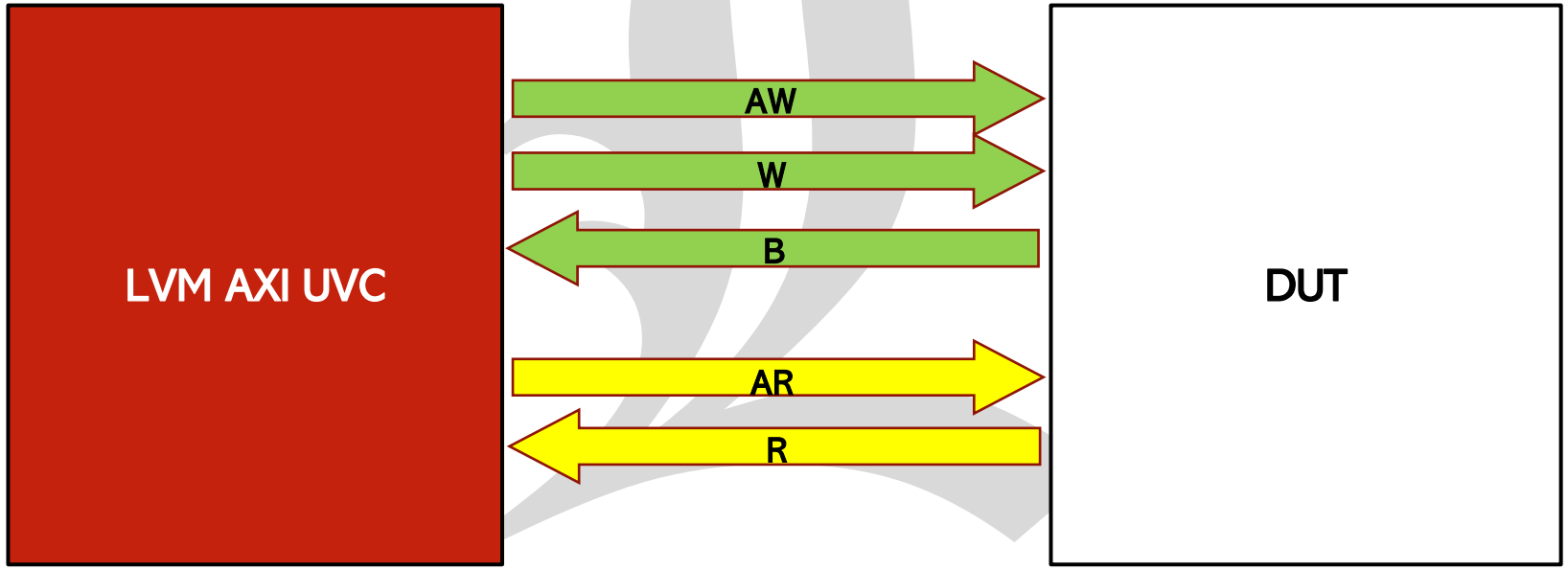
AXI3 / AXI4 / AXI4-Lite VIP

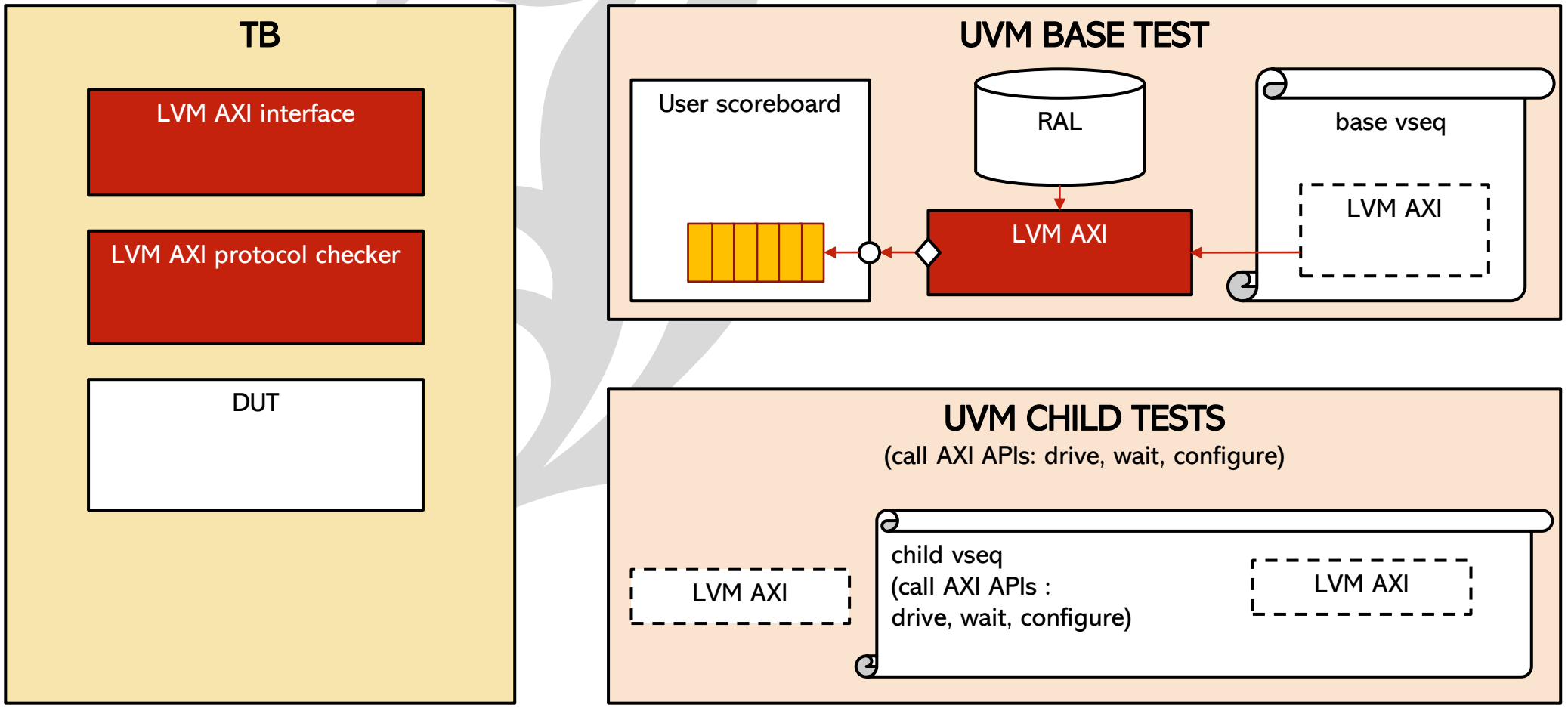
INTRODUCTION

- This is an AXI4 UVC with full UVM compatibility.
- It has been coded to be robust, reusable, measurable, systematic and efficient, with easy debug-ability and user-friendly features.
- Objectives:
  - To help create various AXI4 stimulus with ease (minimal codes).
  - To shorten the time to bring up AXI4 UVM testbench with RAL.
  - To provide users with high quality industry standard protocol checkers.
  - To create an ideal platform for novice UVM users.
  - To provide a **low cost solution** for AXI4 verification in the industry.

THE LVM AXI



EXAMPLE LVM AXI INSTALLATION



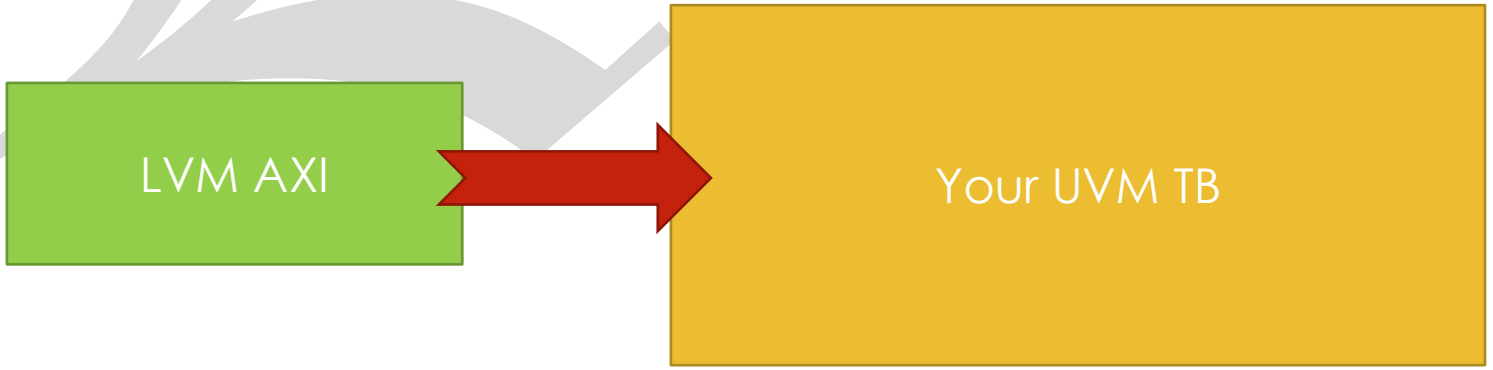
STRENGTHS

USER FRIENDLY

Integration and Configuration

INTEGRATION & CONFIGURATION

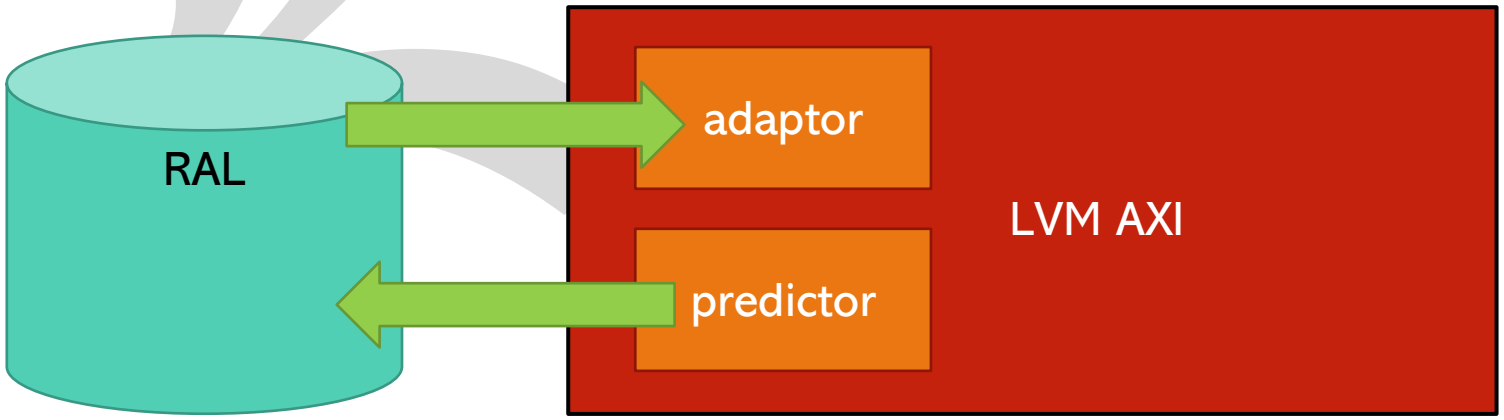
- Very easy to integrate, simpler than other vendor
  - All steps are demonstrate in the self verification testbench.
- Can configure different protocol per instance
- Can configure different signal width per instance
- Easily can on-off components on the fly.
- Just need to instantiate the UVC, no need to do anything on cfg class, sequence, sequencer, adaptor, predictor etc



DRIVING PART

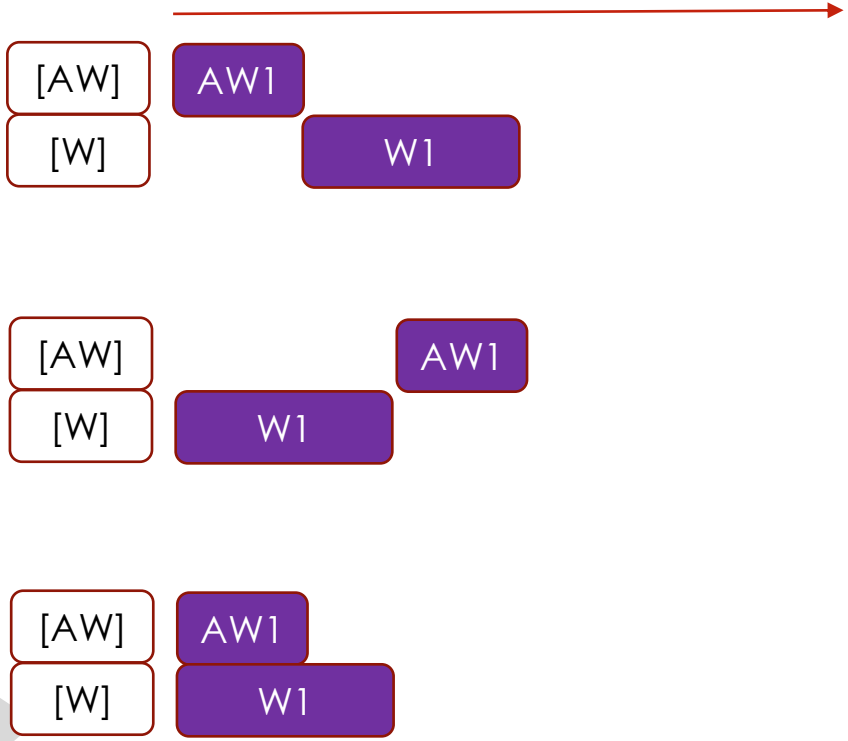
ADVANCED TECH WITH RAL

- Predictor and Adaptor are built in, connections syntax is ready.
- Support partial and full register access (individual accessible modes) for driving
  - 8b accesses
  - 16b accesses
  - 32b accesses
- Predictor works for burst packets, and partial accesses packets



EASE OF DRIVING STIMULUS

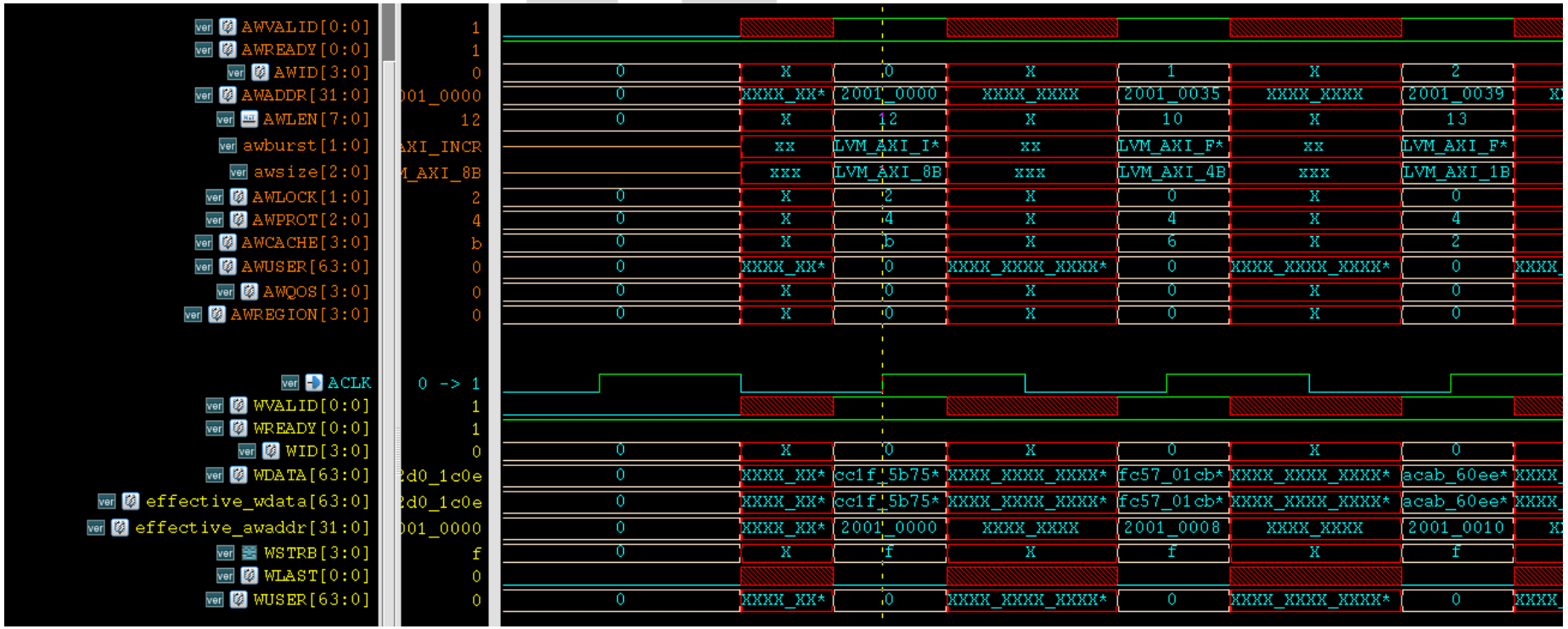
- All fully pipelined.
- API based.
- Various API to do more stimulus style
  - Ease of driving AW and W sequences
    - AW first
    - W first
    - Parallel
  - Parallel AW and AR, busiest traffic
  - Wait and driving capabilities.
  - Ready signals, power signals driving.
- Most API can be done in 1 line of code.
- Full examples at self verification uvm tests.




```
m_axi_env.s_write(.AWADDR(32'h2000_0000),.wdata(32'h1111_2222),.BID(BID),.BRESP(BRESP));
```

SETUP AND HOLD X INJECTION

- Already supported X injection for window outside setup and hold time.
- Can be easily configured and can be turned OFF too.



REUSABLE CODE

- As stimulus mostly done using UVC's API, the code is very reuse friendly, where just the UVC handle is needed.

```
m_axi_env.drv_wait_output(LVM_ON);


m_axi_env.s_write(.AWADDR(32'h2000_0000),.wdata(32'h1111_2222),.BID(BID),.BRESP(BRESP));
`uvm_info(msg_tag, $sformatf("BID=%0h BRESP=%0h", BID, BRESP), UVM_DEBUG)


m_axi_env.s_read (.ARADDR(32'h2000_0000),.rdata(rdata)        ,.RID(RID),.rresp(rresp));
`uvm_info(msg_tag, $sformatf("RID=4'h%0h, RRESP=2'h%0h, RDATA=32'h%0h", RID, rresp, rdata), UVM_DEBUG)
```

SIMPLE SEQ ITEM RETRIEVAL

- Full code for seq item retrieval for all info needed is already part of example user scoreboard
- Already can be used for various high level scoreboard.

```
`uvm_info(msg_tag, $sformatf("AWID='h%h  AWADDR='h%h  AWPROT='h%h  AWLEN='h%h  AWSIZE='h%h  AWBURST='h%h  AWLOCK='h%h  AWCACHE='h%h",
    captured_item.AWID,
    captured_item.AWADDR,
    captured_item.AWPROT,
    captured_item.AWLEN,
    captured_item.AWSIZE,
    captured_item.AWBURST,
    captured_item.AWLOCK,
    captured_item.AWCACHE
), UVM_MEDIUM)
```

```
        foreach(captured_item.WDATA[i]) begin
            // effective read data per address
            write_impact = "";
            foreach (captured_item.mem_impact[i].addr[j])
            write_impact={$sformatf(" %h<-%2h", captured_item.mem_impact[i].addr[j],      captured_item.mem_impact[i].data[j]),write_impact};


            `uvm_info(msg_tag, $sformatf("[Beat %3d %2d:%2d] ADDR='h%h  EFF='h%h  WDATA='h%h  WUSER='h%h  WSTRB='h%h",
                i,
                captured_item.msb[i],     captured_item.lsb[i],
                captured_item.ADDR[i],
                captured_item.effective_wdata[i],
                captured_item.WDATA[i],
                captured_item.WUSER[i],
                captured_item.WSTRB[i]
            ), UVM_MEDIUM)


            // effective write data per address
            write_impact = "";
            foreach (captured_item.mem_impact[i].addr[j])
            write_impact={$sformatf(" %h<-%2h", captured_item.mem_impact[i].addr[j],      captured_item.mem_impact[i].data[j]),write_impact};
            `uvm_info(msg_tag, $sformatf("                IMPACT=%s",write_impact), UVM_MEDIUM)
        end
```

MONITORING PART

COMPREHENSIVE TRACKER

- Full info for each packet in the AXI bus can be observed via tracker.
- Make the debug handy

```
[Packet 547: WRITE 537]     23190.000 ns
       ------------------------------------------------------------------------------------------------------------------------------------------
       AWID=4'h8  AWADDR=32'h20010012  AWLEN=8'h04  AWSIZE=LVM_AXI_1B  AWBURST=LVM_AXI_INCR   AWLOCK=LVM_AXI_LOCKED     AWCACHE=4'hb  AWPROT=3'h0

                          WID   ADDR          EFFECTIVE            WDATA                  WSTRB  IMPACT
       [Beat   0   23:16]  4'h8  32'h20010012  8'h11                64'h18c50a6c14118e19  8'h04  20010012<-11
       [Beat   1   31:24]  4'h8  32'h20010013  8'h74                64'ha610cf3c74ab84b3  8'h08  20010013<-74
       [Beat   2   39:32]  4'h8  32'h20010014  8'hff                64'he9d4ceffa68b78e1  8'h10  20010014<-ff
       [Beat   3   47:40]  4'h8  32'h20010015  8'h71                64'hdf6971101f6bff2a  8'h20  20010015<-71
       [Beat   4   55:48]  4'h8  32'h20010016  8'hb6                64'h69b65808ae1b36a5  8'h40  20010016<-b6

       BID=4'h8  BRESP=LVM_AXI_OKAY

       +-------------------------------------------------------------------------------------------------------+
       |  AWCACHE[3]=1          AWCACHE[2]=0                AWCACHE[1]=1               AWCACHE[0]=1              |
       |  LVM_AXI_ALLOCATE      LVM_AXI_NO_OTHER_ALLOCATE   LVM_AXI_MODIFIABLE         LVM_AXI_BUFFERABLE        |
       |                                                                                                       |
       |                        AWPROT [2]=0                AWPROT [1]=0               AWPROT [0]=0              |
       |                        LVM_AXI_DATA_ACCESS         LVM_AXI_SECURE_ACCESS      LVM_AXI_UNPRIVILEGED_ACCESS  |
       +-------------------------------------------------------------------------------------------------------+
```

END OF TEST MEMORY PRINTER
END OF TEST SCOREBOARD PRINTER

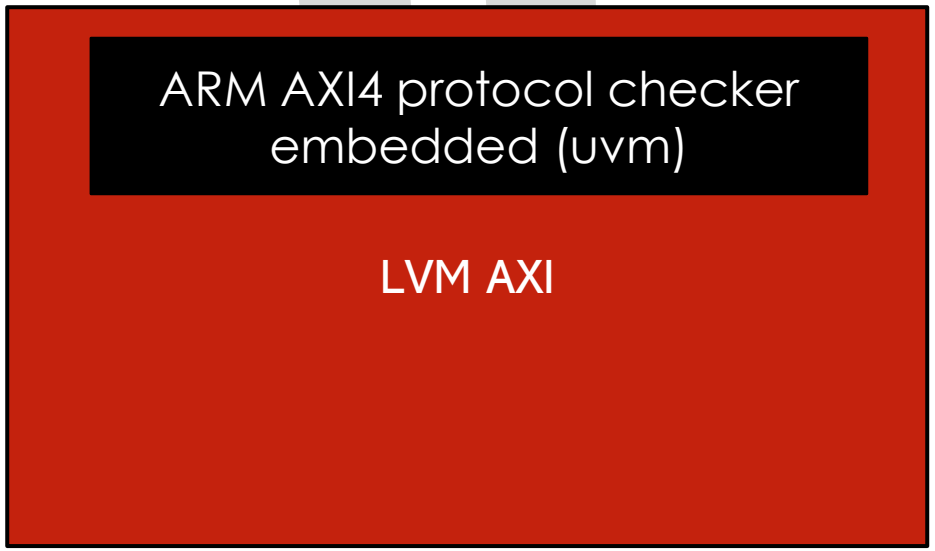
- It prints all memory contents exercised in the test to ease the debug of the AXI traffic impacts in the test.
- Also print total read and total write, ensure test is not empty.

```
--------------------------------
     AXI Scoreboard Report
--------------------------------
    Total AXI Reads :      10
    Total AXI Writes:     538
    Total Checked   :      10
--------------------------------
```

CHECKING PART

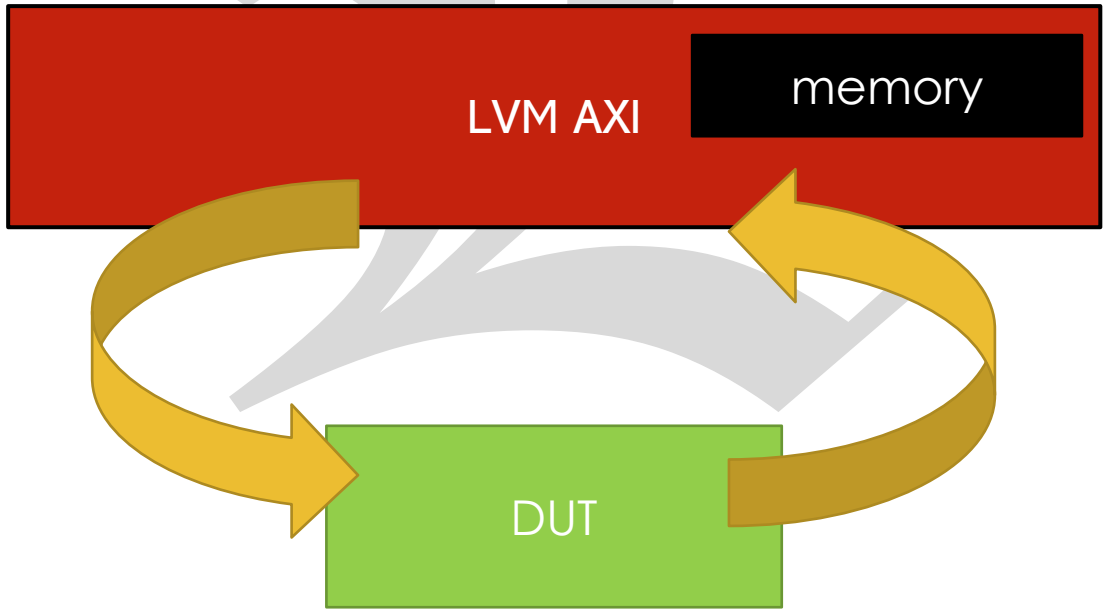
EMBEDDED ARM PROTOCOL CHECKER

- Already integrated SVA from ARM for AXI4 protocol checker.
- Enhanced to become UVM_ERROR when assertions fail.



