.
- Utility package (axi_utils_pkg.sv), containing all relevant environment parameters to run a simulation (dut or back-to-back).
- Parametrized memory model that can be configured (via config_db from test) to be FIFO or RAM/SAM type.
    - Not AXI specific, can be reused as a memory block for further projects.
    - Base class for axi_memory; has axi write/read methods, supports write/read priority arbitration and sends response items via TLM port.
- Reorder mechanism - verified in scoreboard both at master side (as if master/interconnect handles reorder), and slave side (as if transactions arrive in order to interconnect/master).
    - Slave side reorder checkers must be disabled if reorder is handled only in master side.
    - With B2B simulation, slave driver sends write response and read response transactions in randomized order, simulating different memory regions response delay.
- Generated transactions control from test or file.

1.2 Non supported features

- Lock, cache, prot, qos or region signals are currently purely randomized and are not verified by the reference model.
- EXOKAY and DECERR responses are not supported, SLVERR response triggers only for unsupported transfer sizes (>data bus) or FIFO over/underrun.
    - Env. Assumption – write response (b channel) returns last write response as the overall burst response.
      (For example; if a FIFO was overrun during a burst but had successful writes at the beginning, bresp will show SLVERR)
    - Env. Assumption – returned data from read operations with SLVERR is ZERO.
- 4KB address boundary is constrained in the sequence item, however due to the fact that some places tend to not care about this restriction, and have their own DUT implementation for such cases, the scoreboard will issue a warning when 4KB boundary violation occur, and may behave unexpectedly.

1.3 Run instructions

- Create a ‘src’ directory on same hierarchy with ‘sv’, place your verilog files there and change paths in compile.do

Run directory consists of:

- compile.do – compilation script
- clear.do – clears all logs before compilation
- run_regression.do – runs regression according to defined test list and creates merged coverage report.
- regression.sh – runs regression shell scripts (simply runs run_regression.do).
- run.do – runs simulation with coverage with the following format:
  <test_name> <number_of_transactions> <report_verbosity> <seed>
  Default test_name = axi_fixed_test
  Default number_of_transactions = 10
  Default verbosity = UVM_LOW

Default seed = random

2. Driving and sampling

The VIP uses clocking blocks to frame data drive and sample time, and present an option for different master, slave clocks designs.
The system clock and drive time are defined at clk_defines.sv under utils directory.
Drive time is set to be 20% of cycle, and sample time just before the next positive clk edge.

2.1 Coverage sampling

Covergroup sampling occurs on the receiving side.

- Address values and control signals (burst type/len/size/read or write) are sampled on **slave side**, covering generated stimulus.
- Responses are sampled on **master side**.
- Onehot patters for id, addr, data, and strobe sampled on **both sides** and cover integration.

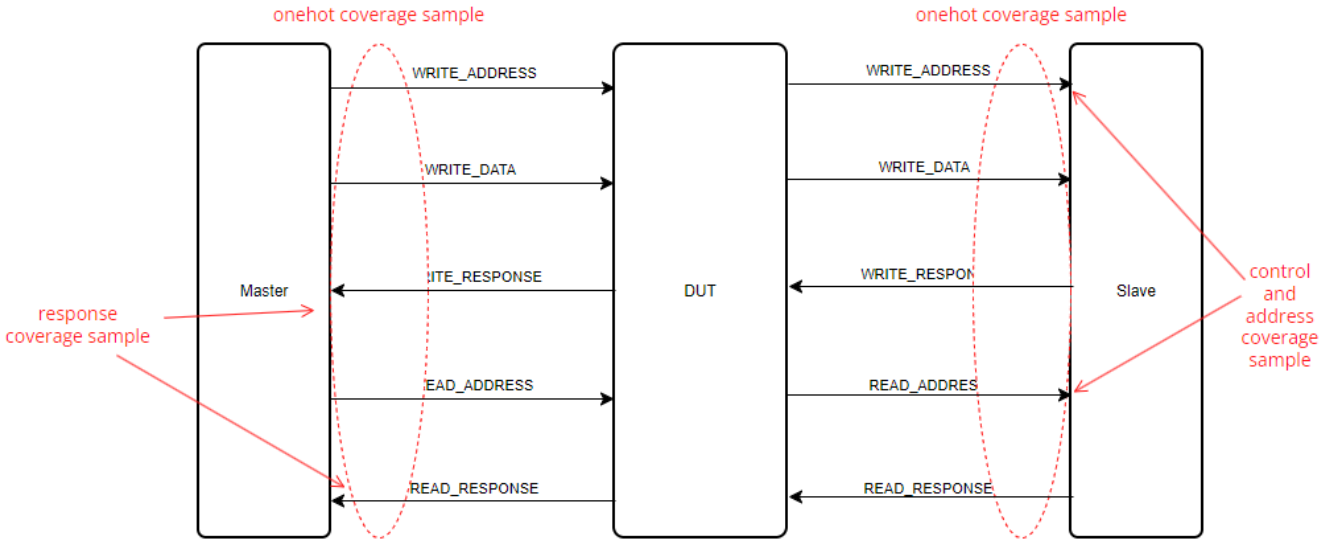

Fig. 2 – covergroup sampling

3. Test and coverage

3.1 Reference model

The reference model is integrated inside the scoreboard, and uses memory to simulate memory model.