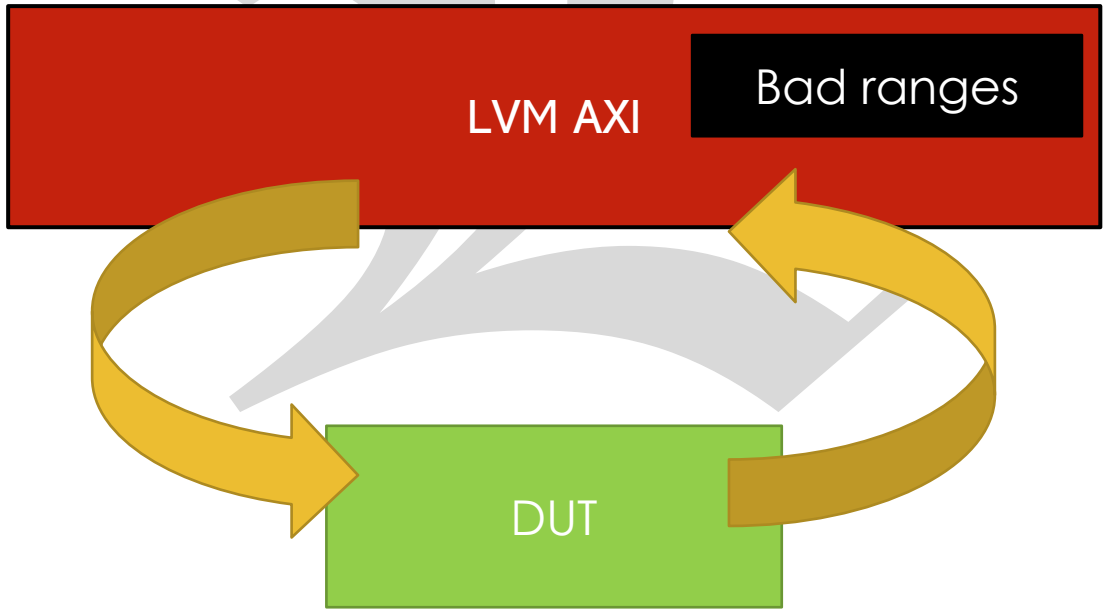
EMBEDDED MEMORY CHECKING

- Built in memory verification scoreboard, where
  - all writes within memory range (configurable) will be keep tracked as new data (fulfil the conditions)
  - All reads within memory range (configurable) will check data matching expectation (per last written data)



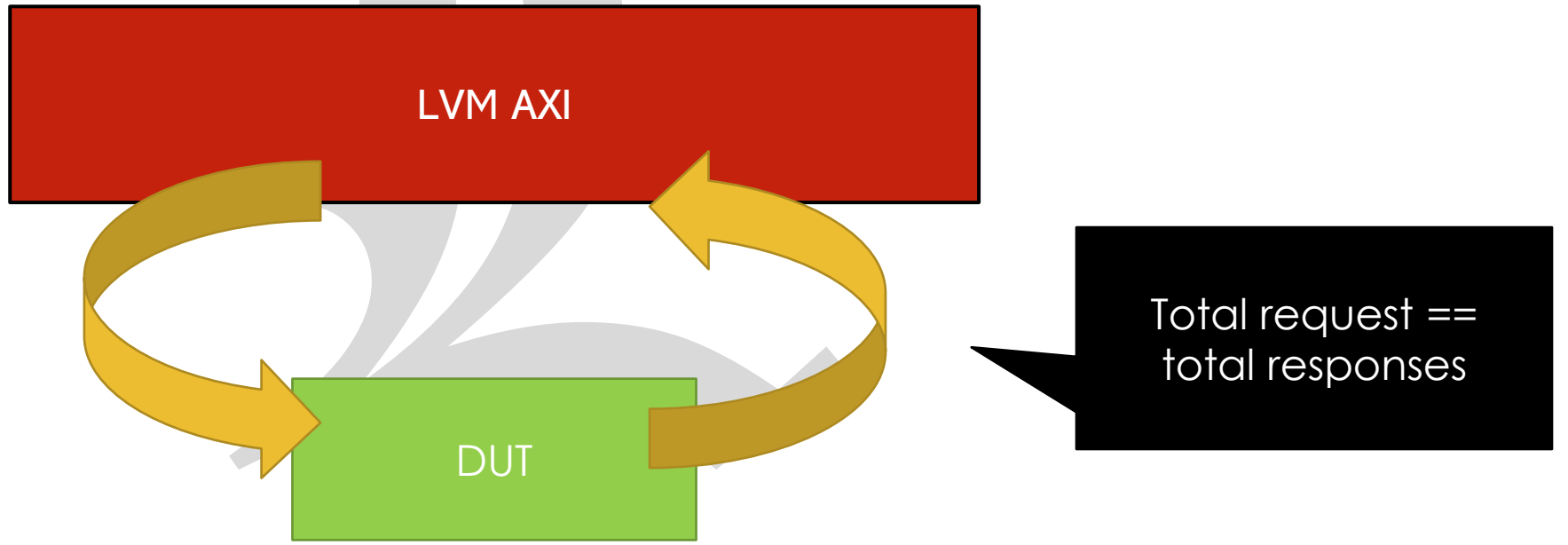
OUT OF BOUND ADDRESS CHECKING

- Built in out of bound address verification scoreboard, where
  - all writes outside valid address range will get BRESP=DECERR/SLVERR (user configurable)
  - All reads outside valid address range will get RRESP=DECERR/SLVERR (user configurable)



AUTO ENSURE SLAVE COMPLETES ALL REQUESTS

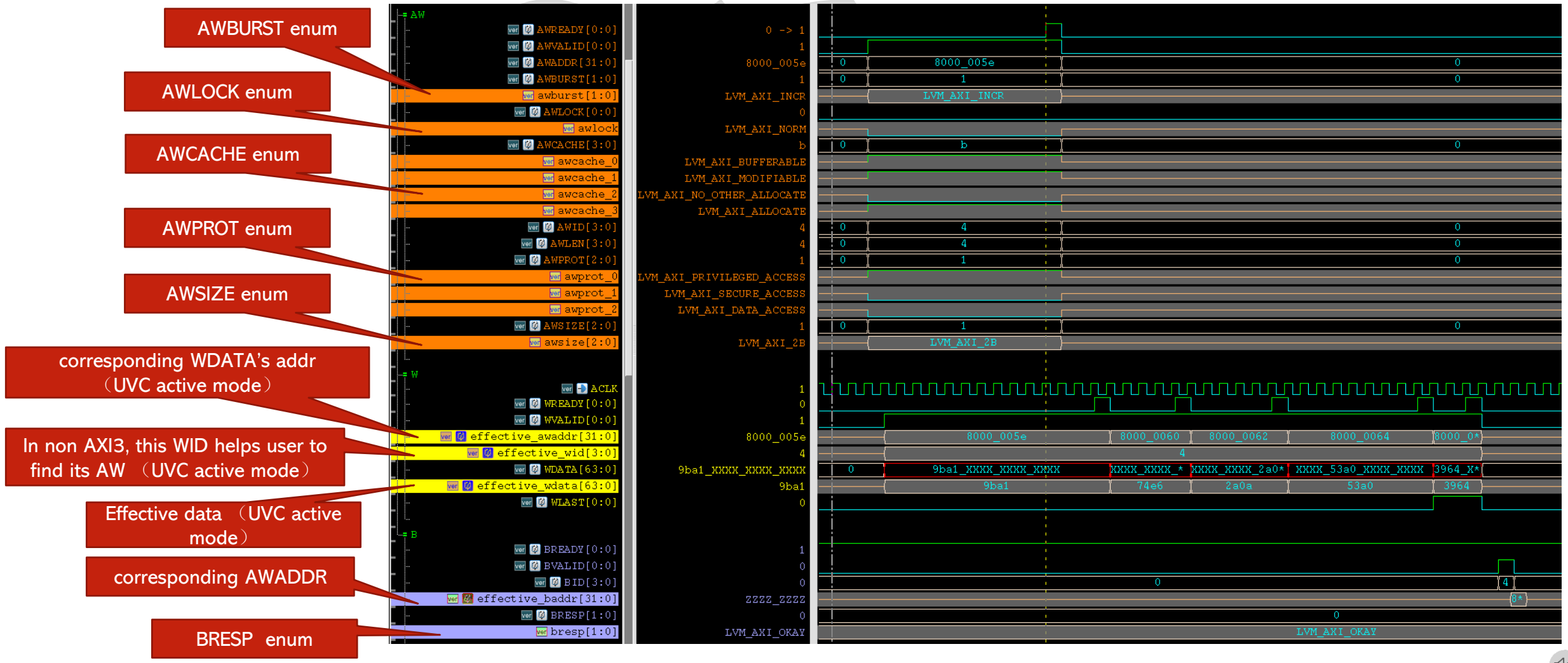
- At the end of test, UVC will ensure slave does not missed out any read / write.



DEBUG PART

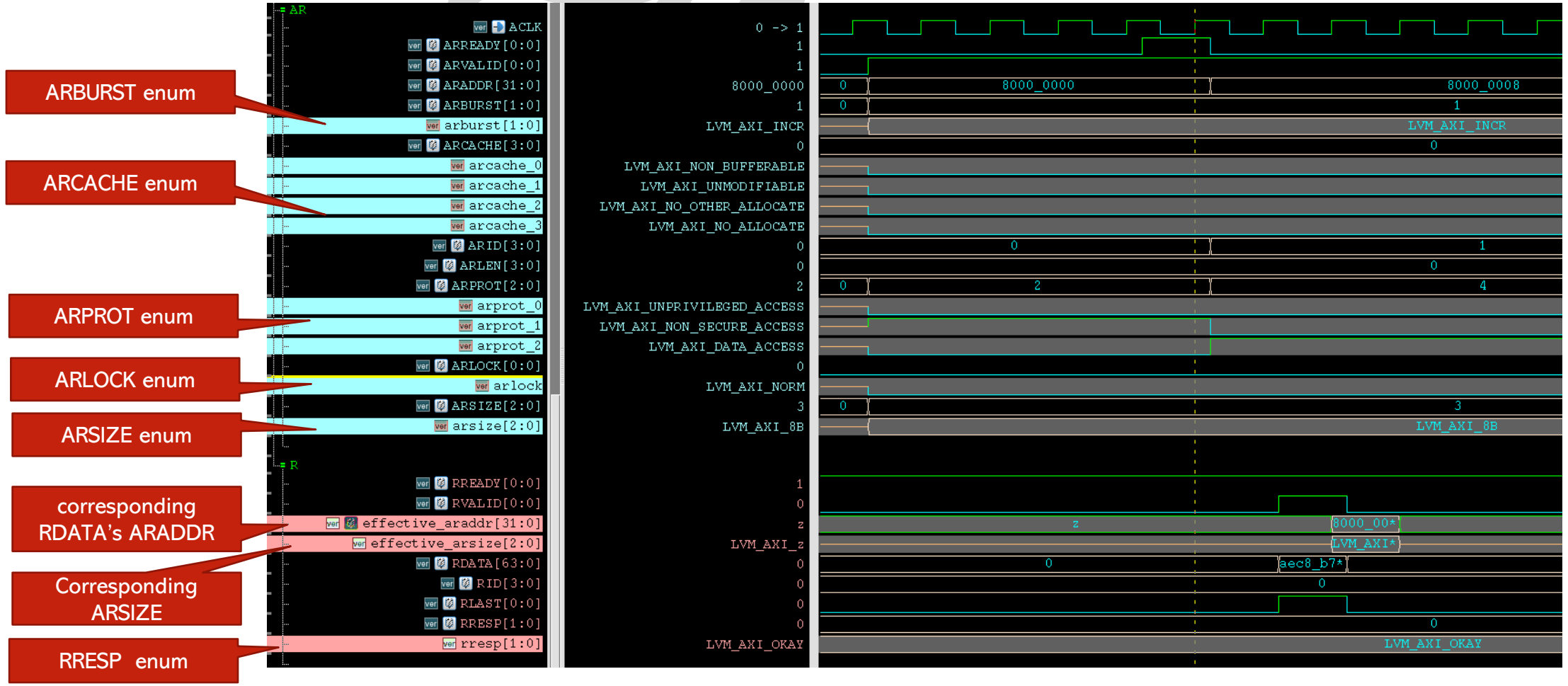
ENUM & DEBUG SIGNALS

- awburst, awsize, bresp, and more



ENUM & DEBUG SIGNALS

- arburst, arsize, rresp and more



TESTSUITE

THE BENEFITS FROM TESTSUITE

- Can verify various aspects, for example:
  - Verify the every channel outstanding depth.
  - Verify the response of every packet to meet expected value.
  - Verify the data integrity, with full blast of randomized AXI packets
  - Verify all types of BREADY, RREADY conditions (for stressing slaves/interconnect)

PERFORMANCE ANALYZER

PERFORMANCE ANALYZER

- Performance per AXI channels

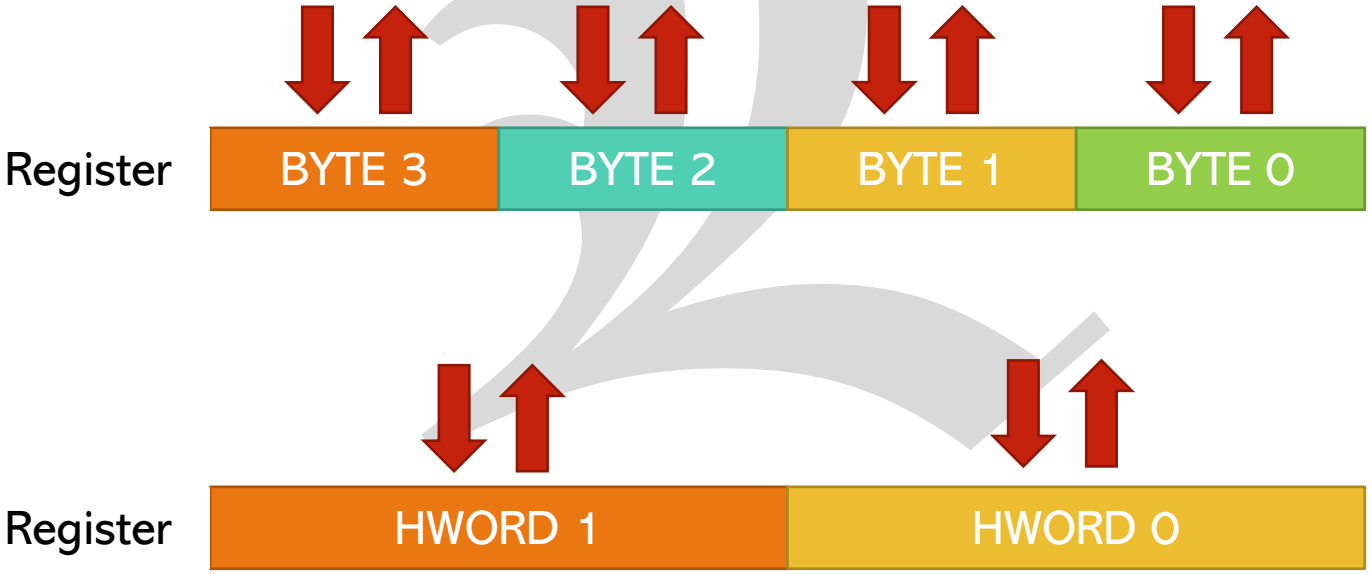
AXI Performance Report

Chnl	Total thru	Total waits	Average wait wait / beat	% Bus utilization	Performance
AW	472	3390	7.182	12	WORST
W	472	3352	7.102	12	WORST
AR	924	7939	8.592	10	WORST
R	924	0	0.000	100	EXCELLENT
B	472	0	0.000	100	EXCELLENT

PARTIAL REGISTER ACCESS VERIFICATION

PARTIAL REGISTER ACCESS VERIFICATION

- Can verify register partial access
  - Full access is fully verified at uvm's bit bashing sequence
- LVM adds the coverage for partial accesses



BURST REGISTER ACCESS VERIFICATION

BURST REGISTER ACCESS VERIFICATION

- Can verify register burst access
  - Single beat full access is fully verified at uvm's bit bashing sequence
- LVM adds the coverage for burst accesses
- This enable the bus read / write more than 1 registers in burst modes, where randomized burst size will be covered as well.
- Example like below, where there are 4 burst packets (in 4 different colors) in 1B to program all the 6 x 32bits registers.

Register 0	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 1	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 2	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 3	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 4	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 5	BYTE 3	BYTE 2	BYTE 1	BYTE 0

WHY CHOOSING LVM VIP?

Strengths

User friendly

Minimum lines of code to send packet.
Friendly for new UVM engineers.
API based UVC.

Reusable

Codes on lvm VIP is highly reusable.

Ease of Integration

RAL-ready with adaptor and predictor.
Minimum steps to integrate.

Light weight

CPU efficient UVC.

Robust

Highly configurable.
Parameterized signal width per instance.

High debug-ability

Useful tracker log, interface signals.

Performance Analyzer

Performance analyser for each channel

Strong & Strict Checker

Industry standard checker embedded.
Support X injection at Read and Write DATA for inactive lanes.
Embedded memory checker, RAL access OK checker

Reset aware

Support reset events.

Ready testsuite

Provided multiple useful tests to verify AXI slaves DUT

Setup and Hold X injection

Inject X outside setup and hold window.

Register Partial Access

Embedded register partial access where user just need 2 lines of codes to start it

Register Burst Access

Embedded register burst access where user just need ~4 lines of codes to start it

STRENGTH SUMMARY

- Robust
  - Easy-to-control packet sending style: pipelined or gated styles.
  - Flexible sending order: for example - W followed by AW.
  - Fully parameterized UVC
    - All the signal's width can be easily parameterized.
    - Supports multiple instances with different parameters.
- User friendly
  - Easy to use as driving, waiting and configuring are API-based.
  - User need not deal with sequence for most of the time (user friendly for engineers new to UVM).
  - Does not require in-depth UVM knowledge to use this UVC.
  - Ease of configuration
    - Component(s) can be turned off.
    - Component(s) can be silenced.
    - UVC can be configured via API.
  - Examples of tests, scoreboard and sequence given as a handy reference for the users.
  - Easy to configure the UVC in passive mode.

STRENGTH SUMMARY

- Reusable
  - Proven to be easily instantiable at module level testbench or SOC level testbench.
  - Easy to reuse code that deals with the UVC.
- Ease of integration
  - Minimum steps needed from integration to sending the first AXI packet.
  - It is RAL ready.
- High debug-ability
  - Tracker file: AXI transactions can be traced in the log file.
  - Interface provides some debug signals.
- Strong checkers
  - Equipped with industry standard protocol checks on the AXI bus.
  - Equipped with memory verification scoreboard (background check read and write within address ranges)
- Reset aware feature
  - The reset-aware UVC flushes the pending transactions if ARESETn goes active.
- Light weight
  - Efficient use of variables for least memory space consumption over the simulation time.
- Setup and hold ready
  - It can be configured to inject X outside setup and hold window.

CONVENTION

CONVENTION

- AW: Write Address Channel
- W : Write Data Channel
- B : Write Response Channel
- AR : Read Address Channel
- R : Read Data+Response Channel

ENUM TYPE READY TO BE USED

```
// X injection or clocking block
   typedef enum bit {
       LVM_CLOCKING        = 1'b0,
       // Using old way of injecting the delay via clocking block
       LVM_X_INJECTION     = 1'b1
       // Outside setup and hold time, X is injected
   } LVM_SETUP_HOLD_em;

typedef enum bit {
       LVM_MASTER          = 1'b1,
       LVM_SLAVE           = 1'b0
   } LVM_ROLE
```

```
// ON OFF
   typedef enum bit {
       LVM_ON              = 1'b1, // ON
       LVM_OFF             = 1'b0  // OFF
   } LVM_ON_OFF_em;

   // TRUE FALSE
   typedef enum bit {
       LVM_TRUE            = 1'b1,
       LVM_FALSE           = 1'b0
   } LVM_TRUE_FALSE_em;

   // RESET
   typedef enum bit {
       LVM_RESET           = 1'b0,
       LVM_OUT_OF_RESET    = 1'b1
   } LVM_RESET_em;
```

ENUM TYPE READY TO BE USED

```
typedef enum bit [2:0] {
        LVM_AXI_1B                      = 3'h0,
        LVM_AXI_2B                      = 3'h1,
        LVM_AXI_4B                      = 3'h2,
        LVM_AXI_8B                      = 3'h3,
        LVM_AXI_16B                     = 3'h4,
        LVM_AXI_32B                     = 3'h5,
        LVM_AXI_64B                     = 3'h6,
        LVM_AXI_128B                    = 3'h7
} lvm_axi_burst_size_em;
```

```
typedef enum bit [1:0] {
    LVM_AXI_OKAY                        = 2'h0,
    LVM_AXI_EXOKAY                      = 2'h1,
    LVM_AXI_SLVERR                      = 2'h2,
    LVM_AXI_DECERR                      = 2'h3
} lvm_axi_response_em;
```

```
typedef enum bit [2:0] {
        LVM_AXI_READ                    = 0, // AR
        LVM_AXI_WRITE                   = 1, // complete write AW+W
        LVM_AXI_WRITE_ADDR              = 2, // just AW
        LVM_AXI_WRITE_DATA              = 3, // just W
} lvm_axi_op_em;
```

```
typedef enum bit [1:0] {
    LVM_AXI_FIXED                       = 2'h0,
    LVM_AXI_INCR                        = 2'h1,
    LVM_AXI_WRAP                        = 2'h2,
    LVM_AXI_RSVD_TYPE                   = 2'h3
} lvm_axi_burst_type_em;
```

AXI UVC COMPONENTS

UVC HIERARCHY PATHS

- If user instantiates the LVM AXI UVC under test, the following paths will be valid:
  - uvm_test_top.m_axi_env
  - uvm_test_top.m_axi_env.agt
  - uvm_test_top.m_axi_env.agt.sqr
  - uvm_test_top.m_axi_env.agt.drv
  - uvm_test_top.m_axi_env.agt.mon
  - uvm_test_top.m_axi_env.cfg
  - uvm_test_top.m_axi_env.prd
  - uvm_test_top.m_axi_env.adp

AXI UVC COMPONENTS

- Driver (drv)
  - Configurable sending style that can be changed on-the-fly:
    - Ungated / Pipelined packets: Everything send in back-to-back.
    - Gated packets: Wait for response, then proceed to send.
- Monitor (mon)
  - Captures all AXI transactions and provides seq item port for user.
  - Prints out tracker log file.
- Configuration (cfg)
  - Contains all the configurable parameters and some APIs.
- Environment (env)
  - Provider of all the user's API.

AXI UVC COMPONENTS

- RAL adaptor (adp)
  - Ready-to-use adaptor to convert user's RAL access commands into AXI transactions.
- RAL predictor (prd)
  - Ready-to-use predictor to convert monitored AXI transactions into RAL update mechanism.
- Protocol Checkers (sva)
  - Complete checker for protocol compliancy.
  - SVA that flags error using “uvm_error”.
  - Provides SVA coverage for user analysis.

UVC CONFIGURATION

CONFIGURABLE UVC

- LVM AXI UVC is designed to be fully configurable to meet the user's needs.
- All the configuration variables are located inside the config class.
- Most of the variables can be controlled by using API inside the env.
- Besides, this UVC is fully configurable using parameters for signal width.
  - For details, please refer to “Step by Step Integration Guide”.

LVM POWERED API

No	API name	Example
1	enter_reset	<env>.enter_reset; // This makes the UVC to enter reset state (auto done when ARESETn fall)
2	exit_reset	<env>.exit_reset; // This makes the UVC to exit reset state (auto done when ARESETn rise)
3	deactivate_UVC	<env>.deactivate_UVC; // Driver, monitor etc. stop their operation (called by enter_reset)
4	activate_UVC	<env>.activate_UVC; // Driver, monitor etc. back to being operational (called by exit_reset)
5	posedge_clk	<env>.posedge_clk; // Wait for next posedge of UVC clock
6	negedge_clk	<env>.negedge_clk; // Wait for next negedge of UVC clock
7	recompute_clk_period	<env>. recompute_clk_period; // Restart the UVC clock frequency calculation

```
// Clock info is shown in log file
[lvm_axi_cfg] clock period = 20.000 ns
[lvm_axi_cfg] window of 1  = 10.000 ns
[lvm_axi_cfg] window of 0  = 10.000 ns
[lvm_axi_cfg] duty cycle   = 50.0%
```

LVM POWERED API

No	API name	Example
1	off_driver	<env>.off_driver; // Turn OFF driver
2	on_driver	<env>.on_driver; // Turn ON driver
3	off_monitor	<env>.off_monitor; // Turn OFF monitor
4	on_monitor	<env>.on_monitor; // Turn ON monitor
5	off_tracker	<env>.off_tracker; // Turn OFF tracker
6	on_tracker	<env>.on_tracker; // Turn ON tracker

Useful when user makes this UVC as passive.

PROTOCOL

- User can select either AXI3 / AXI4 / AXI4-lite for this UVC, using the following cmd:

```
m_axi_env.cfg.protocol = LVM_AXI4_LITE;
```

- Default it is full AXI4 protocol.

SIGNAL IMPLEMENTATION

- In AXI4, xUSER, AxREGION and AxQOS are the additional signals.
- User might choose not to implement them in the design, thus, the UVC can aware of that, by these settings

No	variable name	Example

1	got_user[4:0]	<env>.cfg.got_user='0; // Turn OFF user signals for all channels 4:AW, 3:W, 2:B, 1:AR, 0:R
		// 1 bit for 1 channel, can turn off any bit individually, eg
		<env>.cfg.got_user[3]=1'b0; // this turn off only WUSER, while the rest still there

2	got_qos[1:0]	<env>.cfg.got_qos='0; // Turn OFF qos signals for AW (bit 1) and AR (bit 0)
		// can turn off any bit individually, eg
		<env>.cfg.got_qos[0]=1'b0; // this turn off ARQOS only, while AWQOS still there

3	got_region[1:0]	<env>.cfg.got_region='0; // Turn OFF region signals for AW (bit 1) and AR (bit 0)
		// can turn off any bit individually, eg
		<env>.cfg.got_region[0]=1'b0; // this turn off ARREGION only, while AWREGION still there

SUPPORT_EXCLUSIVE

- User can configure the UVC whether support Exclusive Access
- Default is 1 (support)

```
m_axi_env.cfg.support_exclusive = 1'b0; // not supporting exclusive access
```

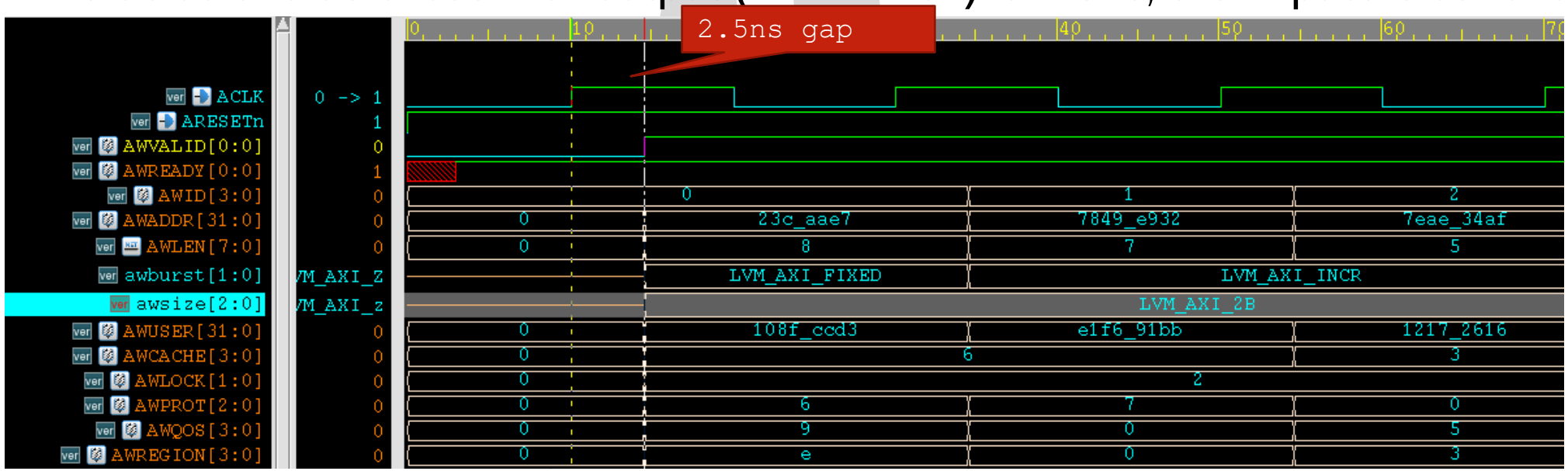
SUPPORT_PWR

- User can configure the UVC whether support power related signals
- Default is 1 (support)

```
m_axi_env.cfg.support_pwr = 1'b0; // not supporting AXI power pins
```

CLOCKING BLOCK CONFIGURATION

- This UVC implements the standard clocking blocks.
- The default value chosen for output (VIP → DUT) is 2.5ns, the impact is as follows:

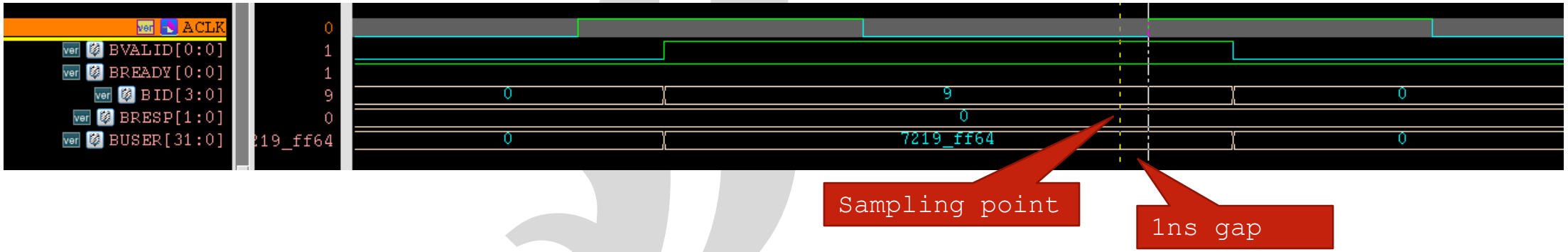


- This gap can be configured using the following cmd:

```
m_axi_env.set_hold_time (2500 ,"ps");
```

CLOCKING BLOCK CONFIGURATION

- The default value chosen for input (DUT → VIP) is 1ns, the impact is as follows:



- This gap can be configured using the following cmd:

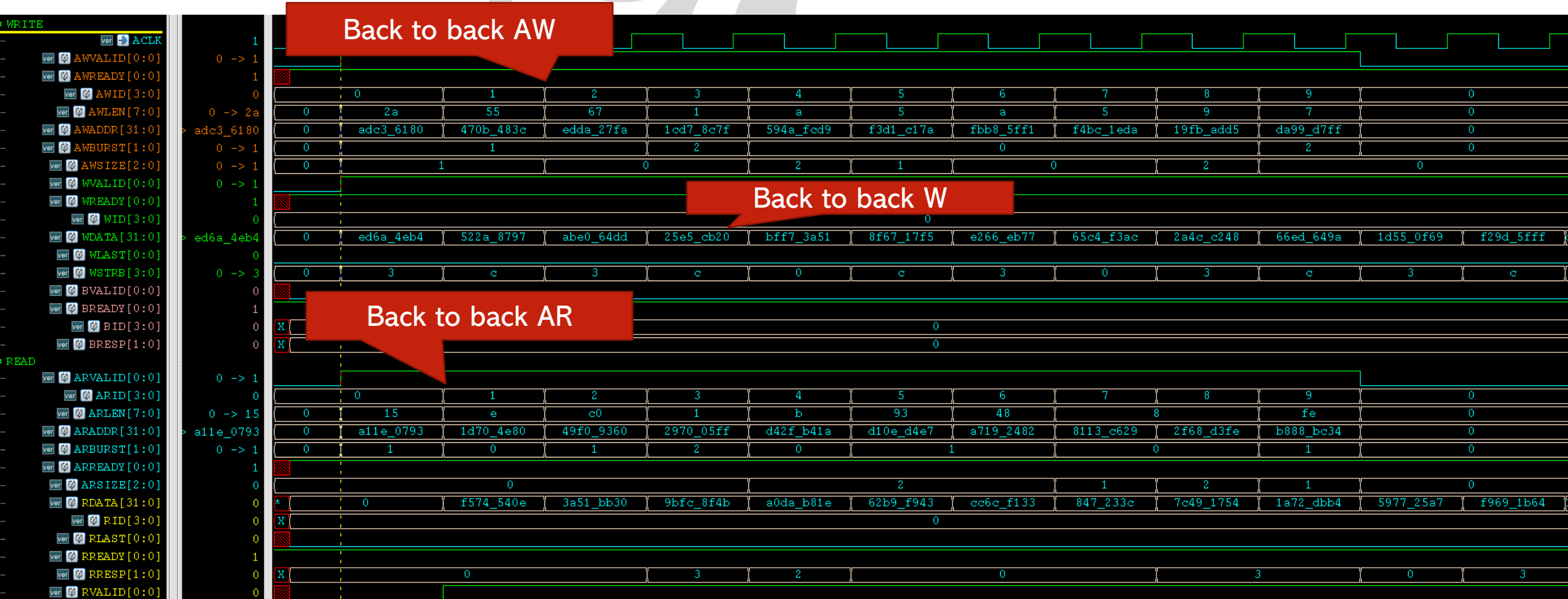
```
m_axi_env.set_setup_time(1     ,"ns");
```

DRIVER WAIT STYLE

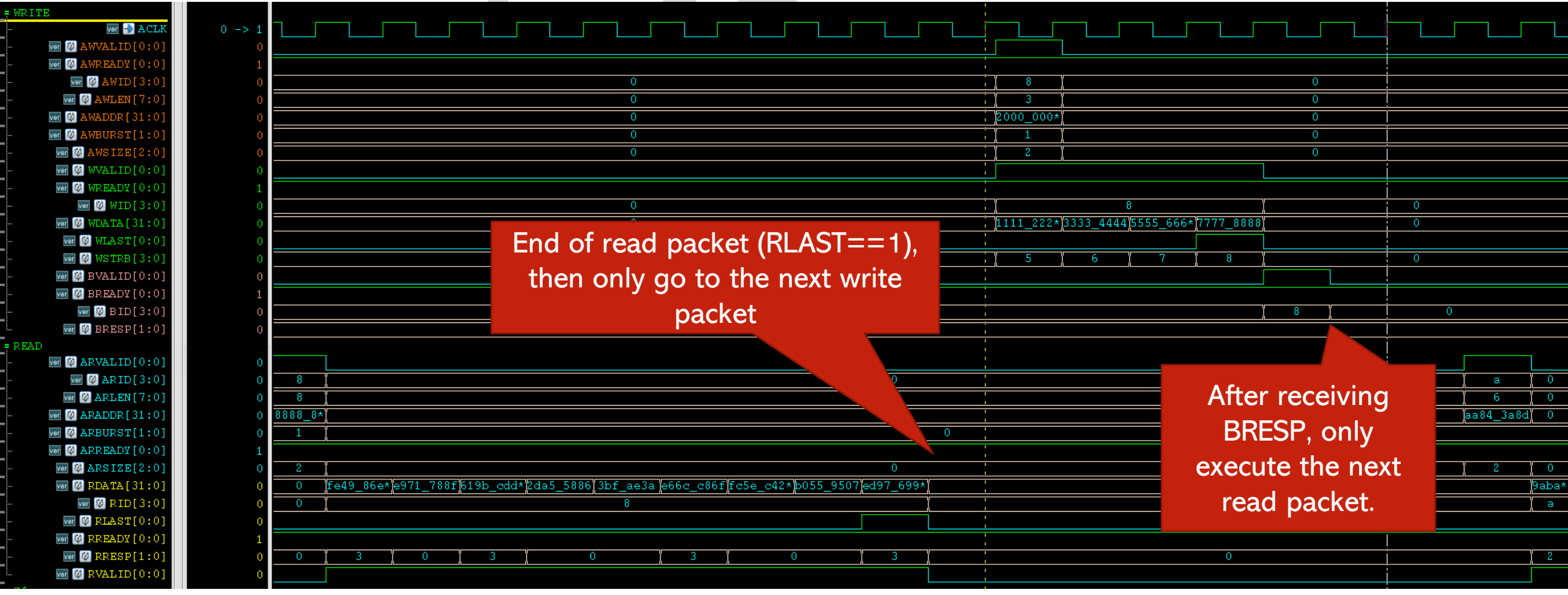
- The driver can be configured by user to wait for the slave response after it has sent a packet. For example,
  - After AR is sent, drv can be made to wait for all RDATAs to come back with RLAST.
  - After AW+W are sent, drv can be made to wait for B to come back.

No	API name	Example
1	drv_wait_output(LVM_ON)	<env>.drv_wait_output(LVM_ON); // Put driver to wait for response before sending next packet.
2	drv_wait_output(LVM_OFF)	<env>.drv_wait_output(LVM_OFF); // Put driver to continuously send stimulus regardless of the response from AXI slave.

PIPELINED TRANSACTION DRV_WAIT_OUTPUT(LVM_OFF)



WAIT FOR RESPONSE TRANSACTION DRV_WAIT_OUTPUT(LVM_ON)



XREADY WAIT TIME

- All the channels must wait for READY ==1 before any transaction can happen.
- This waiting time can be configured. Units supported are “ns” and “clk” only.

No	API name	Example

1	set_max_wait_awready(<total>, <unit>)	<env>.set_max_wait_awready(1000, “ns”); // can choose “clk” too
		// This sets the maximum wait time of 1000ns for AWREADY after the AWVALID is asserted.

2	set_max_wait_wready(<total>, <unit>)	<env>.set_max_wait_wready(16, “clk”); // can choose “ns” too
		// This sets the maximum wait time of 16 AXI clks for WREADY after the WVALID is asserted.

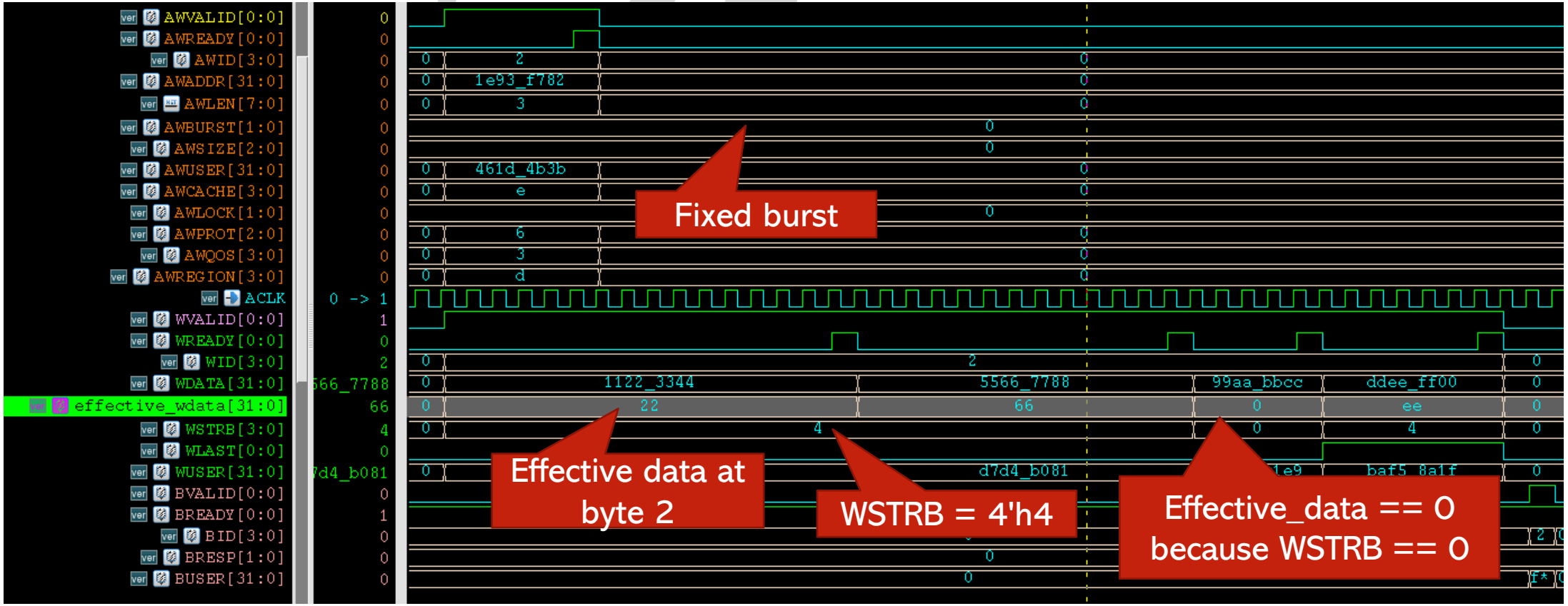
3	set_max_wait_arready(<total>, <unit>)	<env>.set_max_wait_arready(1000, “ns”); // This sets the maximum wait time of 1000ns for ARREADY after the ARVALID is asserted.
4	set_max_wait_bresp(<total>, <unit>)	<env>.set_max_wait_bresp(16, “clk”); // This sets the maximum wait time of 16 AXI clks for BRESP after the WLAST is asserted.
5	set_max_wait_rresp(<total>, <unit>)	<env>.set_max_wait_rresp(16, “clk”); // This sets the maximum wait time of 16 AXI clks for RRESP after the AR is asserted.

WRITE STROBE CONTROL

- If user does not provide WSTRB value, then UVC will randomize it complying to the protocol.
- There are 2 major settings:
  - WSTRB always all 1 (data_always_strb = 1).
  - WSTRB randomized between 1 (higher chance) and 0:
    - For example, for 16-bits data, the WSTRB can be
      - [lower 16-bits case] 4'b0000, 4'b0001, 4'b0010, 4'b0011,
      - [upper 16 bits case] 4'b0000, 4'b0100, 4'b1000, 4'b1100
- User can control this by:

```
m_axi_env.cfg.data_always_strb = 0; // default value is 1
```

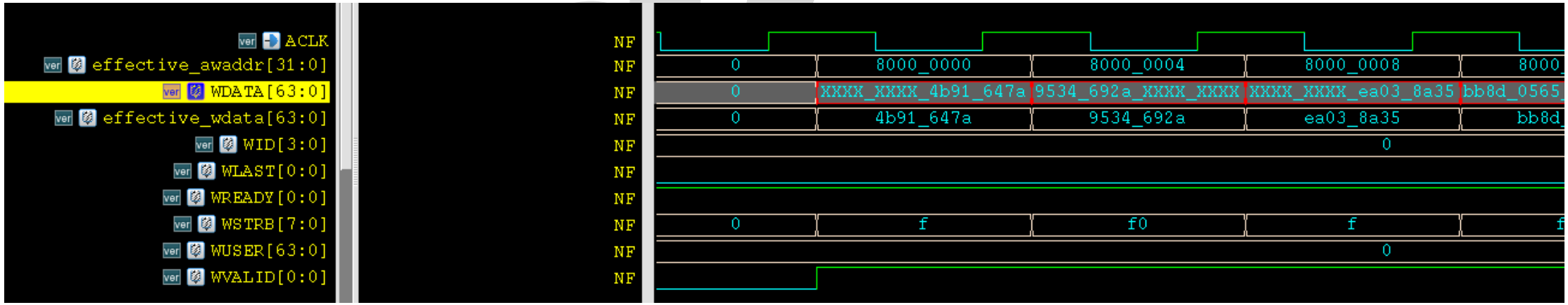
WRITE STROBE CONTROL



WDATA X INJECTION

- When WSTRB=0, the corresponding WDATA will become X
- This feature can be configured:

```
m_axi_env.cfg.inactive_lane_X_data = 1; // default value is 1
```



CARE_BOUNDARY

- This bit will define master UVC to follow the ADDR_TEST_RANGE (more details at “AXI4 testsuites” session), where the upper limit won’t be crossed.
- Default is 1
- If user needs to send packet that cross the ADDR_TEST_RANGE upper limit, then can set this bit to 0

```
m_axi_env.cfg.care_boundary = 1'b0; // default value is 1
```

WDATA_GAP_PERCENT

- This variable decides the chance in percentage that UVC can insert idle cycle during W transmission （WVALID == 1'b0).
- Default is 0 – 5%
- User can adjust the percentage as such:

```
m_axi_env.cfg.wdata_gap_percent = 10; // means it is 10% chance
```

SUPPORT_W_FIRST

- This bit will define master UVC allow write transaction to have W comes before AW.
- Default is 1
- If user DUT does not support that, then can set it to 0 like below:

```
m_axi_env.cfg.support_W_first = 1'b0; // default value is 1
```

TOTAL_W_INTERLEAVING

- This int determine how many WDATA to be interleaved at the same time.
- Default is 3

```
m_axi_env.cfg.total_w_interleaving = 3;
```

- While configuring this value, user shall configure the SVA parameter like below:

```
`define LVM_AXI_SVA_VALUES #(...,  .WDEPTH(3))
```

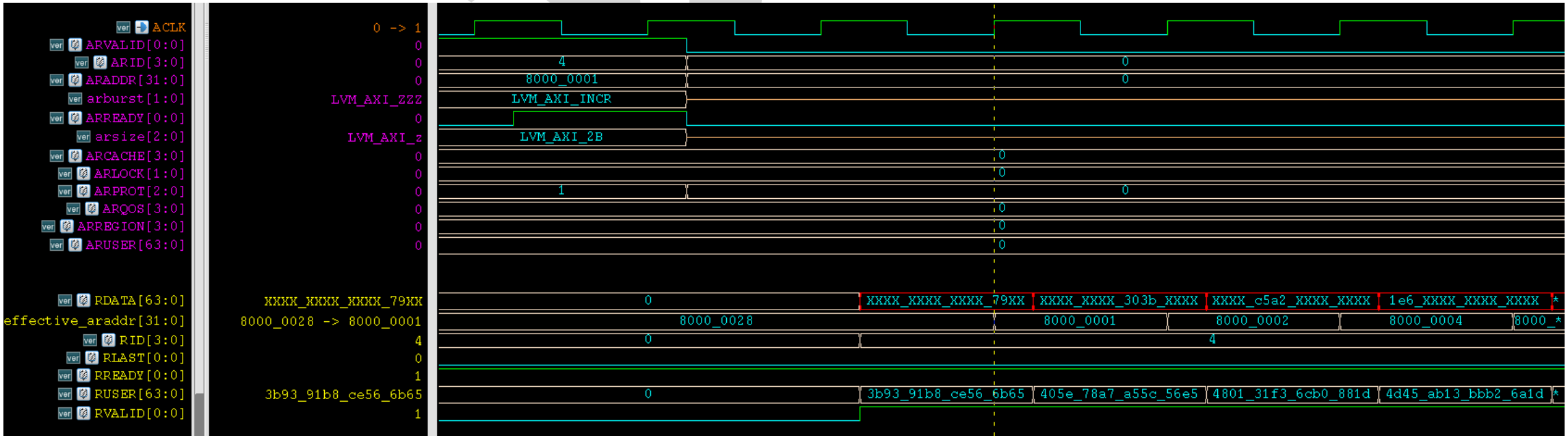
LVM AXI SLAVE CONFIGURATION

- In the self test testbench, LVM slave UVC is connected to LVM master UVC.
- It is configurable to perform different kind of responses to master.
- The note for each variables are documented inside sim/makefile under “Plusargs” keyword

RDATA X INJECTION (SLAVE)

- During read, the RDATA on the inactive byte lane will be injected with X
- This feature can be configured:

```
s_axi_env.cfg.inactive_lane_X_data = 1; // default value is 1
```



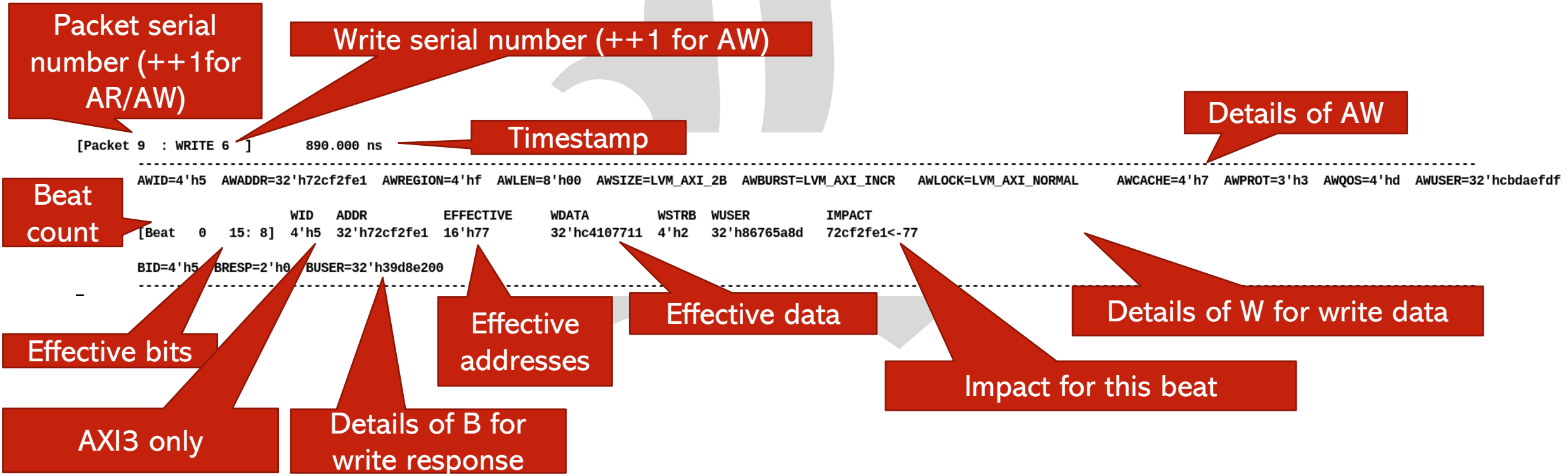
HIGH DEBUG-ABILITY

HIGH DEBUG-ABILITY UVC

- To speed up the debug process, the UVC provides its user with high visibility of AXI bus traffic.
- The simulation with monitor = ON will enable the tracker file dump
  - It is a file that contains all the AXI transactions printed systematically.
- It is very useful for the user:
  - A user who is new to this API can use this file to observe the effect of a testcode.
    - For example, the use of “grep” for the keyword “READ” lets the user know how many reads are being done.
  - During the debug process, it quickly pin-points the transaction that has problem, even before the user opens the waveform to check.
  - User can easily know what is the write data that has taken effect in each beat (auto-generated by UVC after considering WSTRB, AWSIZE, AWADDR etc.).
  - Same goes for the read data: the effective data makes it easy for the user to know the valid data value in each beat (auto-generated by UVC after considering ARSIZE, ARADDR etc.).