- write address and write data channels transactions are being stored in expected queues at **master side**.
  - Issues an error when is not sequential.
- write response channel transactions at **slave side**, trigger write_axi_memory task, which in turn writes down the expected write address and write data transactions to the memory model.
- read address channel transactions are being stored at **master side**, and used both when preparing expected read response transactions, and reordering arriving transactions.
- read response channel at **slave side** triggers prepare_expected task, which reads from memory with every arriving burst beat (to simulate real time response, where a write can occur simultaneously) and checks order.
  - Issues an error for unordered transactions.
- read response channel at **master side** triggers collect_rdata task, which prepares actual data transactions per read address request, according to arriving responses after reordering.
  - Issues an error for unordered transactions.
  - Transactions that are done are being pushed to a queue, which initiates compare task.

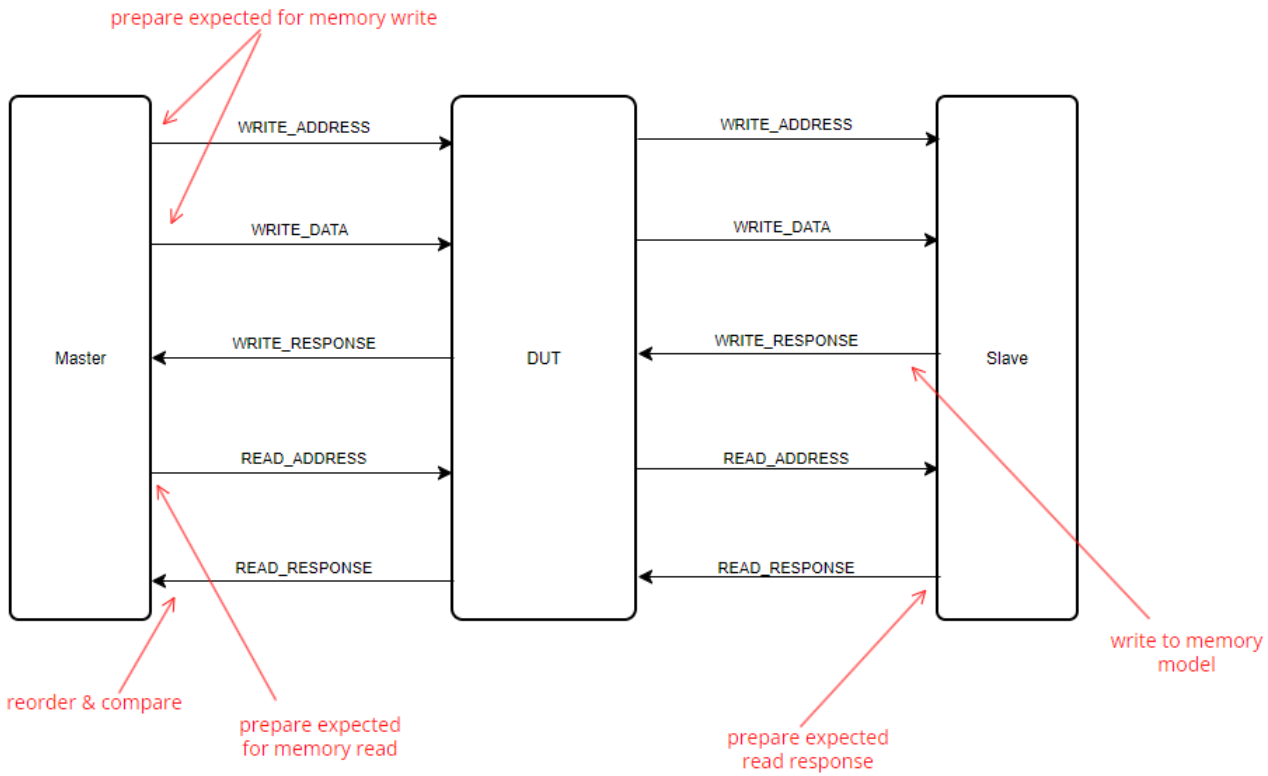

Fig. 3 – scoreboard sample and checkers

3.2 Test plan

Generated transaction data can be controlled from test with enabling select signal; the following fields are modifiable:

- Id
- Data
- Address
- Length
- Size

In addition, the user can force narrow transfers and aligned addresses only with pure randomization.