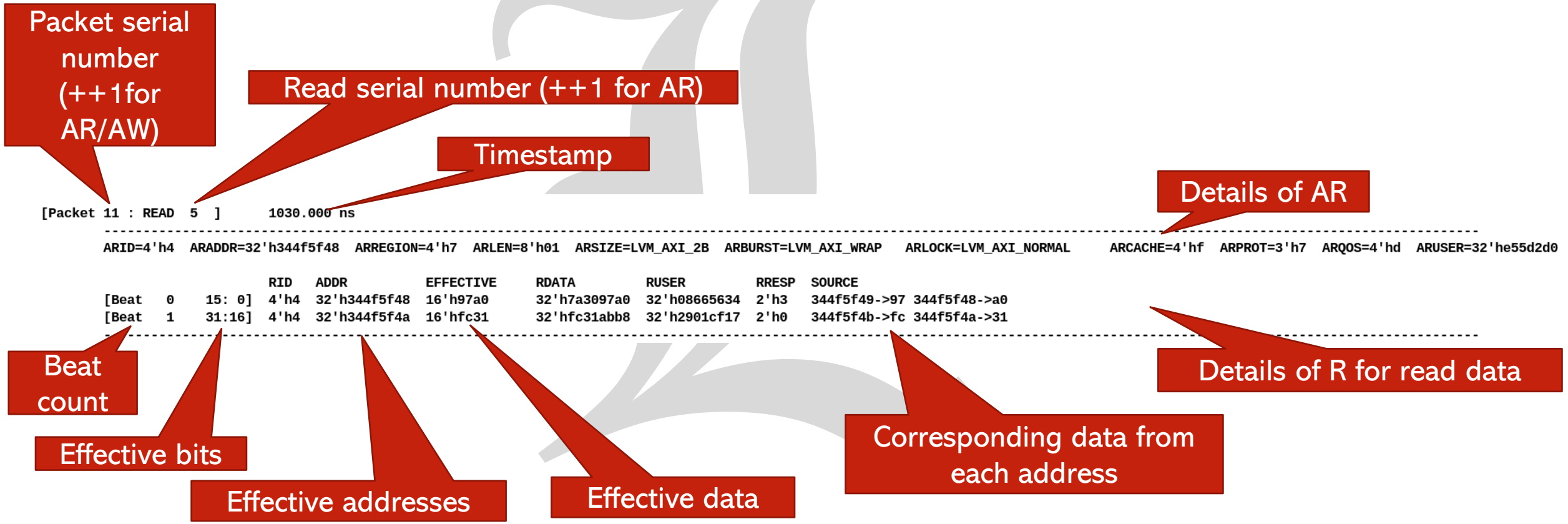
TRACKER LOG

- LVM AXI provides high debugability by preparing log file for all AXI packets that goes through the bus and are captured by the monitor.
  - This feature can be turned OFF: <env>.off_tracker.
- File name: <testname>/<testname>_<seed>.trk.log
- This path can be configured via <env>.cfg.LogFileName = <string>;



TRACKER LOG

- For read packet:



TRACKER LOG

- For both read and write, the AxPROT and AxCACHE (AXI3 and AXI4 only) will be further presented in more readable form:

```
[Packet 19 : READ  9  ]      2030.000 ns
       ------------------------------------------------------------------------------------------------------------------------------------------------
       ARID=4'h8  ARADDR=32'h486284ce  ARLEN=8'h01  ARSIZE=LVM_AXI_2B  ARBURST=LVM_AXI_INCR   ARLOCK=LVM_AXI_NORMAL     ARCACHE=4'hb  ARPROT=3'h2

                          RID   ADDR          EFFECTIVE     RDATA         RRESP  SOURCE
       [Beat   0    31:16]  4'h8  32'h486284ce  16'h4045      32'h404574fc  2'h0   486284cf->40 486284ce->45
       [Beat   1    15: 0]  4'h8  32'h486284d0  16'h87d7      32'hf3cc87d7  2'h0   486284d1->87 486284d0->d7

       +-------------------------------------------------------------------------------------------------------+
       |  ARCACHE[3]=1          ARCACHE[2]=0                ARCACHE[1]=1               ARCACHE[0]=1              |
       |  LVM_AXI_ALLOCATE      LVM_AXI_NO_OTHER_ALLOCATE   LVM_AXI_MODIFIABLE         LVM_AXI_BUFFERABLE        |
       |                                                                                                       |
       |                        ARPROT [2]=0                ARPROT [1]=1               ARPROT [0]=0              |
       |                        LVM_AXI_DATA_ACCESS         LVM_AXI_NON_SECURE_ACCESS  LVM_AXI_UNPRIVILEGED_ACCESS  |
       +-------------------------------------------------------------------------------------------------------+
```

Signal & value

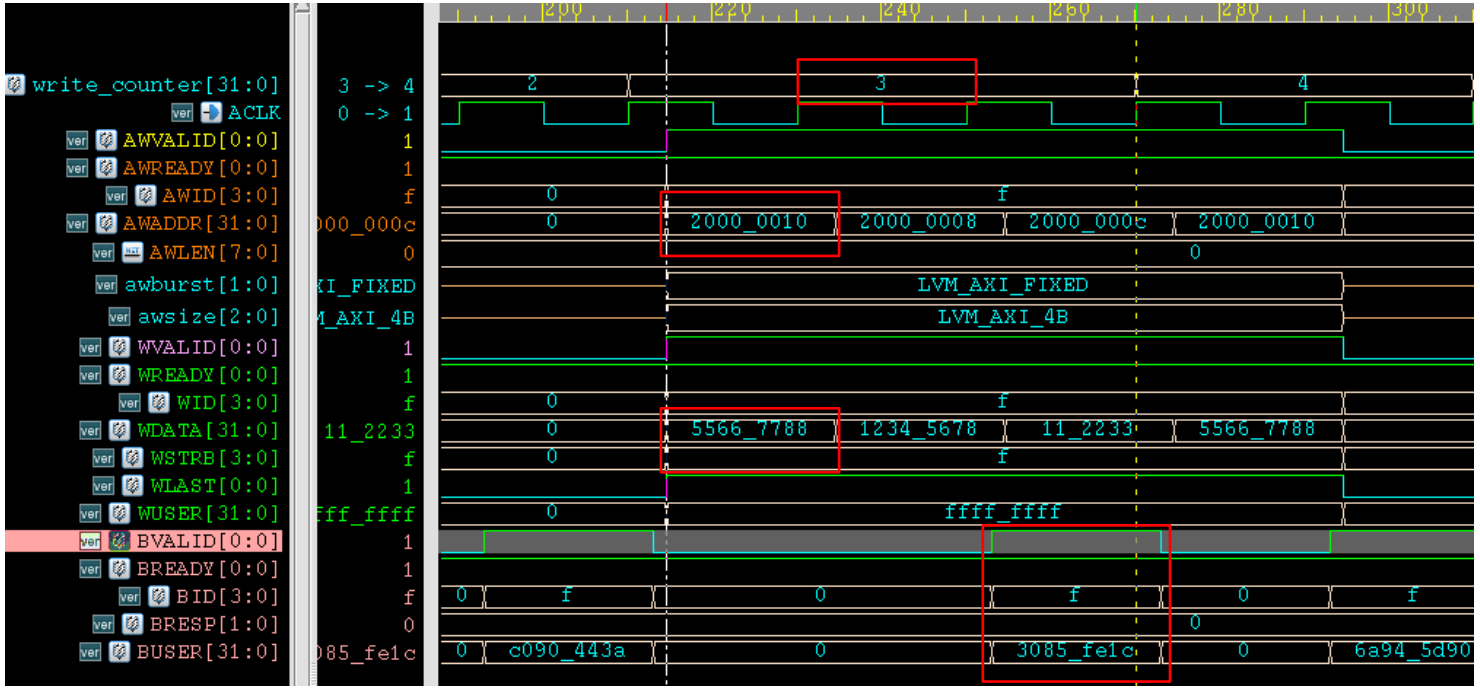
Clear name

READ AND WRITE COUNTER

- There are 2 signals to help user easily point to some specific AXI transactions in the waveform:
  - read_counter: Counts all the read packets.
  - write_counter: Counts all the write packets.

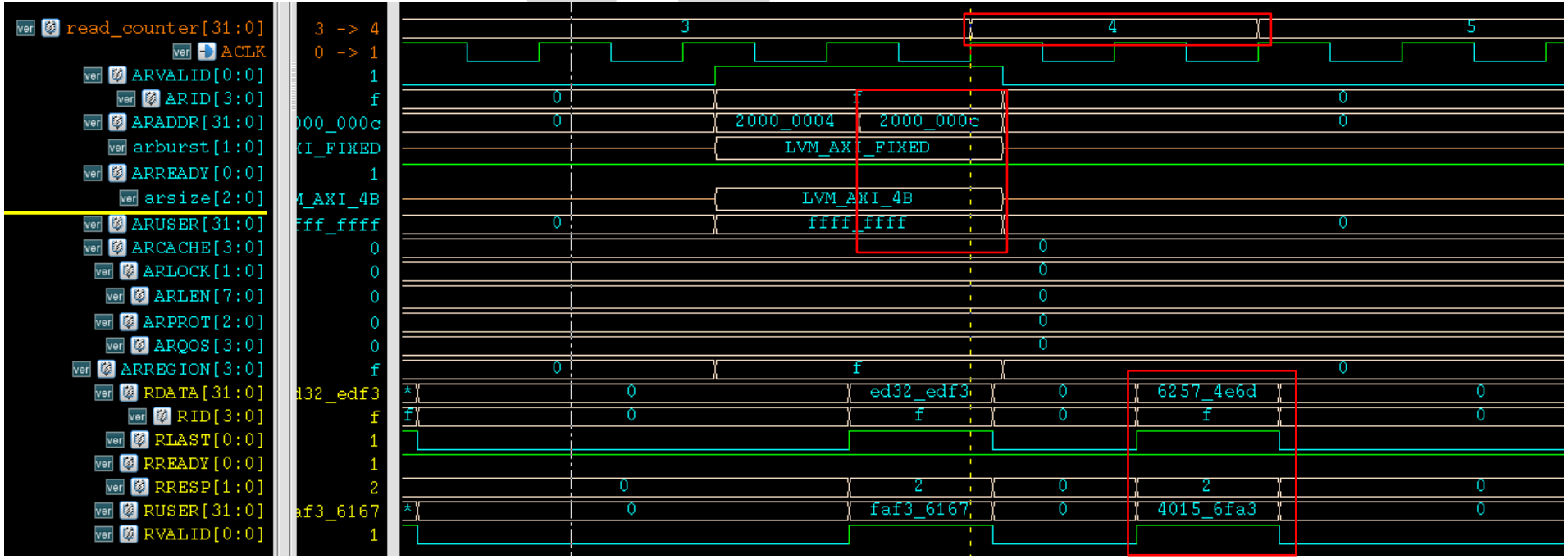
WRITE COUNTER

```
[Packet 6   : WRITE 3  ]      270.000 ns
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
AWID=4'hf  AWADDR=32'h20000010  AWREGION=4'hf  AWLEN=8'h00  AWSIZE=LVM_AXI_4B  AWBURST=LVM_AXI_FIXED  AWLOCK=LVM_AXI_NORMAL     AWCACHE=4'h0  AWPROT=3'h0  AWQOS=4'h0  AWUSER=32'hffffffff

                  ADDR          EFFECTIVE     WDATA         WSTRB  WUSER          IMPACT
[Beat    0   31: 0]  32'h20000010  32'h55667788  32'h55667788  4'hf   32'hffffffff   20000013<-55 20000012<-66 20000011<-77 20000010<-88

BID=4'hf  BRESP=2'h0  BUSER=32'h3085fe1c
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
```



READ COUNTER

```
[Packet 7  : READ  4  ]      310.000 ns
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
ARID=4'hf  ARADDR=32'h2000000c  ARREGION=4'hf  ARLEN=8'h00  ARSIZE=LVM_AXI_4B  ARBURST=LVM_AXI_FIXED  ARLOCK=LVM_AXI_NORMAL     ARCACHE=4'h0  ARPROT=3'h0  ARQOS=4'h0  ARUSER=32'hffffffff

                    RID    ADDR          EFFECTIVE     RDATA         RUSER          RRESP  SOURCE
[Beat   0    31: 0]  4'hf   32'h2000000c   32'h62574e6d  32'h62574e6d  32'h40156fa3   2'h2   2000000f->62 2000000e->57 2000000d->4e 2000000c->6d
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```



PORT WRITE EVENTS

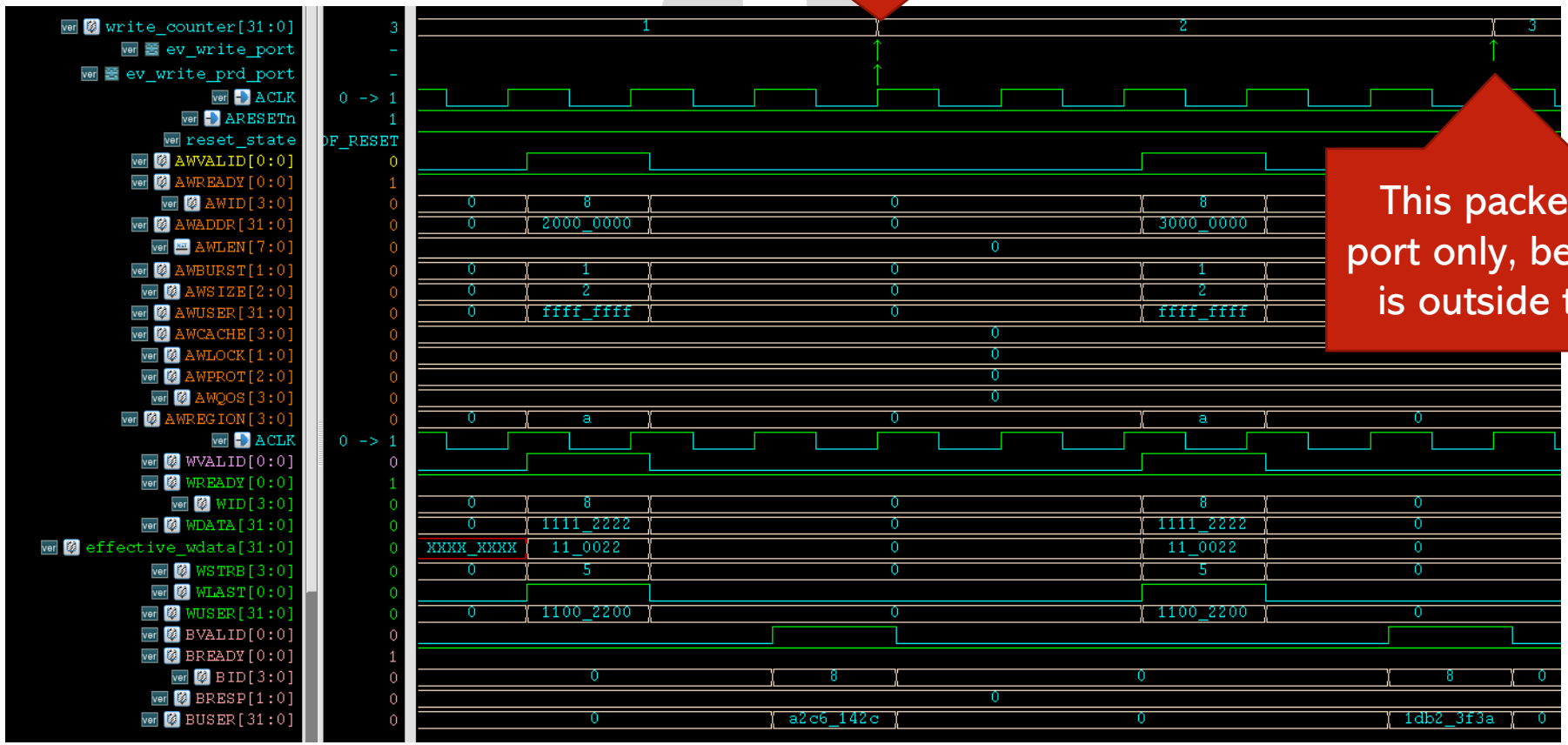
- For the user to easily identify the timing when the packet is written into env's port, the interface has the following events:

No	Event name	Purpose
1	ev_read_port	Marks the time where AXI read packet is written into env's TLM port
2	ev_read_prd_port	Marks the time where AXI read packet is sent to RAL predictor
3	ev_write_port	Marks the time where AXI write packet is written into env's TLM port
4	ev_write_prd_port	Marks the time where AXI write packet is sent to RAL predictor

PORT WRITE EVENTS

- Events for write packets

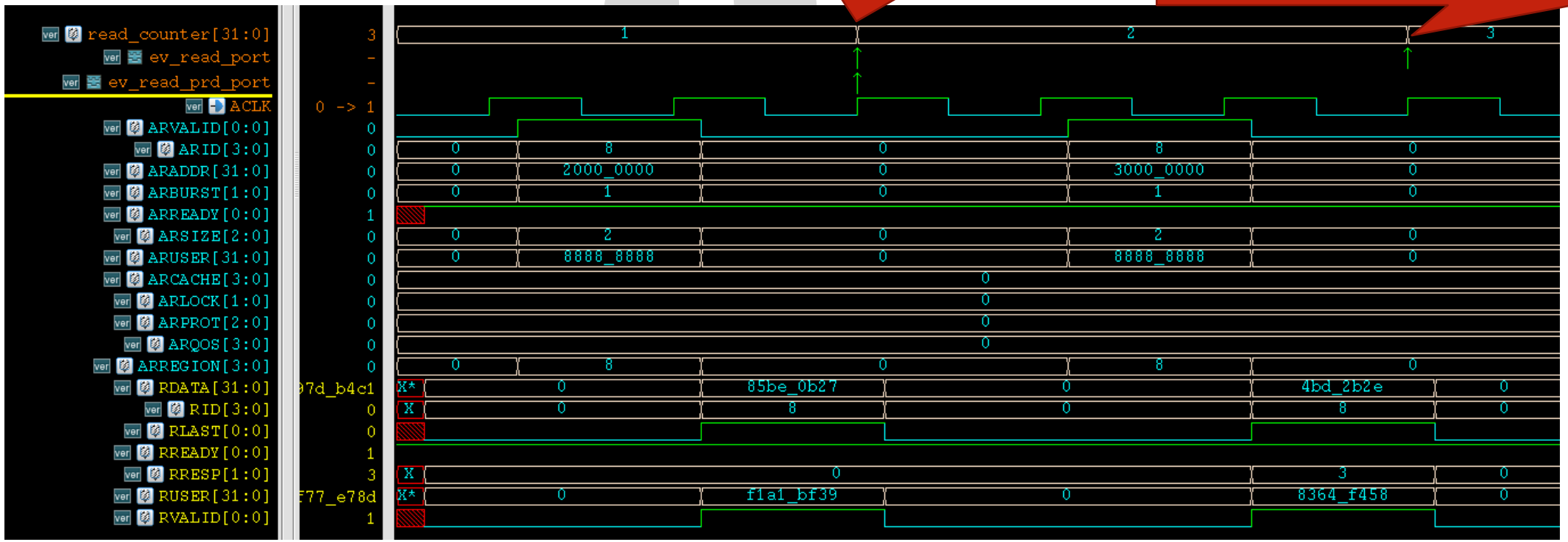
This packet is sent to env's port and also to RAL prd



This packet is sent to env's port only, because the address is outside the range of RAL

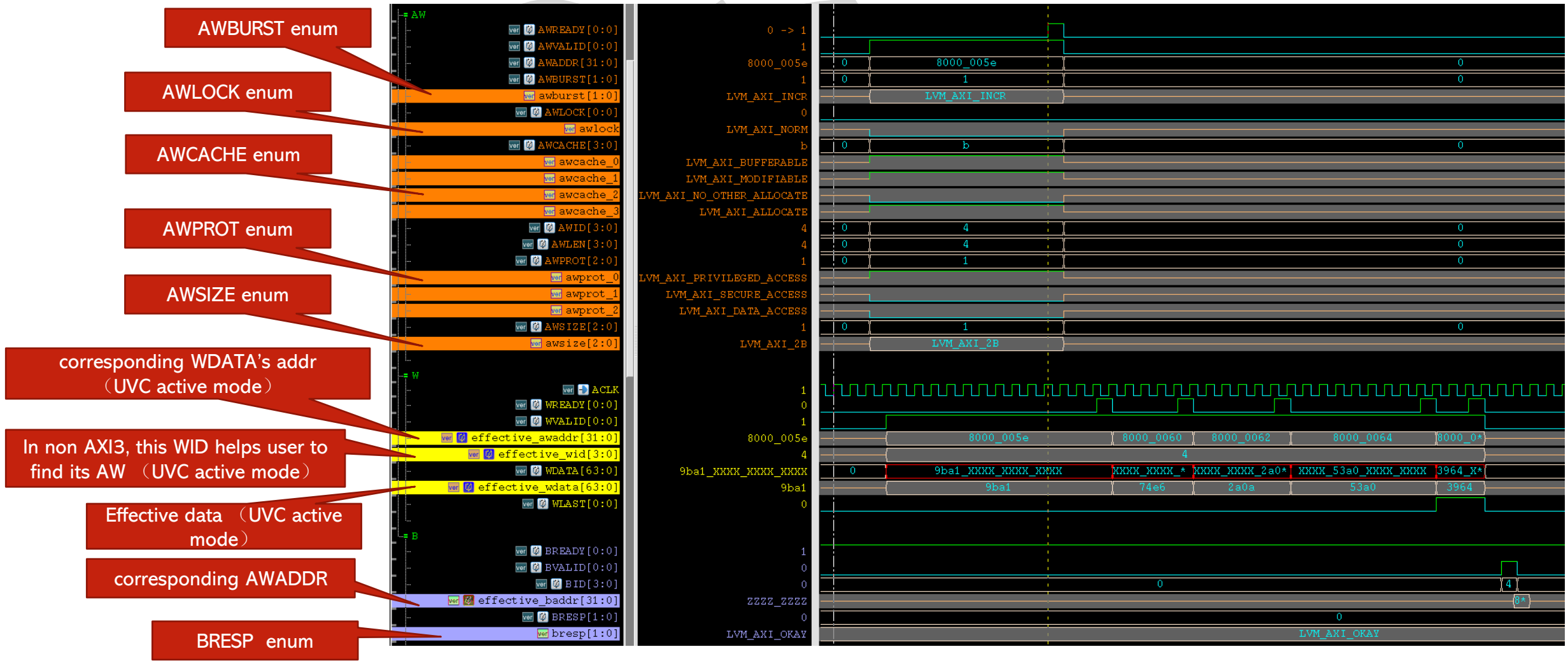
PORT WRITE EVENTS

- Events for read packets



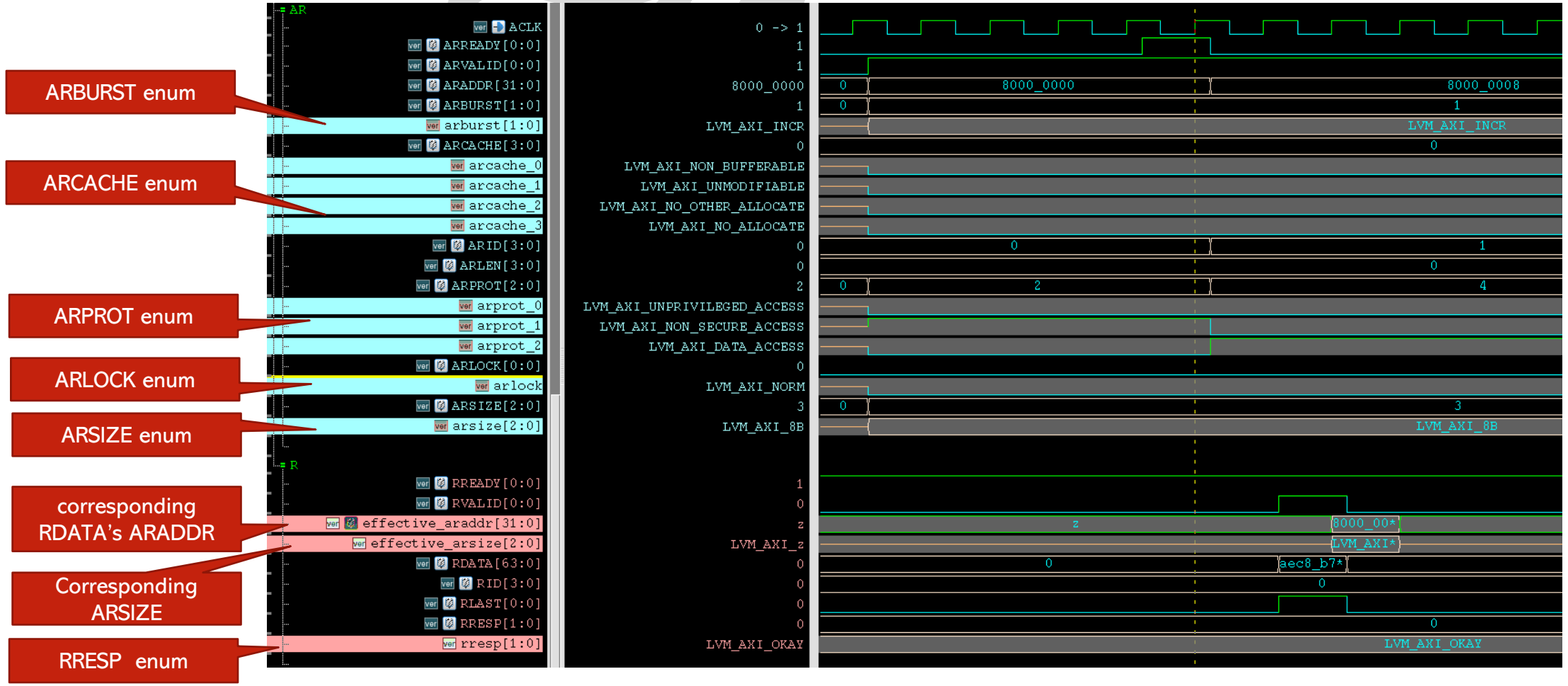
ENUM & DEBUG SIGNALS

- awburst, awsize, bresp, and more



ENUM & DEBUG SIGNALS

- arburst, arsize, rresp and more

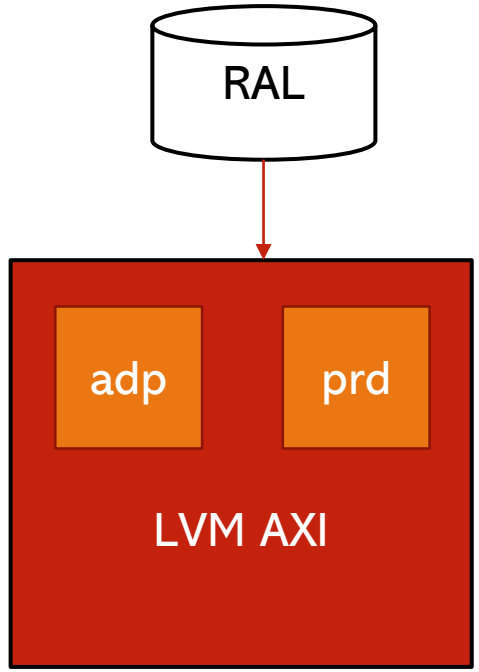


WORKING WITH RAL

RAL READY AXI UVC

- The LVM's AXI is ready to work with RAL.
- It has built-in RAL adaptor and RAL predictor.
- To connect the AXI UVC with RAL, the following is to be done at connect phase:

```
// Passing in the urm
m_axi_env.cfg.urm           = urm;

if(m_axi_env.cfg.has_prd)
    m_axi_env.prd.map       = urm.default_map;

if(m_axi_env.cfg.has_adp)
    urm.default_map.set_sequencer(m_axi_env.agt.sqr, m_axi_env.adp);
```



Ref: tests/lvm_axi_base_test.sv

RAL READ WRITE – DRV_WAIT_OUTPUT = ON

- After the connection is done, user can use the RAL to perform the read/ write access:

```
<env>.drv_wait_output(LVM_ON);

urm.<regA>.read (status, mydata); // mydata will be loaded with correct read returned data
`uvm_info(msg_tag, $sformatf("RRESP=%h", m_axi_env.cfg.ral_RRESP[0]), UVM_NONE) // retrieving RRESP

urm.<regB>.write(status, mydata);
`uvm_info(msg_tag, $sformatf("BRESP=%h", m_axi_env.cfg.ral_BRESP)    , UVM_NONE) // retrieving BRESP
// proceed only after write response is received

urm.<regB>.read (status, mydata);
// mydata is now the valid data after the write above to regB
```

- Ref: tests/lvm_axi_01_ral_access_test.sv

RAL READ WRITE – DRV_WAIT_OUTPUT = OFF

- If the user uses the drv_wait_output = LVM_OFF, the effect is as below:

```
<env>.drv_wait_output(LVM_OFF);
urm.<regA>.read (status, mydata);
    // mydata is X
urm.<regB>.write(status, mydata);
    // Without waiting, this write is launched
urm.<regB>.read (status, mydata);
    // At this point, mydata is X
    // and this packet can happen earlier than the write packet above
```

- It is always recommended for the user to use drv_wait_output == LVM_ON during RAL access.

RAL READ CONFIGURATION

- User can configure the UVC to send AXI packets with desired value for RAL generated packet.
- For example, when user enters the command urm.<regA>.read(...), the value of ARID, ARREGION, ARPROT etc. can be easily controlled by calling the API below during runtime.
- Available variables to configure for RAL's read:

No	Variable	API command
1	ARID	<env>.cfg.ral_ARID = <value>;
2	ARREGION	<env>.cfg.ral_ARREGION = <value>;
3	ARCACHE	<env>.cfg.ral_ARCACHE = <value>;
4	ARPROT	<env>.cfg.ral_ARPROT = <value>;
5	ARQOS	<env>.cfg.ral_ARQOS = <value>;
6	ARUSER	<env>.cfg.ral_ARUSER = <value>;
7	ARLOCK	<env>.cfg.ral_ARLOCK = <value>;

RAL WRITE CONFIGURATION

- Similarly, when user does urm.<regA>.write(...), the value of AWID, AWREGION, AWPROT etc can be easily controlled by calling the API below during runtime.
- Available variables to configure for RAL's write:

No	Variable	API command
1	AWID	<env>.cfg.ral_AWID = <value>;
2	AWREGION	<env>.cfg.ral_AWREGION = <value>;
3	AWCACHE	<env>.cfg.ral_AWCACHE = <value>;
4	AWPROT	<env>.cfg.ral_AWPROT = <value>;
5	AWQOS	<env>.cfg.ral_AWQOS = <value>;
6	AWUSER	<env>.cfg.ral_AWUSER = <value>;
7	AWLOCK	<env>.cfg.ral_AWLOCK = <value>;
8	WUSER	<env>.cfg.ral_WUSER = <value>;

RAL PREDICTION

- To increase the efficiency of the predictor operation, the predictor working range can be configured:
  - ral_max_addr : max address of the RAL
  - ral_min_addr : min address of the RAL

- The built-in RAL predictor works based on range specified by the user for these 2 variables:

```
 urm.default_map.set_base_addr   (32'h2000_0000);
 m_axi_env.cfg.add_RAL_ADDR_RANGE (
    .start_addr   (<ral_min_addr>),
    .end_addr     (<ral_max_addr>),
    .expected_resp({LVM_AXI_OKAY})
);
```

ADP & PRD ON/OFF

- The adp or prd can be easily turned OFF by the user if not required.
  - To turn OFF adaptor:

```
uvm_config_db#(bit)::set(this, "*m_axi_env*", "has_adp", 1'b0);
```

  - To turn OFF predictor:

```
uvm_config_db#(bit)::set(this, "*m_axi_env*", "has_prd", 1'b0);
```

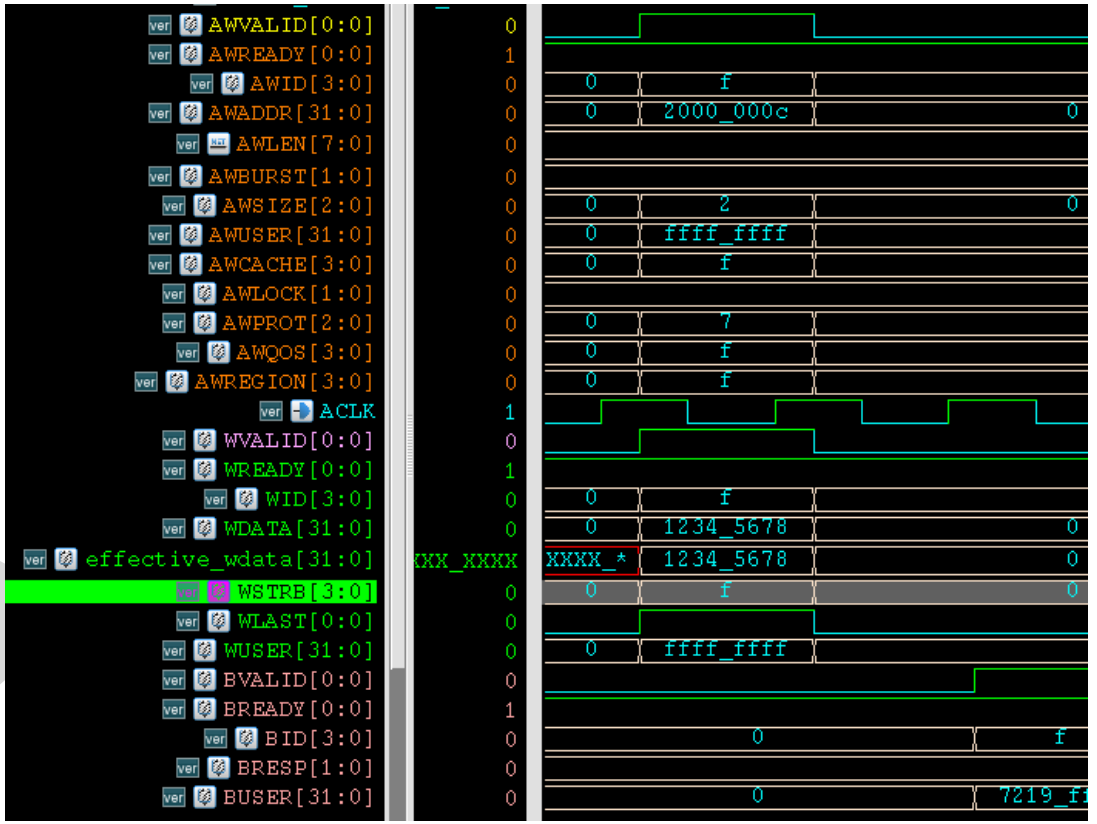
Ref: tests/lvm_axi_30_no_component_test.sv

ADDR AND DATA WIDTH CONFIGURATION

- User shall configure the UVM_REG_ADDR_WIDTH and UVM_REG_DATA_WIDTH properly matching the design.
  - At compile cmd: (Ref: sim/makefile)
    - +define+UVM_REG_DATA_WIDTH=32 +define+UVM_REG_ADDR_WIDTH=32

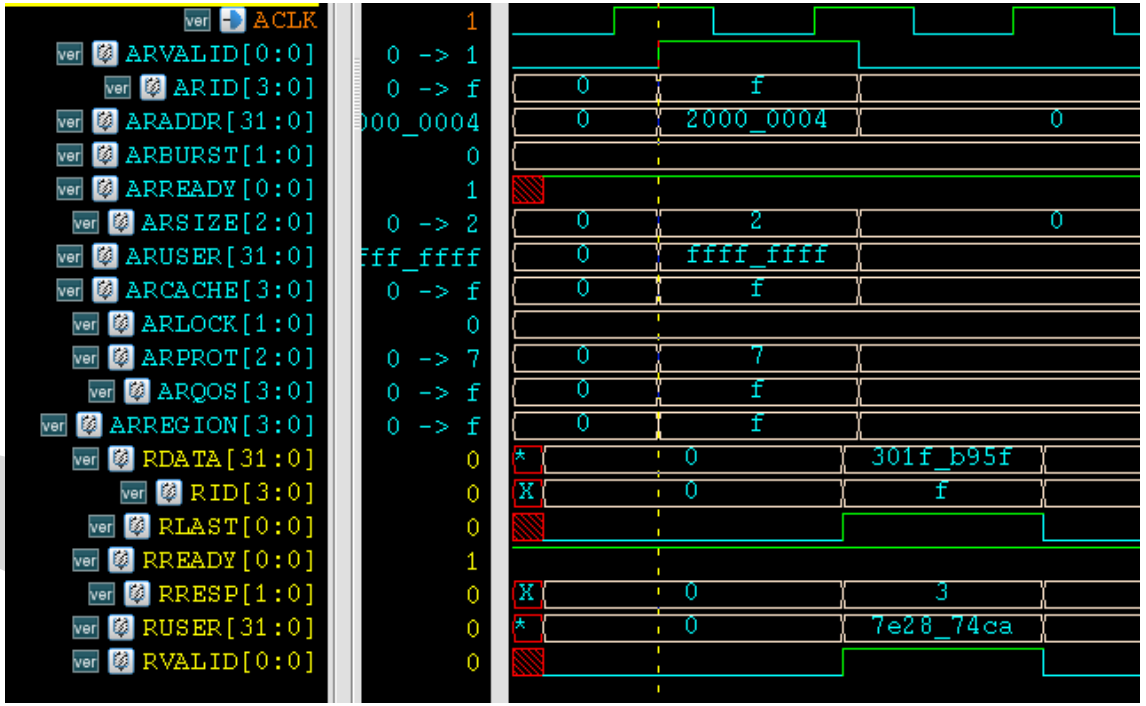
FULL REG WRITE ACCESS

- This UVC supports full register write access as shown previously:
  - `urm.<regB>.write(status, mydata);`
- It will do the AW+W for write.
- During write, WSTRB = 4'b1111 for the case data width = 32



FULL REG READ ACCESS

- This UVC support full register read access as shown previously:
  - `urm.<regA>.read (status, mydata);`
- It will do the AR for read.
- It is always full access.



PARTIAL REG WRITE ACCESS

- If user configures ral field that align with byte boundary as individual_accessible = 1, then this UVC will send out AXI packet with corresponding AxSIZE.
- For example, the field size below is 1 byte, aligned at byte lane 0:

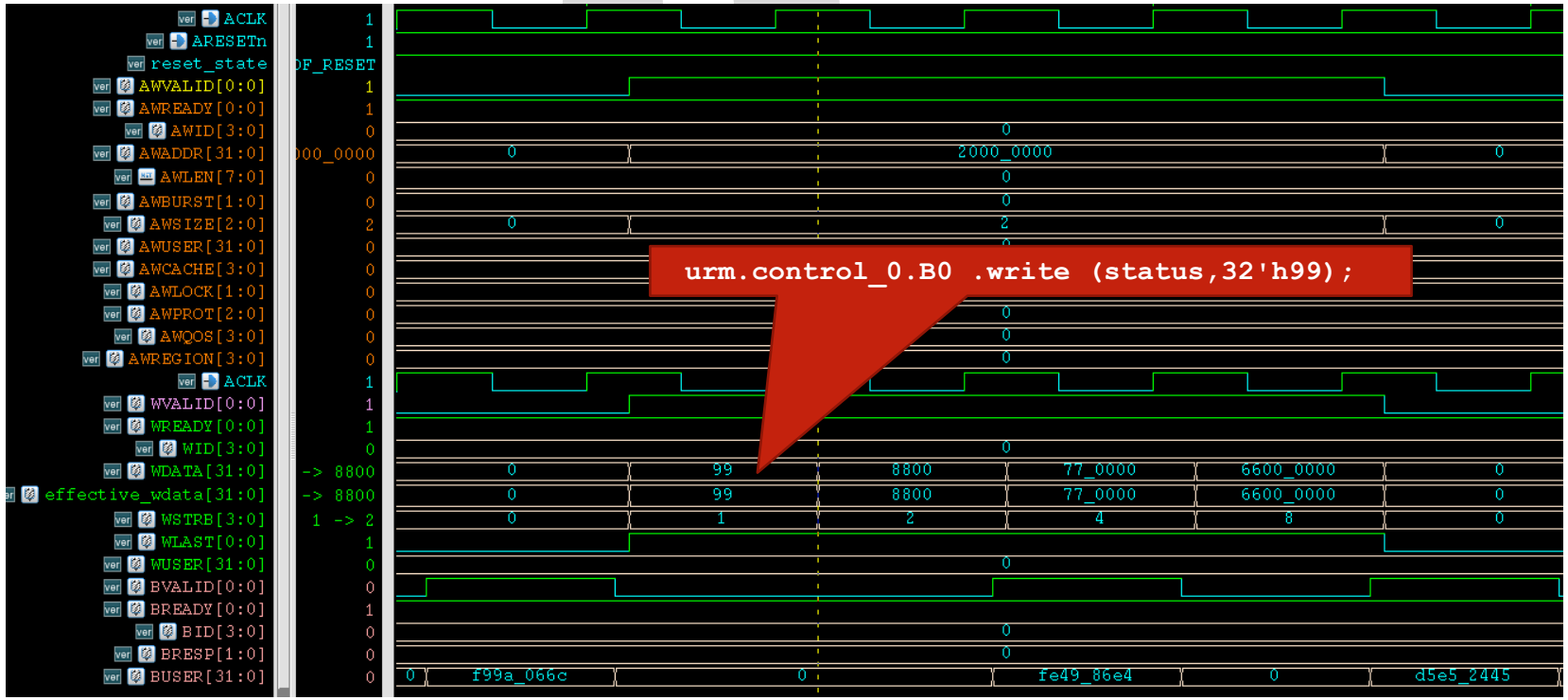
```
B0 = uvm_reg_field::type_id::create("B0");
       B0.configure(
       .parent                 (this),
       .size                   (8),
       .lsb_pos                (0),
       .access                 ("RW"),
       .volatile               (0),
       .reset                  ('0),
       .has_reset              (1),
       .is_rand                (1),
       .individually_accessible(1));
```

The size must be 8 bits

Must be individually accessible

PARTIAL REG WRITE ACCESS

- When user does the reg.field.write (normally is reg.write), it will trigger the UVC to send out W with WSTRB = 4'b0001 as below:



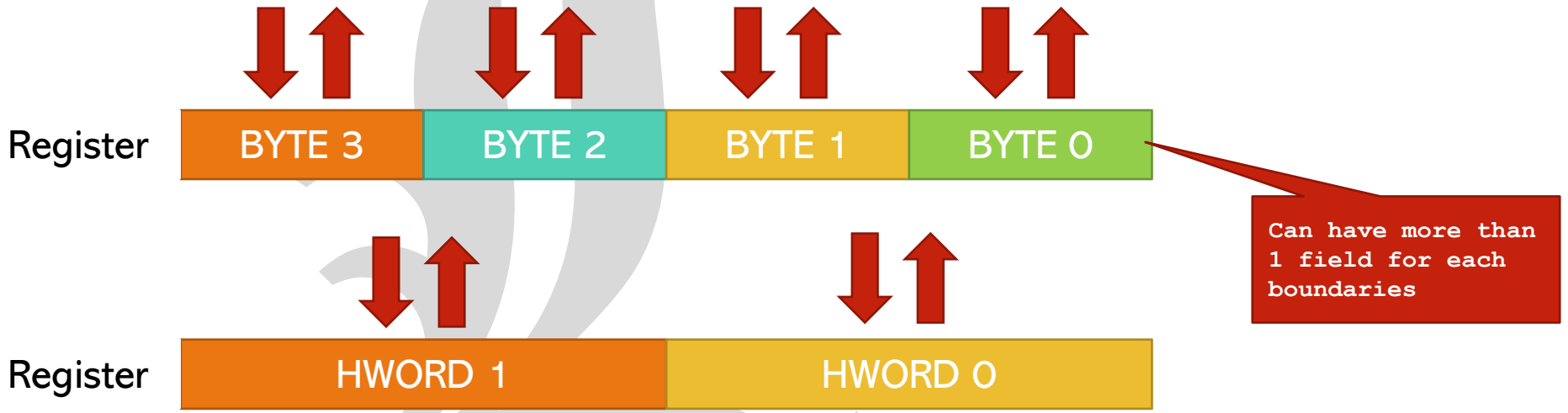
PARTIAL REG WRITE ACCESS

- This means, user can easily access various byte boundary fields, for example:
  - 8 bits, can be at byte 0, 1, 2 or 3 (provided fields are 8 bits wide and fill up whole byte)
  - 16 bits, can be byte 1_0, byte 2_1, or byte 3_2 (provided fields are 16 bits wide and fill up whole hword)
- However, if the fields are smaller than a byte, like case below, then this line cannot be used anymore to achieve byte access.

```
urm.<reg>.<field>.write (status,32'h99);
```

PARTIAL REGISTER ACCESS VERIFICATION

- To solve that problem, LVM adds the coverage for partial accesses



- It supports all IEEE field access types, while systematically verify the byte accesses, hword accesses of DUT.

PARTIAL REGISTER ACCESS VERIFICATION

- [BYTE verification] If there is/are fields that resides from bit 0 to 7, then it will do byte write to BYTE 0 with random data
  - Then byte read to BYTE 0 to confirm the effect, considering the field access (RW, RO, W1C etc)
  - Repeat above with various random data.
- Repeat for BYTE 1, 2, and 3
- If there are fields that reside from bit 0 to 15, then it will do hword write to HWORD 0 with random data
  - Then hword read to HWORD 0 to confirm the effect, considering the field access (RW, RO, W1C etc)
  - Repeat above with various random data.
- Repeat for HWORD 1

PARTIAL REGISTER ACCESS VERIFICATION

- To enable this, user just need to do so at the testcase:

```
// Example of skipping a field
urm.control_0_0C.B0.set_compare(UVM_NO_CHECK);


m_axi_env.ral_partial_access.map = urm.default_map; // connect to your desired map to verify
m_axi_env.ral_partial_access.start(null);
```

PARTIAL REGISTER ACCESS VERIFICATION

```
[lvm_ral_partial] ----< Verifying HWORD 0 access >----
[lvm_ral_partial]
[lvm_ral_partial]     LVM_HWORD 0  ADDR          VECTOR   FIELD_PATH                                                ACCESS  RESET        DESIRED      MIRRORED     VOLATILE
[lvm_ral_partial]     LVM_HWORD 0  32'h10000080  [ 0: 0]  urm.control_80.b0                                        RW       1'h0         1'h1         1'h1         0
[lvm_ral_partial]     LVM_HWORD 0  32'h10000080  [ 1: 1]  urm.control_80.b1                                        RW       1'h0         1'h1         1'h1         0
[lvm_ral_partial]     LVM_HWORD 0  32'h10000080  [ 2: 2]  urm.control_80.b2                                        RW       1'h0         1'h0         1'h0         0
[lvm_ral_partial]     LVM_HWORD 0  32'h10000080  [ 3: 3]  urm.control_80.b3                                        RW       1'h0         1'h0         1'h0         0
[lvm_ral_partial]     LVM_HWORD 0  32'h10000080  [15: 4]  urm.control_80.b15_4                                     RW      12'h0        12'hc1a      12'hc1a      0
[lvm_ral_partial]
[lvm_ral_partial]          HWORD write addr='h10000080  data='h7de3
[lvm_ral_partial] Correct! HWORD Read  addr='h10000080  data='h7de3 as expected
[lvm_ral_partial] Correct! FULL  Read  addr='h10000080  data='hc4c27de3 as expected
[lvm_ral_partial]          HWORD write addr='h10000080  data='h2623
[lvm_ral_partial] Correct! HWORD Read  addr='h10000080  data='h2623 as expected
[lvm_ral_partial] Correct! FULL  Read  addr='h10000080  data='hc4c22623 as expected
[lvm_ral_partial]          HWORD write addr='h10000080  data='h6dbe
[lvm_ral_partial] Correct! HWORD Read  addr='h10000080  data='h6dbe as expected
[lvm_ral_partial] Correct! FULL  Read  addr='h10000080  data='hc4c26dbe as expected
[lvm_ral_partial]          HWORD write addr='h10000080  data='hbe0e
[lvm_ral_partial] Correct! HWORD Read  addr='h10000080  data='hbe0e as expected
[lvm_ral_partial] Correct! FULL  Read  addr='h10000080  data='hc4c2be0e as expected
```

PARTIAL REGISTER ACCESS VERIFICATION

```
[lvm_ral_partial] ----< Verifying BYTE  3 access >----
[lvm_ral_partial]
[lvm_ral_partial]     LVM_BYTE 3  ADDR          VECTOR   FIELD_PATH                                                ACCESS  RESET       DESIRED      MIRRORED     VOLATILE
[lvm_ral_partial]     LVM_BYTE 3  32'h10000080  [28:28]  urm.control_80.b28                                        RW      1'h0        1'h0         1'h0         0
[lvm_ral_partial]     LVM_BYTE 3  32'h10000080  [29:29]  urm.control_80.b29                                        RW      1'h0        1'h0         1'h0         0
[lvm_ral_partial]     LVM_BYTE 3  32'h10000080  [30:30]  urm.control_80.b30                                        RW      1'h0        1'h0         1'h0         0
[lvm_ral_partial]     LVM_BYTE 3  32'h10000080  [31:31]  urm.control_80.b31                                        RW      1'h0        1'h0         1'h0         0
[lvm_ral_partial]
[lvm_ral_partial]          BYTE  write addr='h10000083  data='h44
[lvm_ral_partial] Correct! BYTE  Read  addr='h10000083  data='h44 as expected
[lvm_ral_partial] Correct! FULL  Read  addr='h10000080  data='h44c2c1a3 as expected
[lvm_ral_partial]
[lvm_ral_partial]          BYTE  write addr='h10000083  data='hf4
[lvm_ral_partial] Correct! BYTE  Read  addr='h10000083  data='hf4 as expected
[lvm_ral_partial] Correct! FULL  Read  addr='h10000080  data='hf4c2c1a3 as expected
[lvm_ral_partial]
[lvm_ral_partial]          BYTE  write addr='h10000083  data='hb4
[lvm_ral_partial] Correct! BYTE  Read  addr='h10000083  data='hb4 as expected
[lvm_ral_partial] Correct! FULL  Read  addr='h10000080  data='hb4c2c1a3 as expected
[lvm_ral_partial]
[lvm_ral_partial]          BYTE  write addr='h10000083  data='hc4
[lvm_ral_partial] Correct! BYTE  Read  addr='h10000083  data='hc4 as expected
[lvm_ral_partial] Correct! FULL  Read  addr='h10000080  data='hc4c2c1a3 as expected
[lvm_ral_partial]
```

BURST REGISTER ACCESS VERIFICATION

- Apart from partial register access, LVM add register burst access verification.
  - Single beat full access is fully verified at uvm's bit bashing sequence
- Example: [LEFT] 4 burst write packets (in 4 different colors) in 1B to program all the 6 x 32bits registers.
- [RIGHT] Then 3 burst read packet in 1B to read all registers and check the data to match expected values.

Register 0	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 1	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 2	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 3	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 4	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 5	BYTE 3	BYTE 2	BYTE 1	BYTE 0

Register 0	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 1	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 2	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 3	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 4	BYTE 3	BYTE 2	BYTE 1	BYTE 0
Register 5	BYTE 3	BYTE 2	BYTE 1	BYTE 0

BURST REGISTER ACCESS VERIFICATION

- Example: [LEFT] there are 3 burst packets (in 3 different colors) in 2B to program all the 6 x 32bits registers.
- [RIGHT] Then 3 burst read packet in 2B to read all registers and check the data to match expected values.

Register 0	HWORD 1	HWORD 0
Register 1	HWORD 1	HWORD 0
Register 2	HWORD 1	HWORD 0
Register 3	HWORD 1	HWORD 0
Register 4	HWORD 1	HWORD 0
Register 5	HWORD 1	HWORD 0

Register 0	HWORD 1	HWORD 0
Register 1	HWORD 1	HWORD 0
Register 2	HWORD 1	HWORD 0
Register 3	HWORD 1	HWORD 0
Register 4	HWORD 1	HWORD 0
Register 5	HWORD 1	HWORD 0

BURST REGISTER ACCESS VERIFICATION

- Example: [LEFT] there are 3 burst packets (in 3 different colors) in 4B to program all the 6 x 32bits registers.
- [RIGHT] Then 3 burst read packet in 4B to read all registers and check the data to match expected values.

BURST REGISTER ACCESS VERIFICATION

- To enable this, user just need to do so at the testcase:

```
// Example of skipping a field
urm.control_0_0C.B0.set_compare(UVM_NO_CHECK);
urm.control_0_0C.B1.set_compare(UVM_NO_CHECK);

// Sequence configuration
m_axi_env.ral_burst_access.support_partial  = 1;              // 1 means support 1B, 2B accesses (smaller than bus size access)
m_axi_env.ral_burst_access.max_beats_num    = 16;             // this is the max number of beats that the sequence will launch.