There's also an option to import a .txt file formatted with a certain pattern (see README.txt under sequences directory) to generate transactions.

Name/Section	Type	Description	Pass/fail criteria
axi_connectivity_test	Test	Generates onehot pattern data signals to id, addr, data, len and size to verify correct integration	All transactions reach their destination and match expected
axi_corner_cases_test	Test	Uses corner_cases.txt to generate: 1. minimum size with ZERO id, data and address 2. maximum size with ONES id, data and address 3. WRAP burst with wrapping condition	All transactions reach their destination and match expected
axi_gen_from_file_test	Test	Generate from file test, uses gen_file.txt as example with different transactions with various parameters	All transactions reach their destination and match expected
axi_fixed_test	Test	Purely randomized FIXED burst test	All transactions reach their destination and match expected
axi_incr_test	Test	Purely randomized INCR burst test	All transactions reach their destination and match expected
axi_wrap_test	Test	Purely randomized WRAP burst test	All transactions reach their destination and match expected
Assertions			
check_reset_low	Assertion	Verifies all valid signals are ZERO when active reset	Assertion pass at all times
check_bvalid_assert	Assertion	Verifies bvalid was raised only after wvalid, wready and wlast were asserted	Assertion pass at all times
check_rvalid_assert	Assertion	Verifies rvalid was raised only after arvalid and arready were asserted	Assertion pass at all times

check_wa_stable	Assertion	Verifies all write address channel signals are stable when awvalid is asserted, for at least one clock cycle after awready is asserted	Assertion pass at all times
check_wd_stable	Assertion	Verifies all write data channel signals are stable when wvalid is asserted, for at least one clock cycle after wready is asserted (if wready is not already high when wvalid is asserted)	Assertion pass at all times
check_wr_stable	Assertion	Verifies all write response channel signals are stable when bvalid is asserted, for at least one clock cycle after bready is asserted	Assertion pass at all times
check_ra_stable	Assertion	Verifies all read address channel signals are stable when arvalid is asserted, for at least one clock cycle after arready is asserted	Assertion pass at all times
check_rd_stable	Assertion	Verifies all read data channel signals are stable when rvalid is asserted, for at least one clock cycle after rready is asserted (if rready is not already high when rvalid is asserted)	Assertion pass at all times
check_wvalid_stable	Assertion	Verifies wvalid remains stable throughout burst	Assertion pass at all times
check_legal_awlen	Assertion	Verifies valid lengths for different write burst types	Assertion pass at all times
check_legal_arlen	Assertion	Verifies valid lengths for different read burst types	Assertion pass at all times

3.1 Coverage plan

Section	Description	Type
Integration		
Id_onehot	Id was assigned all onehot values	Covergroup
Addr_onehot	Addr was assigned all onehot values	Covergroup
Data_onehot	Data was assigned all onehot values	Covergroup
Strobe_onehot	Strobe was assigned all onehot values	covergroup
Control		
Axi_control	Write and read operations were covered, all burst types, all length and all sizes with cross coverage	Covergroup
Rresp_vals	All read response values were covered	Covergroup
Bresp_vals	All write response values were covered	Covergroup
Addr_vals	Address boundries (min, max) were covered, address range (splitted by user defined parameter) was covered	Covergroup
Reset_toggle	Reset was toggled at least once during the simulation	Assertion coverage
Reset_high	One of the valid signals was raised at earliest point after reset de assertion	Assertion coverage
Restrictions		
Max_wr_range	4Kb boundary was hit at least once at write transaction	Assertion coverage
Max_rd_range	4Kb boundary was hit at least once at read transaction	Assertion coverage
Functionality		
Wlast	Wlast signal raised one cycle before wvalid is de asserted	Assertion coverage
Rlast	Rlast signal raised one cycle before rvalid is de asserted	Assertion coverage
Awvalid	Awvalid is de asserted at least one clock cycle after awready was raised	Assertion coverage
wvalid	wvalid is de asserted at least one clock cycle after wready was raised	Assertion coverage
bvalid	bvalid is de asserted at least one clock cycle after bready was raised	Assertion coverage
Arvalid	Arvalid is de asserted at least one clock cycle after arready was raised	Assertion coverage

rvalid	rvalid is de asserted at least one clock cycle after rready was raised	Assertion coverage
Timing		
Write_delay	Write response arrived at user defined clk delay range after request	Assertion coverage
Read_delay	Read response arrived at user defined clk delay range after request	Assertion coverage