// connect the map to be verified
m_axi_env.ral_burst_access.map              = urm.default_map;  // connect to your desired map to verify

// User may let it randomized, or fix to certain desired size
m_axi_env.ral_burst_access.bytes_per_beat   = 'z;             // Randomize bytes per beat, user can put 1/2/4/8
m_axi_env.ral_burst_access.start(null);
```

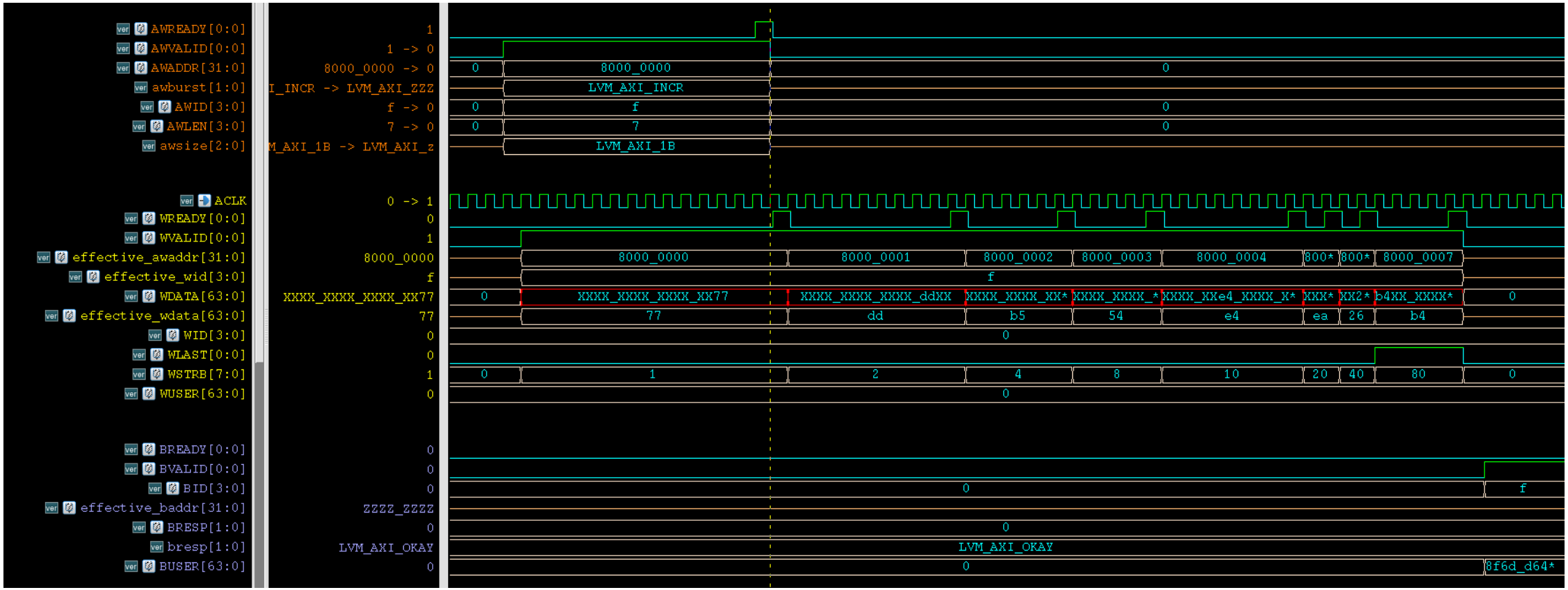
BURST REGISTER ACCESS VERIFICATION

```
[lvm_ral_burst] WRITE  Beat=0   addr='h10000060  size=LVM_HWORD  data='h31aa
[lvm_ral_burst] WRITE  Beat=1   addr='h10000062  size=LVM_HWORD  data='h170d
[lvm_ral_burst] WRITE  Beat=2   addr='h10000064  size=LVM_HWORD  data='hadb5
[lvm_ral_burst] WRITE  Beat=3   addr='h10000066  size=LVM_HWORD  data='h7dc1
[lvm_ral_burst]
[lvm_ral_burst] WRITE  Beat=0   addr='h10000068  size=LVM_HWORD  data='hccf4
[lvm_ral_burst] WRITE  Beat=1   addr='h1000006a  size=LVM_HWORD  data='h465f
[lvm_ral_burst]
[lvm_ral_burst] WRITE  Beat=0   addr='h1000006c  size=LVM_HWORD  data='h7e70
[lvm_ral_burst] WRITE  Beat=1   addr='h1000006e  size=LVM_HWORD  data='hff7f
[lvm_ral_burst]
[lvm_ral_burst] WRITE  Beat=0   addr='h10000080  size=LVM_HWORD  data='hac55
[lvm_ral_burst]
[lvm_ral_burst] WRITE  Beat=0   addr='h10000082  size=LVM_HWORD  data='hea04
[lvm_ral_burst]
[lvm_ral_burst] READ   Beat=0   addr='h10000000  size=LVM_HWORD  data=7024
[lvm_ral_burst] READ   Beat=1   addr='h10000002  size=LVM_HWORD  data=b3fa
[lvm_ral_burst] READ   Beat=2   addr='h10000004  size=LVM_HWORD  data=4401
[lvm_ral_burst] READ   Beat=3   addr='h10000006  size=LVM_HWORD  data=a3eb
[lvm_ral_burst] READ   Beat=4   addr='h10000008  size=LVM_HWORD  data=2b82
[lvm_ral_burst] READ   Beat=5   addr='h1000000a  size=LVM_HWORD  data=e5c2
[lvm_ral_burst] READ   Beat=6   addr='h1000000c  size=LVM_HWORD  data=8797
[lvm_ral_burst] READ   Beat=7   addr='h1000000e  size=LVM_HWORD  data=454
[lvm_ral_burst] READ   Beat=8   addr='h10000010  size=LVM_HWORD  data=ef73
[lvm_ral_burst] READ   Beat=9   addr='h10000012  size=LVM_HWORD  data=53e1
[lvm_ral_burst]
```

Example of multiple burst writes in HWORD size and covering 1 and more registers

Example of 1 burst read in HWORD size and covering more registers

BURST REGISTER ACCESS VERIFICATION



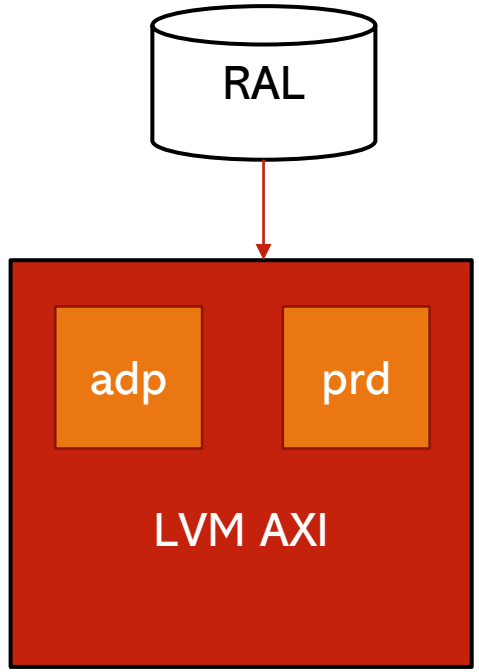
WORKING WITH RAL

RAL READY AXI UVC

- The LVM's AXI is ready to work with RAL.
- It has built-in RAL adaptor and RAL predictor.
- To connect the AXI UVC with RAL, the following is to be done at connect phase:

```
// Passing in the urm
m_axi_env.cfg.urm          = urm;
if(m_axi_env.cfg.has_prd)
    m_axi_env.prd.map      = urm.default_map;

if(m_axi_env.cfg.has_adp)
    urm.default_map.set_sequencer(m_axi_env.agt.sqr, m_axi_env.adp);
```

Ref: tests/lvm_axi_base_test.sv



RAL READ WRITE – DRV_WAIT_OUTPUT = ON

- After the connection is done, user can use the RAL to perform the read/ write access:

```
<env>.drv_wait_output(LVM_ON);

urm.<regA>.read (status, mydata); // mydata will be loaded with correct read returned data
`uvm_info(msg_tag, $sformatf("RRESP=%h", m_axi_env.cfg.ral_RRESP[0]), UVM_NONE) // retrieving RRESP

urm.<regB>.write(status, mydata);
`uvm_info(msg_tag, $sformatf("BRESP=%h", m_axi_env.cfg.ral_BRESP)    , UVM_NONE) // retrieving BRESP
// proceed only after write response is received

urm.<regB>.read (status, mydata);
// mydata is now the valid data after the write above to regB
```

- Ref: tests/lvm_axi_01_ral_access_test.sv

RAL READ WRITE – DRV_WAIT_OUTPUT = OFF

- If the user uses the drv_wait_output = LVM_OFF, the effect is as below:

```
<env>.drv_wait_output(LVM_OFF);
urm.<regA>.read (status, mydata);
    // mydata is X
urm.<regB>.write(status, mydata);
    // Without waiting, this write is launched
urm.<regB>.read (status, mydata);
    // At this point, mydata is X
    // and this packet can happen earlier than the write packet above
```

- It is always recommended for the user to use drv_wait_output == LVM_ON during RAL access.

RAL READ CONFIGURATION

- User can configure the UVC to send AXI packets with desired value for RAL generated packet.
- For example, when user enters the command urm.<regA>.read(...), the value of ARID, ARREGION, ARPROT etc. can be easily controlled by calling the API below during runtime.
- Available variables to configure for RAL's read:

No	Variable	API command
1	ARID	<env>.cfg.ral_ARID = <value>;
2	ARREGION	<env>.cfg.ral_ARREGION = <value>;
3	ARCACHE	<env>.cfg.ral_ARCACHE = <value>;
4	ARPROT	<env>.cfg.ral_ARPROT = <value>;
5	ARQOS	<env>.cfg.ral_ARQOS = <value>;
6	ARUSER	<env>.cfg.ral_ARUSER = <value>;
7	ARLOCK	<env>.cfg.ral_ARLOCK = <value>;

RAL WRITE CONFIGURATION

- Similarly, when user does urm.<regA>.write(...), the value of AWID, AWREGION, AWPROT etc can be easily controlled by calling the API below during runtime.
- Available variables to configure for RAL's write:

No	Variable	API command
1	AWID	<env>.cfg.ral_AWID = <value>;
2	AWREGION	<env>.cfg.ral_AWREGION = <value>;
3	AWCACHE	<env>.cfg.ral_AWCACHE = <value>;
4	AWPROT	<env>.cfg.ral_AWPROT = <value>;
5	AWQOS	<env>.cfg.ral_AWQOS = <value>;
6	AWUSER	<env>.cfg.ral_AWUSER = <value>;
7	AWLOCK	<env>.cfg.ral_AWLOCK = <value>;
8	WUSER	<env>.cfg.ral_WUSER = <value>;

RAL PREDICTION

- To increase the efficiency of the predictor operation, 2 array variables can be programmed:
  - ral_max_addr : to tell the max address of the RAL
  - ral_min_addr : to tell the min address of the RAL

- The built-in RAL predictor works based on range specified by the user for these 2 variables:

```
urm.default_map.set_base_addr   (32'h2000_0000);
 m_axi_env.cfg.add_RAL_ADDR_RANGE (
    .start_addr   (<ral_min_addr>),
    .end_addr     (<ral_max_addr>),
    .expected_resp({LVM_AXI_OKAY})
);
```

RAL PREDICTION

- RAL predictor works perfectly even on unaligned AXI transactions, example unaligned 64 bits AXI write as below:

```
WDATA     = new[2];
WDATA[0]  = 64'h4444_4444_1111_1188;
WDATA[1]  = 64'h3333_2222_aaaa_bbbb;
m_axi_env.write(
    .AWADDR(urm.control_0_00.get_address+1),.AWLEN(1),.AWSIZE(LVM_AXI_8B),.AWBURST(LVM_AXI_INCR),.AWLOCK('0),.WDATA(WDATA),
    .BID(BID),.BRESP(BRESP),.BUSER(BUSER)
);
```

- Tracker log:

```
AWID=4'h5  AWADDR=32'h80000001  AWLEN=3'h1  AWSIZE=LVM_AXI_8B  AWBURST=LVM_AXI_INCR   AWLOCK=LVM_AXI_NORMAL     AWCACHE=4'ha  AWPROT=3'h3
                    WID   ADDR          EFFECTIVE             WDATA                 WSTRB  IMPACT
[Beat   0   63: 8]  4'h5  32'h80000001  64'h44444444111111    64'h44444444111111xx  8'hfe  80000007<-44 80000006<-44 80000005<-44 80000004<-44 80000003<-11 80000002<-11 80000001<-11
[Beat   1   63: 0]  4'h5  32'h80000008  64'h33332222aaaabbbb  64'h33332222aaaabbbb  8'hff  8000000f<-33 8000000e<-33 8000000d<-22 8000000c<-22 8000000b<-aa 8000000a<-aa 80000009<-bb 80000008<-
bb
BID=4'h5  BRESP=LVM_AXI_OKAY
```

RAL PREDICTION

```
[REG_PREDICT] Observed WRITE transaction to register urm.control_0_00 : before='haabbccdd  incoming='h11111100 (byte_en='he)  updated_value='h111111dd
[REG_PREDICT]     ADDR          VECTOR   FIELD_PATH             ACCESS  RESET         BEFORE        MIRRORED
[REG_PREDICT]     32'h80000000  [31:24]  urm.control_0_00.B3    RW       8'h0          8'haa         8'h11         <------
[REG_PREDICT]     32'h80000000  [23:16]  urm.control_0_00.B2    RW       8'h0          8'hbb         8'h11         <------
[REG_PREDICT]     32'h80000000  [15: 8]  urm.control_0_00.B1    RW       8'h0          8'hcc         8'h11         <------
[REG_PREDICT]     32'h80000000  [ 7: 0]  urm.control_0_00.B0    RW       8'h0          8'hdd         8'hdd
[REG_PREDICT] Observed WRITE transaction to register urm.control_0_04 : before='h66665555  incoming='h44444444 (byte_en='hf)  updated_value='h44444444
[REG_PREDICT]     ADDR          VECTOR   FIELD_PATH             ACCESS  RESET         BEFORE        MIRRORED
[REG_PREDICT]     32'h80000004  [31:24]  urm.control_0_04.B3    RW       8'h0          8'h66         8'h44         <------
[REG_PREDICT]     32'h80000004  [23:16]  urm.control_0_04.B2    RW       8'h0          8'h66         8'h44         <------
[REG_PREDICT]     32'h80000004  [15: 8]  urm.control_0_04.B1    RW       8'h0          8'h55         8'h44         <------
[REG_PREDICT]     32'h80000004  [ 7: 0]  urm.control_0_04.B0    RW       8'h0          8'h55         8'h44         <------
[REG_PREDICT] Observed WRITE transaction to register urm.control_0_08 : before='he628b7d5  incoming='haaaabbbb (byte_en='hf)  updated_value='haaaabbbb
[REG_PREDICT]     ADDR          VECTOR   FIELD_PATH             ACCESS  RESET         BEFORE        MIRRORED
[REG_PREDICT]     32'h80000008  [31:24]  urm.control_0_08.B3    RW       8'h0          8'he6         8'haa         <------
[REG_PREDICT]     32'h80000008  [23:16]  urm.control_0_08.B2    RW       8'h0          8'h28         8'haa         <------
[REG_PREDICT]     32'h80000008  [15: 8]  urm.control_0_08.B1    RW       8'h0          8'hb7         8'hbb         <------
[REG_PREDICT]     32'h80000008  [ 7: 0]  urm.control_0_08.B0    RW       8'h0          8'hd5         8'hbb         <------
[REG_PREDICT] Observed WRITE transaction to register urm.control_0_0C : before='hed6a4eb4  incoming='h33332222 (byte_en='hf)  updated_value='h33332222
[REG_PREDICT]     ADDR          VECTOR   FIELD_PATH             ACCESS  RESET         BEFORE        MIRRORED
[REG_PREDICT]     32'h8000000c  [31:24]  urm.control_0_0C.B3    RW       8'h0          8'hed         8'h33         <------
[REG_PREDICT]     32'h8000000c  [23:16]  urm.control_0_0C.B2    RW       8'h0          8'h6a         8'h33         <------
[REG_PREDICT]     32'h8000000c  [15: 8]  urm.control_0_0C.B1    RW       8'h0          8'h4e         8'h22         <------
[REG_PREDICT]     32'h8000000c  [ 7: 0]  urm.control_0_0C.B0    RW       8'h0          8'hb4         8'h22         <------
```

ADP & PRD ON/OFF

- The adp or prd can be easily turned OFF by the user if not required.
  - To turn OFF adaptor:

```
uvm_config_db#(bit)::set(this, "*m_axi_env*", "has_adp", 1'b0);
```

  - To turn OFF predictor:

```
uvm_config_db#(bit)::set(this, "*m_axi_env*", "has_prd", 1'b0);
```

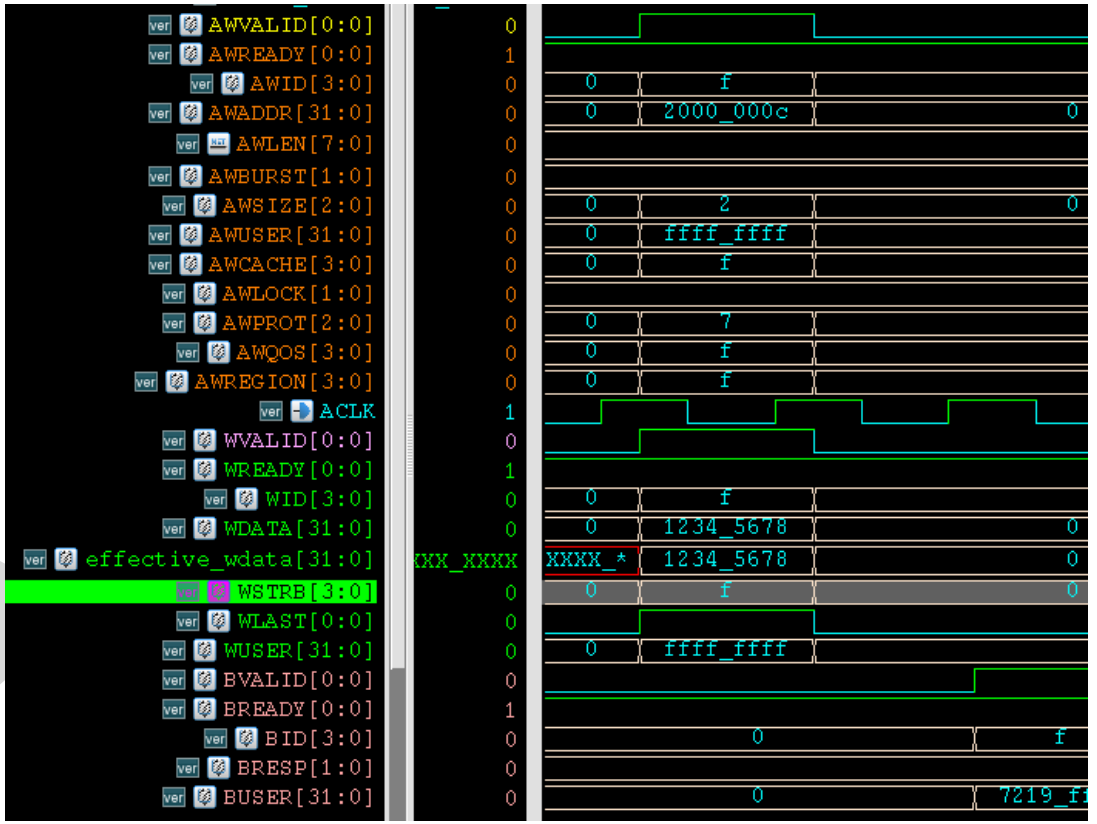
Ref: tests/lvm_axi_30_no_component_test.sv

ADDR AND DATA WIDTH CONFIGURATION

- User shall configure the UVM_REG_ADDR_WIDTH and UVM_REG_DATA_WIDTH properly matching the design.
  - At compile cmd: (Ref: sim/makefile)
    - +define+UVM_REG_DATA_WIDTH=32 +define+UVM_REG_ADDR_WIDTH=32

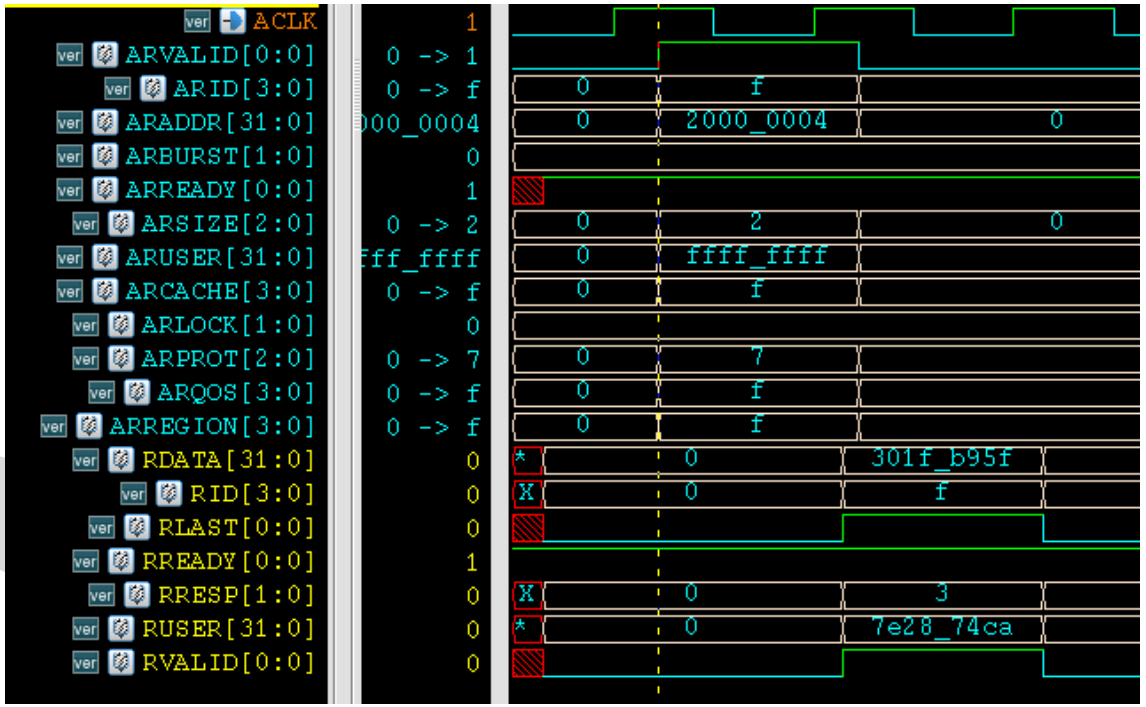
FULL REG WRITE ACCESS

- This UVC supports full register write access as shown previously:
  - `urm.<regB>.write(status, mydata);`
- It will do the AW+W for write.
- During write, WSTRB = 4'b1111 for the case data width = 32



FULL REG READ ACCESS

- This UVC support full register read access as shown previously:
  - `urm.<regA>.read (status, mydata);`
- It will do the AR for read.
- It is always full access.



PARTIAL REG WRITE ACCESS

- If user configures ral field that align with byte boundary as individual_accessible = 1, then this UVC will send out W with corresponding WSTRB.
- For example, the field size below is 1 byte, aligned at byte lane 0:

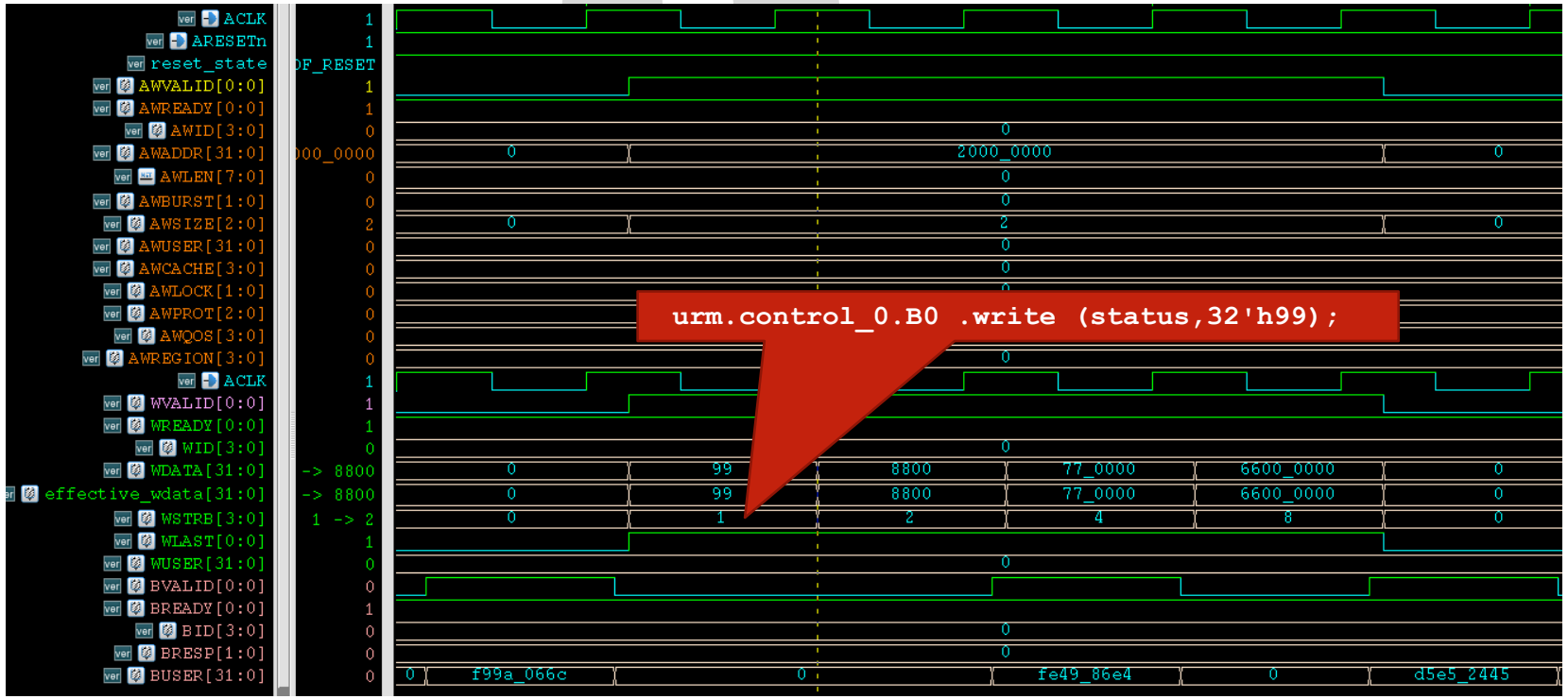
```
B0 = uvm_reg_field::type_id::create("B0");
       B0.configure(
       .parent                 (this),
       .size                   (8),
       .lsb_pos                (0),
       .access                 ("RW"),
       .volatile               (0),
       .reset                  ('0),
       .has_reset              (1),
       .is_rand                (1),
       .individually_accessible(1));
```

The size must be 8 bits

Must be individually accessible

PARTIAL REG WRITE ACCESS

- When user does the reg.field.write (normally is reg.write), it will trigger the UVC to send out W with WSTRB = 4'b0001 as below:



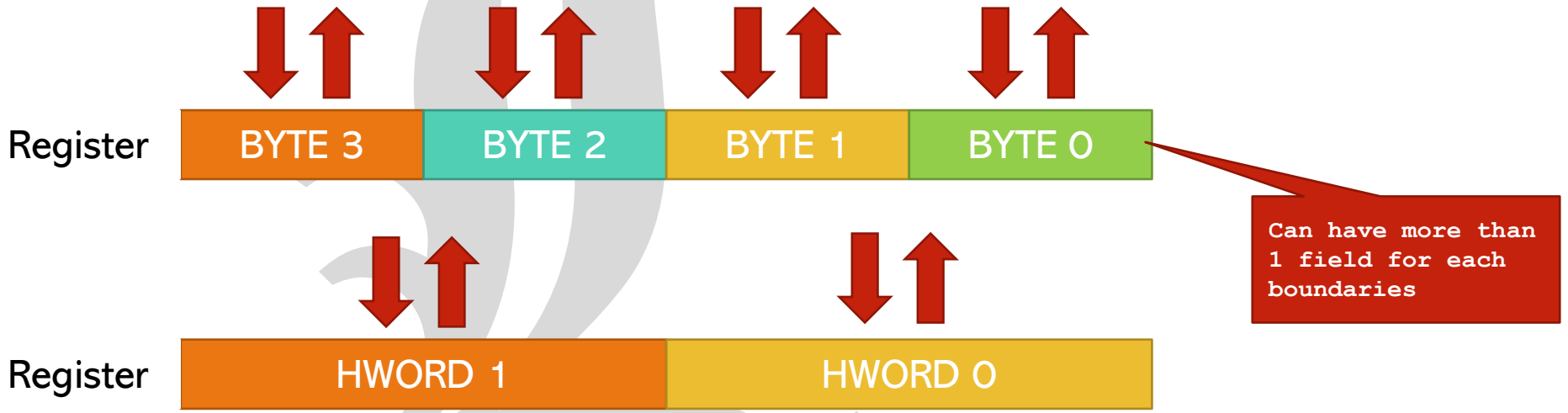
PARTIAL REG WRITE ACCESS

- This means, user can easily access various byte boundary fields, for example:
  - 8 bits, can be at byte 0, 1, 2 or 3 (provided fields are 8 bits wide and fill up whole byte)
  - 16 bits, can be byte 1_0, byte 2_1,or byte 3_2 (provided fields are 16 bits wide and fill up whole hword)
- On the other hand, if the fields are smaller than a byte, for example like case below, then this line cannot be used anymore to achieve byte access.

```
urm.<reg>.<field>.write (status,32'h99);
```

PARTIAL REGISTER ACCESS VERIFICATION

- To solve that problem, LVM adds the coverage for partial accesses



- It supports all IEEE field access types, while systematically verify the byte accesses, hword accesses of DUT.

PARTIAL REGISTER ACCESS VERIFICATION

- If there is/are fields that resides from bit 0 to 7, then it will do byte write to BYTE 0 with random data
  - Then byte read to BYTE 0 to confirm the effect, considering the field access (RW, RO, W1C etc)
  - Repeat above with various random data.
- Repeat for BYTE 1, 2, and 3
- If there are fields that reside from bit 0 to 15, then it will do hword write to HWORD 0 with random data
  - Then hword read to HWORD 0 to confirm the effect, considering the field access (RW, RO, W1C etc)
  - Repeat above with various random data.
- Repeat for HWORD 1

PARTIAL REGISTER ACCESS VERIFICATION

- To enable this, user just need to do so at the testcase:

```
// Example of skipping a field
urm.control_0_0C.B0.set_compare(UVM_NO_CHECK);


m_axi_env.ral_partial_access.map = urm.default_map; // connect to your desired map to verify
m_axi_env.ral_partial_access.start(null);
```

BACKGROUND AXI READ DATA CHECK

BACKGROUND AXI READ DATA CHECK

- For all the writes with address fall within user specific range, this UVC will capture the impact of the write data, considering all write packet parameters, like AWADDR, AWSIZE, WSTRB etc.
- For example, user set UVC to be as such:

```
m_axi_env.cfg.got_mem        = 1'b1;  // this bit turn on the background data check mechanism
m_axi_env.cfg.add_ADDR_RANGE(.start_addr(32'h1000_0000), .end_addr(32'h1000_1000));
// memory start & end addresses (support >1 entry)
```

- This means all the AW+W with address range falls between **`32'h1000_0000`** and **`32'h1000_1000`** will cause UVC to update embedded memory.
- The data will be the expected data when the read happens for the address within the range.
- UVC will check each data when user set **`got_mem == 1`**, and uvm_error will be flagged if data is mismatched.

BACKGROUND AXI READ DATA CHECK

- If user has initial value for the memory, then can update the UVC scoreboard (backdoor) first using the following API:

```
for(bit [31:0] addr=32'h1000_0000 ; addr<=32'h1000_0FFF ; addr+=4)
    m_axi_env.cfg.mem.store_4_data(addr,32'h0000_0000); // Addr and Data
```

- Then the write impact and read check will be working based on these initialized values.
- Note:
  - store_8_data (8 bytes), store_2_data (2 bytes) and store_1_data (1 byte) can be used as well

FRONTDOOR MEMORY INITIALIZATION

- To verify AXI memory, user commonly will initialize the memory with random value, via frontdoor write.
- The API below will initialize the memory within the range with AXI writes (multiple single beat writes)

```
m_axi_env.frontdoor_init_mem;
```

END OF TEST MEMORY PRINTING

- At the end of test, the UVC can be programmed to print the full content of the memory when

```
m_axi_env.cfg.got_mem = 1'b1;
```

- It is based on the writes that happen throughout the test.

```
MEMORY TRACKER
+------+----------+----------------+----------------+----------------+----------------+
|   NO |   ADDR   |  f   e   d   c |  b   a   9   8 |  7   6   5   4 |  3   2   1   0 |
+------+----------+----------------+----------------+----------------+----------------+
|    1 | 00000000 | 9B  8C  80  06 | A8  D5  A6  56 | 49  87  96  6B | 72  8D  B6  39 |
|    2 | 00000010 |                |                | E6  60  44  49 | 1B  50  47  1E |
|    3 | 01000000 | D6  45  19  8A | 43  DE  DA  B1 | E1  1C  14  80 | 50  A9  6E  69 |
|    4 | 01000010 |                |                | 46  63  9E  5B | 6E  4E  BF  37 |
+------+----------+----------------+----------------+----------------+----------------+
|    5 | 02000000 | 01  53  BE  CD | A2  B0  2A  80 | B6  7E  83  4B | 58  0B  E9  B4 |
|    6 | 02000010 |                |                | 29  06  89  B4 | 4C  AF  28  82 |
|    7 | 03000000 | 83  5D  C4  5C | 9D  A8  CB  2E | E4  AB  B7  03 | F6  7D  9C  0C |
|    8 | 03000010 |                |                | 9F  04  45  4C | 87  B7  E4  84 |
+------+----------+----------------+----------------+----------------+----------------+
```

- This feature can be turned OFF by

```
m_axi_env.cfg.end_of_test_mem_print = 1'b0;
```

END OF TEST MEMORY PRINTING

- Based on user preference, the size per line can be configured:

```
m_axi_env.cfg.mem_print_size = 128; // support 32,64,128
```

- Example below on left is 64b version, right is 32b version

MEMORY TRACKER

NO	ADDR	7	6	5	4	3	2	1	0
1	00000000	49	87	96	6B	72	8D	B6	39
2	00000008	9B	8C	80	06	A8	D5	A6	56
3	00000010	E6	60	44	49	1B	50	47	1E
4	01000000	E1	1C	14	80	50	A9	6E	69
5	01000008	D6	45	19	8A	43	DE	DA	B1
6	01000010	46	63	9E	5B	6E	4E	BF	37
7	02000000	B6	7E	83	4B	58	0B	E9	B4
8	02000008	01	53	BE	CD	A2	B0	2A	80
9	02000010	29	06	89	B4	4C	AF	28	82
10	03000000	E4	AB	B7	03	F6	7D	9C	0C
11	03000008	83	5D	C4	5C	9D	A8	CB	2E
12	03000010	9F	04	45	4C	87	B7	E4	84

MEMORY TRACKER

NO	ADDR	3	2	1	0
1	00000000	72	8D	B6	39
2	00000004	49	87	96	6B
3	00000008	A8	D5	A6	56
4	0000000C	9B	8C	80	06
5	00000010	1B	50	47	1E
6	00000014	E6	60	44	49
7	01000000	50	A9	6E	69
8	01000004	E1	1C	14	80
9	01000008	43	DE	DA	B1
10	0100000C	D6	45	19	8A
11	01000010	6E	4E	BF	37
12	01000014	46	63	9E	5B
13	02000000	58	0B	E9	B4
14	02000004	B6	7E	83	4B
15	02000008	A2	B0	2A	80
16	0200000C	01	53	BE	CD

BACKDOOR MEM DATA RETRIEVAL

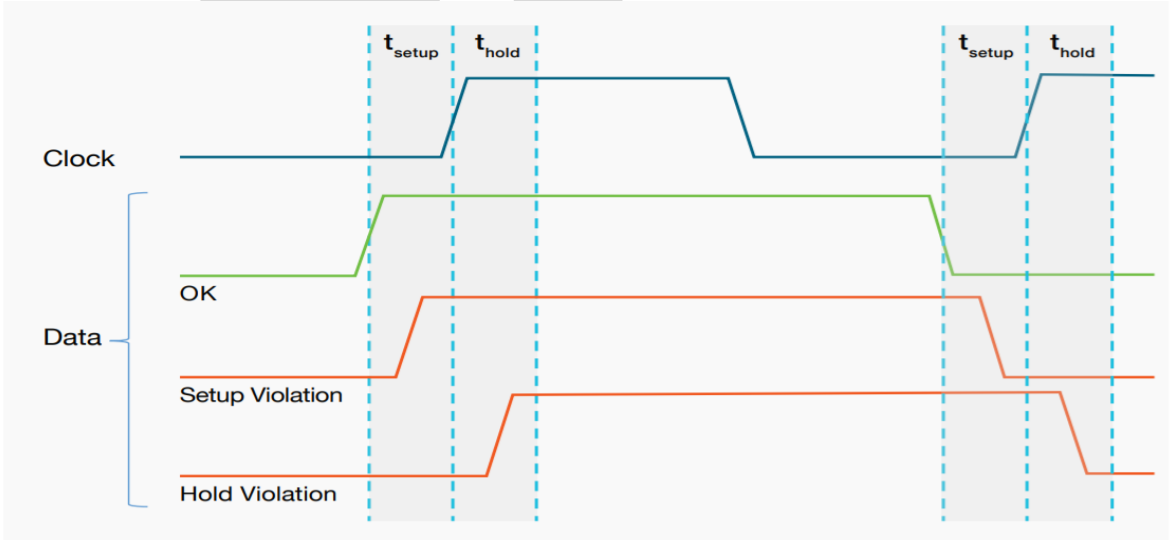
- User can retrieve the current stored data from memory database using the following API as well.

```
bit [31:0] my_spy_data;
my_spy_data = m_axi_env.cfg.mem.get_4_data(32'h1000) ; // data from 1003,1002,1001,1000 addresses
my_spy_data = m_axi_env.cfg.mem.get_2_data(32'h1000) ; // data from           1001,1000 addresses
my_spy_data = m_axi_env.cfg.mem.get_1_data(32'h1000) ; // data from                1000 addresses
```

SETUP AND HOLD TIME

BACKGROUND

- Setup time:
  - signals must be stable for t_{setup} before the rising clock edge sampling.
- Hold time:
  - signals must be stable for t_{hold} after the rising clock edge sampling.

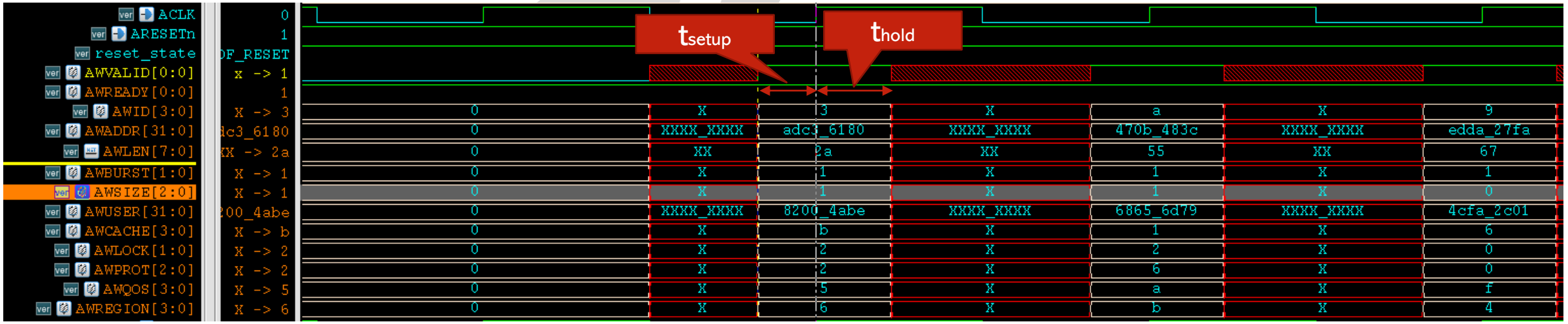


Ref:

https://download.tek.com/document/55W_61095_0_Identifying_Setup-and-Hold_AN_03.pdf

X INJECTION

- LVM AXI UVC has been programmed to be able to verify the setup and hold violations using X injection.
- This is most straight forward method.
- Windows outside setup and hold will be injected with X.
- If X propagates into the DUT, it implies that the DUT side does not meet the *tsetup* and *thold* values.



STEPS TO ENABLE X INJECTION

- Configure the UVC for the setup and hold length:

```
m_axi_env.cfg.setup_hold = LVM_X_INJECTION;
m_axi_env.set_setup_time(3.5 ,"ns");// Recommended to move this to base test
```

- Make sure to wait long enough to allow the UVC to calculate the UVC clock period.

```
`define LVM_AXI_CLK_STABLE 3
```

  - Note: this macro is located at src/lvm_axi_defines.svp. It defines the total clocks the UVC has to wait for before calculating the clock period.
  - This is to make the UVC flexible enough to accommodate those designs where the clock is unstable at the beginning.
- After that, just send packet like usual.

API PART 1: PACKET SENDING

1. READ
2. WRITE
3. WRITE_ADDR
4. WRITE_DATA
5. S_READ
6. S_WRITE
7. rand_write
8. rand_read
9. rand_access
10. rand_1B_access to rand_128B_access
11. unaligned_write
12. unaligned_read

1. READ

- UVC sends an AXI read packet.
- When drv_wait_output == LVM_ON, the read data will be returned as arguments to the API, thus gating it until all the read data are received.
- When drv_wait_output == LVM_OFF, the read data returned is **not** captured as arguments. Read data is invalid.

1. READ

- Arguments accepted:
  - ARID
  - ARADDR
  - ARREGION
  - ARLEN
  - ARSIZE
  - ARBURST
  - ARLOCK
  - ARCACHE
  - ARPROT
  - ARQOS
  - ARUSER
  - ahb_lite

Outputs:
RID
RDATA[]
RRESP []
RUSER []

```
logic   [ 3:0]                  RID     ;
logic   [31:0]                  RDATA [];
logic   [ 1:0]                  RRESP [];
logic   [31:0]                  RUSER [];

    m_axi_env.read(
        .ARID       (4'h8),
        .ARADDR     (32'h8888_8888),
        .ARREGION   (4'h8),
        .ARLEN      (8),
        .ARSIZE     (LVM_AXI_4B),
        .ARBURST    (LVM_AXI_INCR),
        .ARLOCK     ('0),
        .ARCACHE    ('0),
        .ARPROT     ('0),
        .ARQOS      ('0),
        .ARUSER     (32'h8888_8888),
        .ahb_lite   (1'b0),
        .RID        (RID),
        .RDATA      (RDATA),
        .RRESP      (RRESP),
        .RUSER      (RUSER)
    );
```

1. READ

- User can retrieve the data inside the testcase / sequence using the following return variables:

```
foreach(RDATA[i]) begin
    `uvm_info(msg_id, $sformatf("[Index %0d] RID='h%3h RDATA='h%8h RUSER='h%8h RRESP='h%1h",
    i, RID, RDATA[i], RUSER[i], RRESP[i]), UVM_DEBUG)
end
```

- Note: to have valid RDATA[i], RRESP[i] etc, drv_wait_output must be LVM_ON.

2. WRITE

- UVC to send a write packet (AW+W).
- When drv_wait_output == LVM_ON, write response will be returned as arguments to the API, thus gating it until all the write response are received.
- drv_wait_output == LVM_OFF, no wait on response thus it is **not** captured in the arguments.

2. WRITE

- Arguments accepted:
  - ID
  - AWADDR
  - AWREGION
  - AWLEN
  - AWSIZE
  - AWBURST
  - AWLOCK
  - AWCACHE
  - AWPROT
  - AWQOS
  - AWUSER
  - ahb_lite

WDATA
WSTRB
WUSER

Outputs:
BID
BRESP []
BUSER []

Special Control:
order
delay

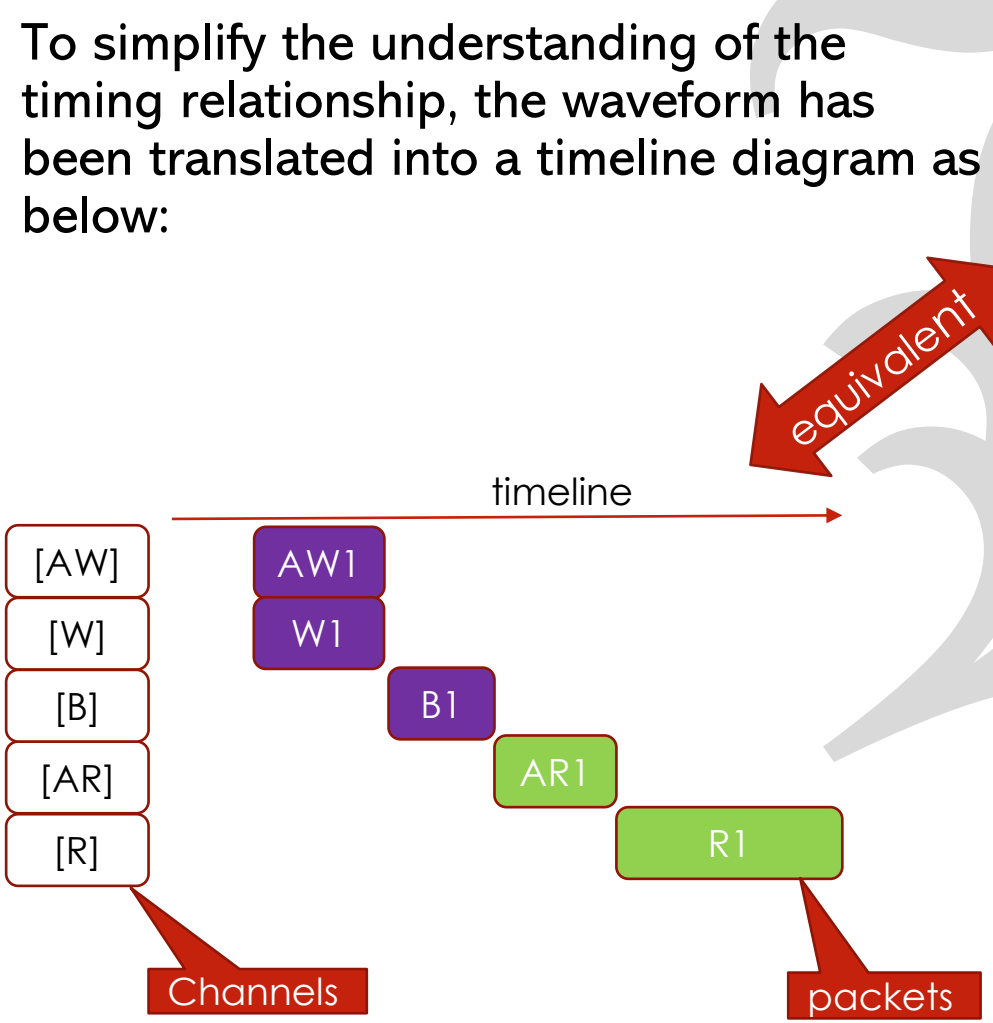
2. WRITE

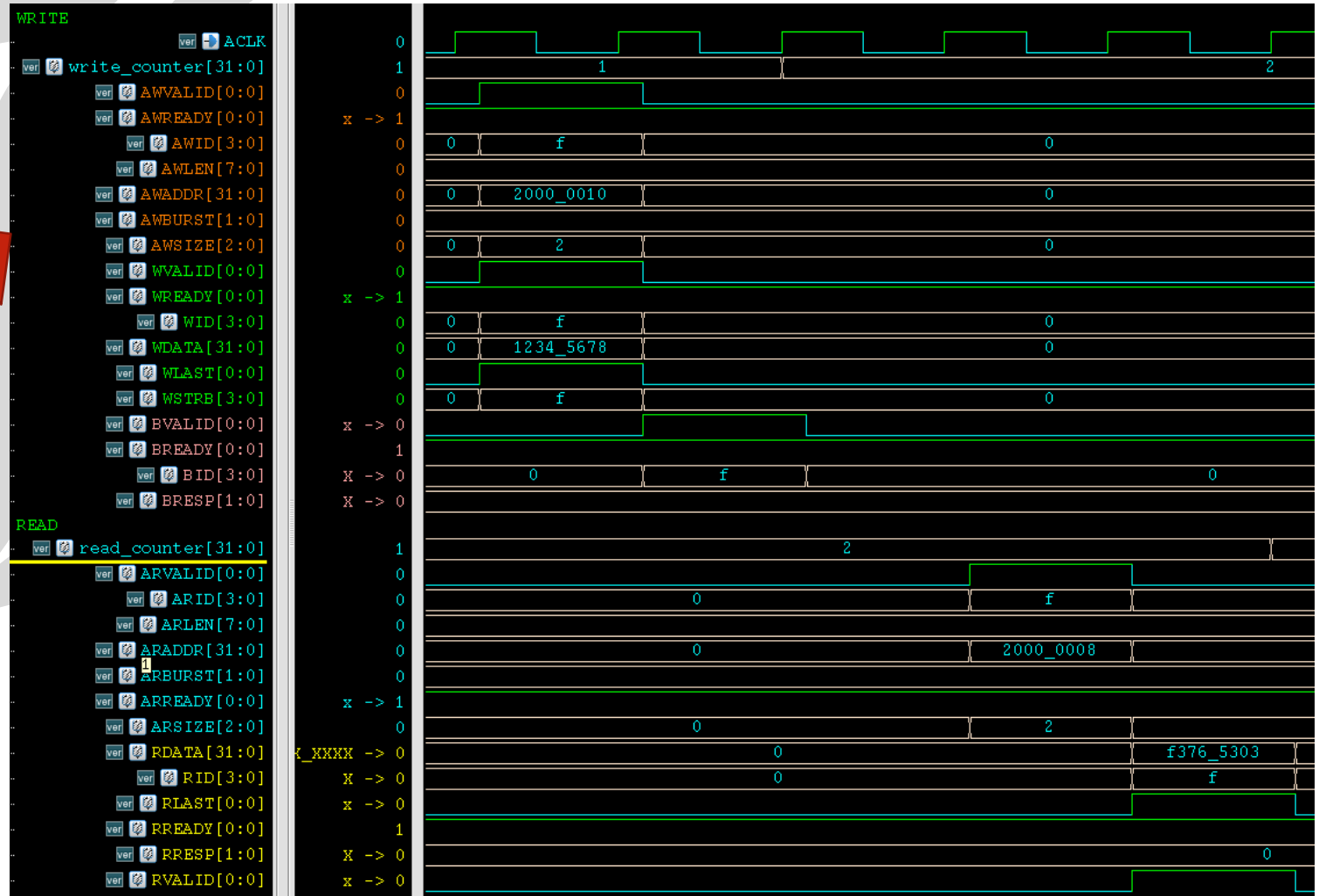

```
logic   [31:0]              WDATA[];
logic   [ 3:0]              WSTRB[];
logic   [31:0]              WUSER[];
logic   [ 3:0]              BID    ;
logic   [ 1:0]              BRESP  ;
logic   [31:0]              BUSER  ;

WDATA  = new[4];
WSTRB  = new[4];
WUSER  = new[4];

WDATA[0] = 32'h1111;   WDATA[1] = 32'h3333;
WDATA[2] = 32'h5555;   WDATA[3] = 32'h7777;
WSTRB[0] = 4'h5;       WSTRB[1] = 4'h6;
WSTRB[2] = 4'h7;       WSTRB[3] = 4'h8;
WUSER[0] = 32'h2200;   WUSER[1] = 32'h4400;
WUSER[2] = 32'h6600;   WUSER[3] = 32'h8800;
```

```
m_axi_env.write(
    .ID         (4'h8),// This is shared (AWID and WID)
    .AWADDR     (32'h2000_0000),
    .AWREGION   (4'hA),
    .AWLEN      (3),
    .AWSIZE     (LVM_AXI_4B),
    .AWBURST    (LVM_AXI_INCR),
    .AWLOCK     ('0), .AWCACHE    ('0),
    .AWPROT     ('0), .AWQOS      ('0),
    .AWUSER     ('1),
    .ahb_lite   ('0), // it is a full axi slave, supports partial wstrb, unaligned addr
    .WDATA      (WDATA),
    .WSTRB      (WSTRB),
    .WUSER      (WUSER),
    .order      (LVM_AXI_AW_THEN_W),// user can control AW finish sending then W
    .delay      (5), // the delay between AW and W will be 5 clocks.

    .BID        (BID),
    .BRESP      (BRESP),
    .BUSER      (BUSER)
);
`uvm_info(msg_id, $sformatf(
    "BID='h%3h  BRESP='h%1h BUSER='h%8h", BID, BRESP, BUSER), UVM_DEBUG)
```

QUICK EXPLANATION IN THE DIAGRAM

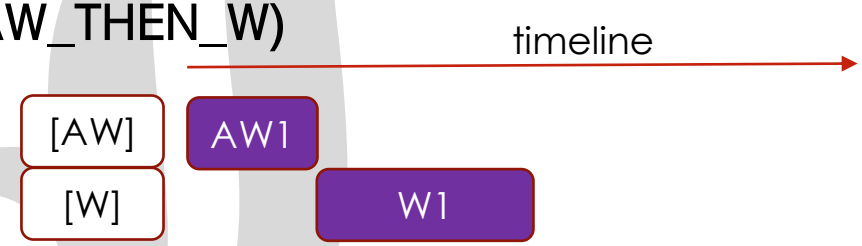
- To simplify the understanding of the timing relationship, the waveform has been translated into a timeline diagram as below:





ORDER EFFECT IN WRITE

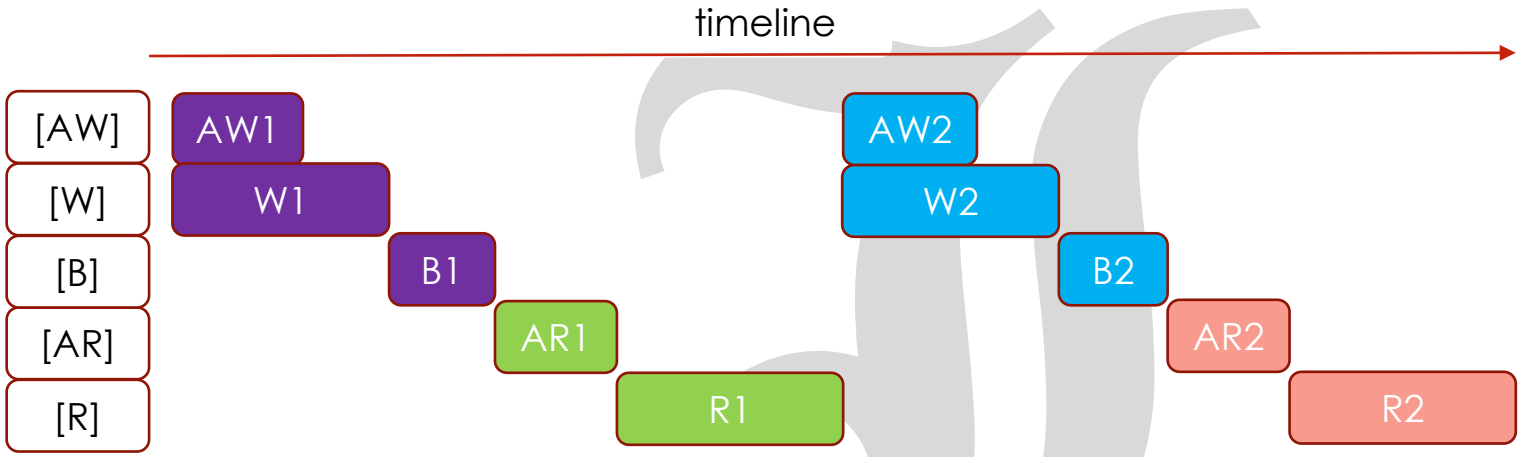
- Order: determine the order of AW and W.
- 3 choices:
  - AW first (order = LVM_AXI_AW_THEN_W)



  - W first (order = LVM_AXI_W_THEN_AW)

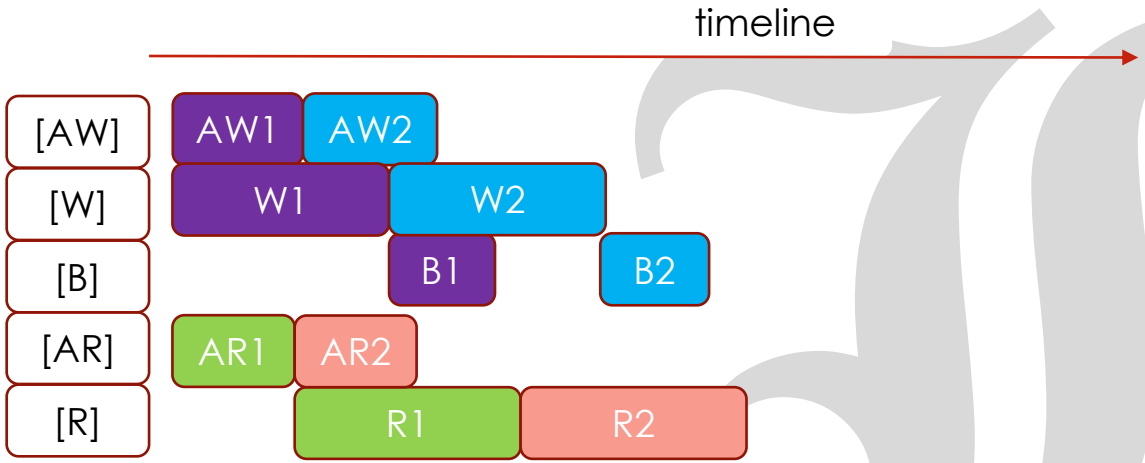
  - Both sent simultaneously (order = LVM_AXI_AW_W_TOGETHER)
    - default setting

QUICK EXAMPLES (ASSUME XREADY ==1)




```
m_axi_env.drv_wait_output(LVM_ON);
m_axi_env.write(...); // AW1+W1, wait for B1
m_axi_env.read(...);  // AR1, wait for R1
m_axi_env.write(...); // AW2+W2, wait for B2
m_axi_env.read(...);  // AR2, wait for R2
```

QUICK EXAMPLES (ASSUME XREADY ==1)




```
// Same effect when the order between read and writes is changed
m_axi_env.write(...); // AW1+W1, B1 is ignored
m_axi_env.write(...); // AW2+W2, B2 is ignored
m_axi_env.read(...); // AR1,R1 is ignored
m_axi_env.read(...); // AR2, R2 is ignored

// OR
m_axi_env.read(...); // AR1,R1 is ignored
m_axi_env.read(...); // AR2, R2 is ignored
m_axi_env.write(...); // AW1+W1, B1 is ignored
m_axi_env.write(...); // AW2+W2, B2 is ignored
```

```
m_axi_env.drv_wait_output(LVM_OFF);
m_axi_env.write(...); // AW1+W1, B1 is ignored
m_axi_env.read(...);  // AR1,    R1 is ignored
m_axi_env.write(...); // AW2+W2, B2 is ignored
m_axi_env.read(...);  // AR2,    R2 is ignored
```

3. WRITE_ADDR

- UVC to send AW.
- W will not be sent.
- driver wait mode does not impact the API behavior.

3. WRITE_ADDR

- Arguments accepted:
  - AWID
  - AWADDR
  - AWREGION
  - AWLEN
  - AWSIZE
  - AWBURST
  - AWLOCK
  - AWCACHE
  - AWPROT
  - AWQOS
  - AWUSER
  - ahb_lite

```
m_axi_env.write_addr(
    .AWID       (4'h9),
    .AWADDR     (32'h3000_0000),
    .AWREGION   (4'hB),
    .AWLEN      (5),
    .AWSIZE     (LVM_AXI_4B),
    .AWBURST    (LVM_AXI_INCR),
    .AWLOCK     ('0),
    .AWCACHE    ('0),
    .AWPROT     ('0),
    .AWQOS      ('0),
    .AWUSER     ('1),
    .ahb_lite   ('0)
);
```

4. WRITE_DATA

- UVC to send W.
- AW will not be sent.
- driver wait mode does not impact the API behavior.

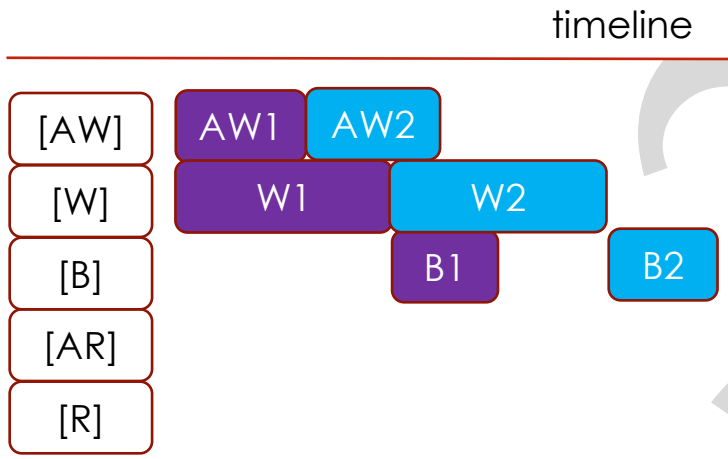
4. WRITE_DATA

Arguments accepted:

- WID
- AWADDR
- AWLEN
- AWSIZE
- AWBURST
- WDATA
- WSTRB
- WUSER
- ahb_lite

```
logic   [31:0]                  WDATA [];
logic   [ 3:0]                  WSTRB [];
logic   [31:0]                  WUSER [];

WDATA = new[3]; WSTRB = new[3]; WUSER = new[3];

WDATA[0] = 32'h8888; WDATA[1] = 32'h4444; WDATA[2] = 32'h6666;

WSTRB[0] = 4'h1;     WSTRB[1] = 4'h2;     WSTRB[2] = 4'h3;

WUSER[0] = 32'h2222; WUSER[1] = 32'h4444; WUSER[2] = 32'h6666;

        m_axi_env.write_data(
            .AWLEN      (2),
            .AWADDR     (32'h3000_0000),
            .AWBURST    (LVM_AXI_INCR),
            .AWSIZE     (LVM_AXI_4B),
            .WID        (4'h9),
            .WDATA      (WDATA),
            .WSTRB      (WSTRB),
            .WUSER      (WUSER)
        );
```

QUICK EXAMPLES (ASSUME XREADY ==1)




```
// Same effect when the order between write_data and
write_addr is changed
m_axi_env.write_data(...); // W1, no wait
m_axi_env.write_data(...); // W2, no wait
m_axi_env.write_addr(...); // AW1, no wait
m_axi_env.write_addr(...); // AW2, no wait

// OR
m_axi_env.write_data(...); // W1, no wait
m_axi_env.write_addr(...); // AW1, no wait
m_axi_env.write_data(...); // W2, no wait
m_axi_env.write_addr(...); // AW2, no wait
```

```
m_axi_env.drv_wait_output(LVM_ON); // or LVM_OFF
m_axi_env.write_addr(...); // AW1, no wait
m_axi_env.write_addr(...); // AW2, no wait
m_axi_env.write_data(...); // W1, no wait
m_axi_env.write_data(...); // W2, no wait
```

5. S_READ

- It will launch bus size read for 1 beat
- Arguments accepted:
  - ARID
  - ARADDR
  - ARREGION
  - ARLEN
  - ARSIZE
  - ARBURST
  - ARLOCK
  - ARCACHE
  - ARPROT
  - ARQOS
  - ARUSER
  - ahb_lite

Outputs:
RID
rdata
rresp
~~RUSER []~~

```
logic   [31      :0] rdata;
logic   [2     -1:0] rresp;

m_axi_env.s_read(
    .ARADDR(32'h2000_0000),
    .rdata (rdata),
    .RID   (RID),
    .rresp (rresp)
);
```

6. S_WRITE

- It will launch bus size write for 1 beat
- Arguments accepted:
  - ID
  - AWADDR
  - AWREGION
  - AWLEN
  - AWSIZE
  - AWBURST
  - AWLOCK
  - AWCACHE
  - AWPROT
  - AWQOS
  - AWUSER
  - ahb_lite

wdata
WSTRB
WUSER

Outputs:
BID
BRESP []
~~BUSER []~~

Special Control:
order
delay

```
logic   [31      :0] rdata;
logic   [2     -1:0] rresp;

m_axi_env.s_write(
    .AWADDR(32'h2000_0000),
    .wdata (32'h1111_2222),
    .BID   (BID),
    .BRESP (BRESP)
);
```

7. RAND_WRITE

- It will launch a random write with lock == 0
- Arguments accepted:
  - m_addr - compulsory
  - m_size - optional, if no input, its value will be randomized
  - m_len - optional, if no input, its value will be randomized
  - ahb_lite - 1'b0 as default
- The order of AW and W will be randomized.
  - If cfg.support_W_first == 1'b0, then only these 2 possibilities:
    - AW -> W
    - AW+W in parallel

```
// Example below shows writing to address 2000_0000, it is a AXI-lite slave eventually
// AXI lite slave means the address must be aligned, with all WSTRB to be continuous
// The API will help to adjust the address internally to make it aligned, if it is not
m_axi_env.rand_write(.m_addr(32'h2000_0000), .ahb_lite(1'b1));
```

8. RAND_READ

- It will launch a random read with lock == 0
- Arguments accepted:
  - m_addr - compulsory
  - m_size - optional, if no input, its value will be randomized
  - m_len - optional, if no input, its value will be randomized
  - ahb_lite - 1'b0 as default

```
// Example below shows reading to address 2000_0000, it is a AXI-lite slave eventually
// AXI lite slave means the address must be aligned, with all WSTRB to be continuous
m_axi_env.rand_read (.m_addr(32'h2000_0000), .ahb_lite(1'b1));
```

9. RAND_ACCESS

- It will launch a random read / write with lock == 0 (if the read_only==1, then only write will be launched)
- Arguments accepted:
  - m_addr - compulsory
  - m_size - optional, if no input, its value will be randomized
  - m_len - optional, if no input, its value will be randomized
  - ahb_lite - 1'b0 as default
  - read_only - 1'b0 as default

```
// Example below shows writing/reading to address 2000_0000, it is a AXI-lite slave eventually
// AXI lite slave means the address must be aligned, with all WSTRB to be continuous
m_axi_env.rand_access (.m_addr(32'h2000_0000), .ahb_lite(1'b1));
```

10. RAND_1B_ACCESS TO RAND_8B_ACCESS

- It will launch a random read / write with lock == 0 (if the read_only==1, then only write will be launched), with the size is fixed
- Arguments accepted:
  - m_addr - compulsory
  - m_len - optional, if no input, its value will be randomized
  - ahb_lite - 1'b0 as default
  - read_only - 1'b0 as default

```
m_axi_env.rand_1B_access (.m_addr(32'h2000_0000), .ahb_lite(1'b1));
m_axi_env.rand_2B_access (.m_addr(32'h2000_0000), .ahb_lite(1'b1));
m_axi_env.rand_4B_access (.m_addr(32'h2000_0000), .ahb_lite(1'b1));
m_axi_env.rand_8B_access (.m_addr(32'h2000_0000), .ahb_lite(1'b1));
```

11. UNALIGNED_WRITE

- It will launch a write with lock == 0 with unaligned address
- Arguments accepted:
  - m_addr - compulsory
  - m_len - optional, if no input, its value will be randomized
  - m_size - optional, if no input, its value will be randomized
  - ahb_lite - 1'b0 as default

```
// if ahb_lite is 0, means can accept unaligned address
// if the addr provided is aligned, then it will modify the last few bits to make it
// unaligned
m_axi_env.unaligned_write(.m_addr(32'h2000_0000), .ahb_lite(1'b0));
```

12. UNALIGNED_READ

- It will launch a read with lock == 0 with unaligned address
- Arguments accepted:
  - m_addr - compulsory
  - m_len - optional, if no input, its value will be randomized
  - m_size - optional, if no input, its value will be randomized
  - ahb_lite - 1'b0 as default

```
// if ahb_lite is 0, means can accept unaligned address
// if the addr provided is aligned, then it will modify the last few bits to make it
// unaligned
m_axi_env.unaligned_read (.m_addr(32'h2000_0000), .ahb_lite(1'b0));
```

SUMMARY: IMPACT OF DRV_WAIT_OUTPUT

API name	drv_wait_output=LVM_ON	drv_wait_output=LVM_OFF
read/s_read	[WAIT] The read (AR) will be sent and gated until the read data is returned, only then next API is executed. Output arguments like RDATA, RRESP etc can be used after the API.	[NO WAIT] The read (AR) will be sent. Without waiting for RDATA, RRESP etc, next line after the API will get executed.
write/s_write	[WAIT] The write (AW+W) will be sent and gated until the write response is returned, only then next API is executed. Output arguments like BRESP, BID etc can be used after the API.	[NO WAIT] The write (AW+W) will be sent. Without waiting for BRESP etc, next line after the API will get executed.
write_addr	[NO WAIT] The write address (AW) will be sent and no gating.	[NO WAIT] Same as left.
write_data	[NO WAIT] The write data (W) will be sent and no gating.	[NO WAIT] Same as left.

QUICK NOTE: NO NEED TO MENTION ALL ARGUMENTS

- User can choose to fix some variables in the API while leaving others to be randomized. For example:

```
m_axi_env.read(
    .ARID          (id),
    .ARADDR        (32'hFFFF_0000),
    .ARLOCK        (lock[1:0]),
    // output cannot be skipped
    .RID           (RID),
    .RDATA         (RDATA),
    .RRESP         (RRESP),
    .RUSER         (RUSER)
);
```

Other variables like ARLEN, ARUSER etc are fully randomized based on protocol constraint.

QUICK NOTE: NO NEED TO MENTION ALL ARGUMENTS

- Similar for write API:

```
m_axi_env.write(
    .ID          (id),
    .AWADDR      (32'hEEEE_0000),
    .AWLOCK      (lock[1:0]),
    .AWLEN       (len),
    // output cannot be skipped
    .BID         (BID),
    .BRESP       (BRESP),
    .BUSER       (BUSER)
);
```

Other variables like AWLEN, AWUSER etc are fully randomized based on protocol constraint.

API PART 2: PACKET WAITING

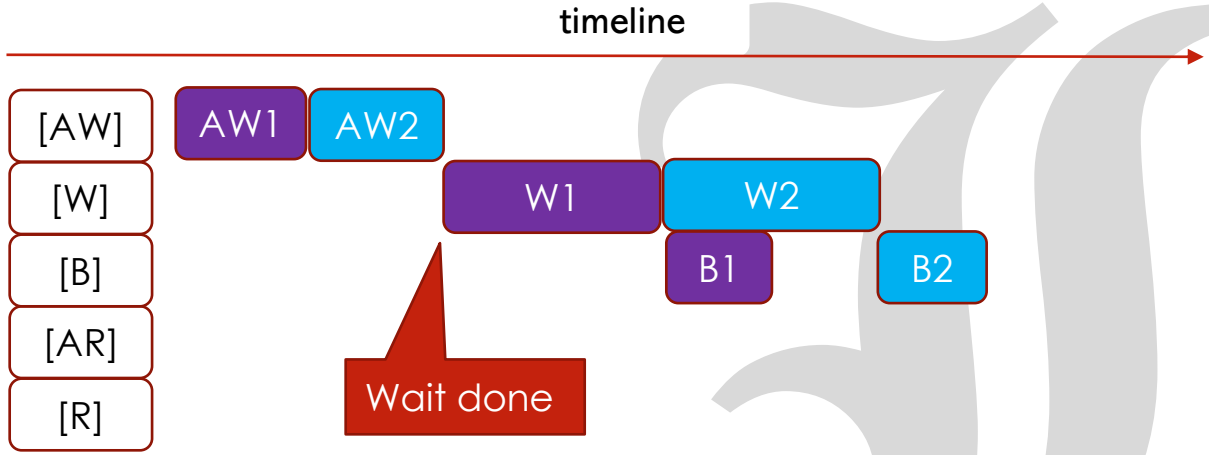
1. WAIT_AW_SENT
2. WAIT_W_SENT
3. WAIT_AR_SENT
4. WAIT_B
5. wait_write_done
6. wait_read_done
7. wait_all_done

1. WAIT_AW_SENT

- This API waits for all the 'AW' to be send out by driver.
- It can be used when drv_wait_output is LVM_OFF but user wishes to gate a point by waiting for all the AW to finish sending.
- This API does not care about sending status of 'W'.
- Note: in drv_wait_output = LVM_ON mode, the wait will be done automatically in write API, but not in the write_addr API.

```
m_axi_env.wait_AW_sent;
```

QUICK EXAMPLES (ASSUME XREADY ==1)



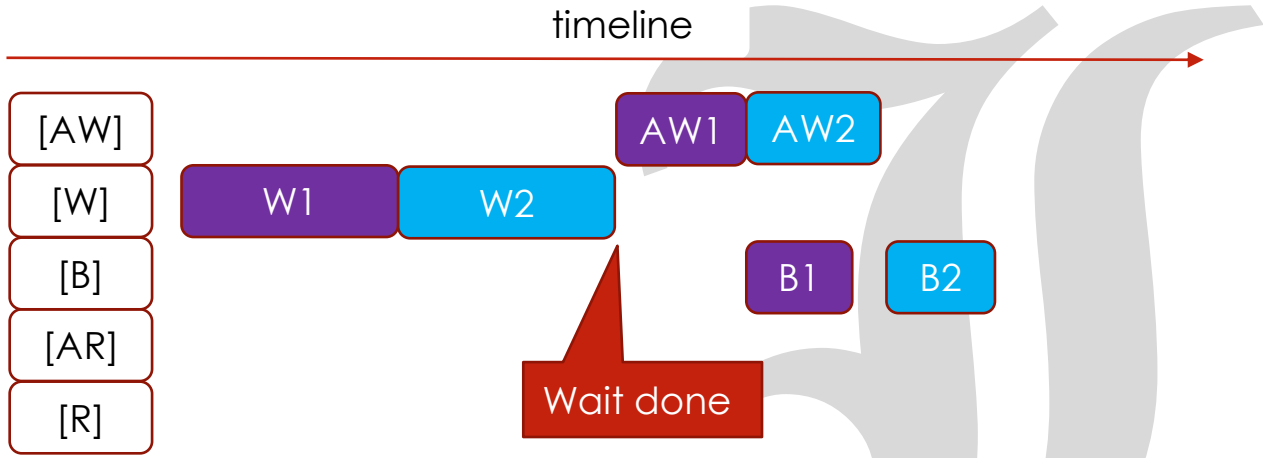

```
m_axi_env.drv_wait_output(LVM_ON); // does not matter
m_axi_env.write_addr(...); // AW1, no wait
m_axi_env.write_addr(...); // AW2, no wait
m_axi_env.wait_AW_sent;    // Wait all AW above to finish
m_axi_env.write_data(...); // W1, no wait
m_axi_env.write_data(...); // W2, no wait
```

2. WAIT_W_SENT

- This API waits for all the 'W' to be send out by driver.
- It can be used when drv_wait_output is LVM_OFF but user wishes to gate a point by waiting to finish sending of all the 'W'.
- Note: in drv_wait_output = LVM_ON mode, the wait will be done automatically in write API, but not in the write_data API.

```
m_axi_env.wait_W_sent;
```

QUICK EXAMPLES (ASSUME XREADY ==1)




```
m_axi_env.drv_wait_output(LVM_ON); // does not matter
m_axi_env.write_data(...); // W1, no wait
m_axi_env.write_data(...); // W2, no wait
m_axi_env.wait_W_sent;     // Wait all W above to finish
m_axi_env.write_addr(...); // AW1, no wait
m_axi_env.write_addr(...); // AW2, no wait
```

3. WAIT_B

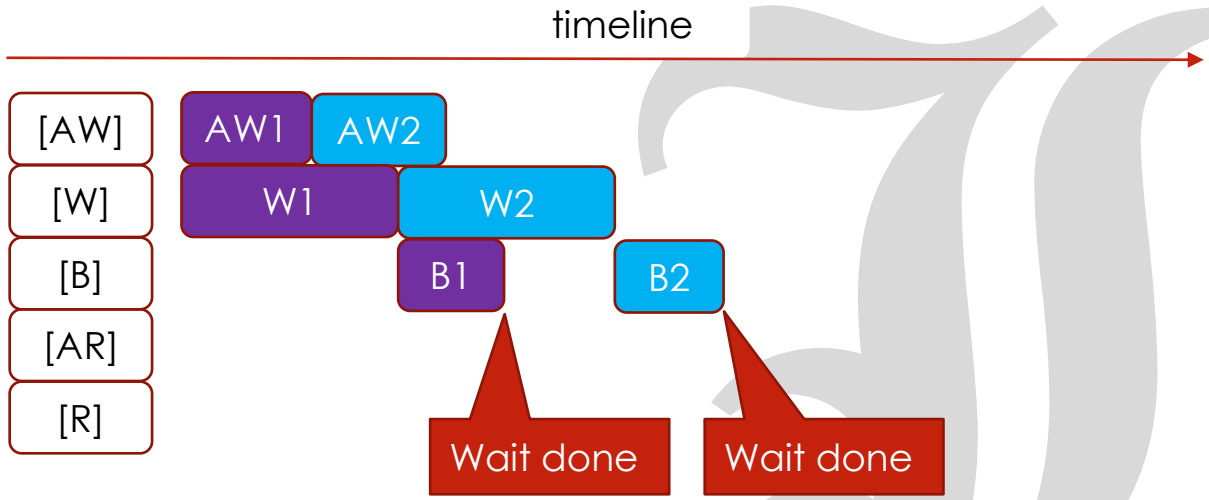
- This API waits for the next 'B' (just one B) to be returned from slave.
- It can be used when drv_wait_output is LVM_OFF but user wishes to gate a point by waiting for the next write response to come back.
- Note: in drv_wait_output = LVM_ON mode, the wait will be done automatically in write API.

3. WAIT_B

- Arguments accepted:
  - BID
  - BRESP
  - BUSER

```
logic   [ 3:0]                  BID     ;
logic   [ 1:0]                  BRESP   ;
logic   [31:0]                  BUSER   ;

...

 m_axi_env.wait_B(BID, BRESP, BUSER);
`uvm_info(msg_id, $sformatf("Captured BID=%0h BRESP=%0h BUSER=%0h", BID, BRESP, BUSER),
UVM_DEBUG)
```

QUICK EXAMPLES (ASSUME XREADY ==1)



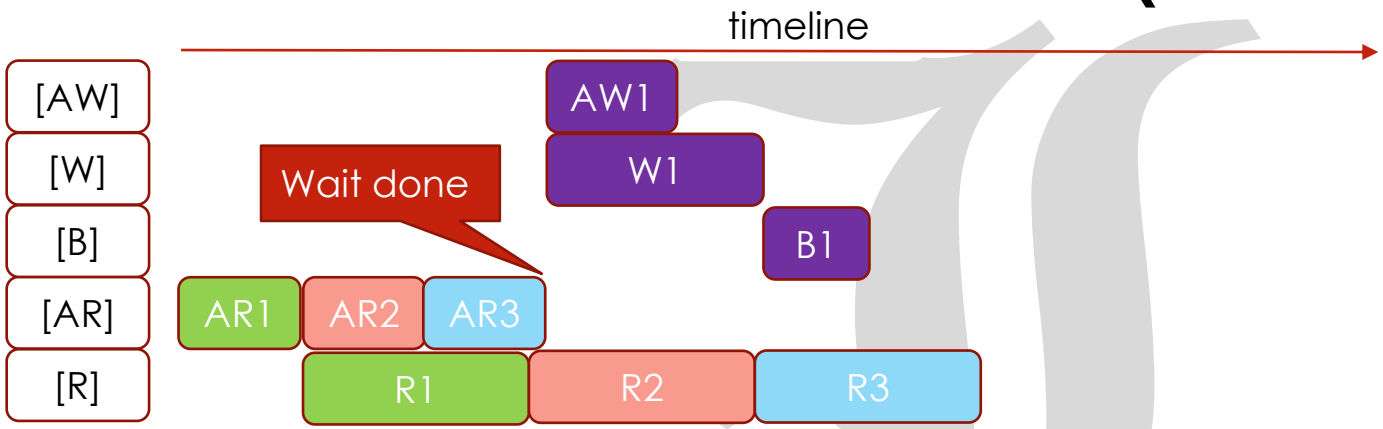

```
m_axi_env.drv_wait_output(LVM_ON); // does not matter
m_axi_env.write_data(...); // W1, no wait
m_axi_env.write_data(...); // W2, no wait
m_axi_env.write_addr(...); // AW1, no wait
m_axi_env.write_addr(...); // AW2, no wait
m_axi_env.wait_B(...);     // Wait to fetch 1 B (B1)
m_axi_env.wait_B(...);     // Wait to fetch 1 B (B2)
```

4. WAIT_AR_SENT

- This API waits for all the 'AR' to be send out by driver.
- It can be used when drv_wait_output is LVM_OFF but user wishes to gate a point by waiting to finish sending of all the 'AR'.
- Note: in drv_wait_output = LVM_ON mode, the wait will be done automatically in read API.

```
m_axi_env.wait_AR_sent;
```

QUICK EXAMPLES (ASSUME XREADY ==1)



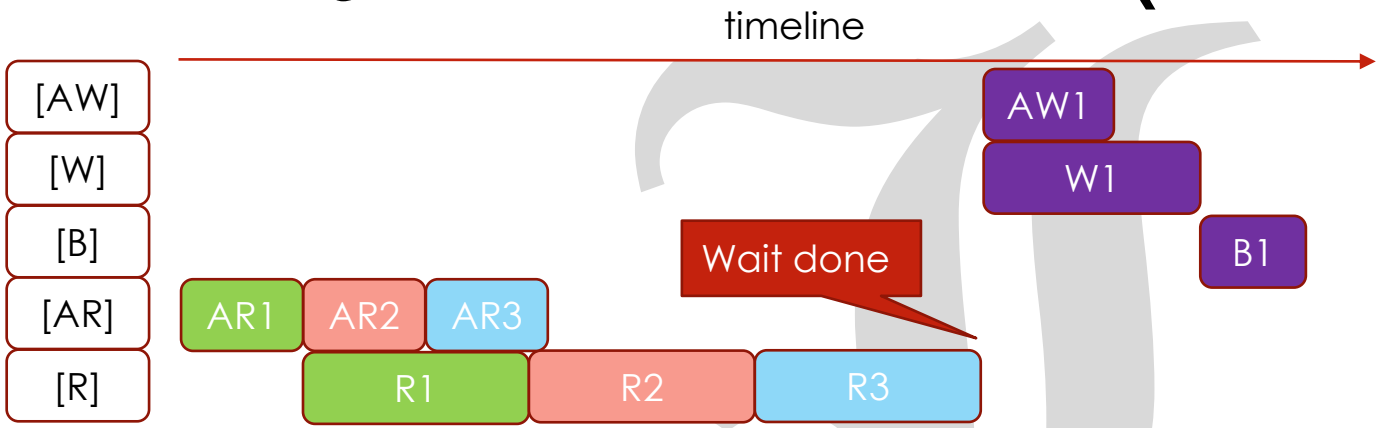

```
m_axi_env.drv_wait_output(LVM_OFF);
m_axi_env.read(...);  // AR1
m_axi_env.read(...);  // AR2
m_axi_env.read(...);  // AR3
m_axi_env.wait_AR_sent(...);// wait for all AR sent done
m_axi_env.write(...);       // AW1+W1
```

5. WAIT_READ_DONE

- This API waits for all the 'AR' to be send out by driver and for all 'R' to return.
- It can be used when drv_wait_output is LVM_OFF but user wishes to gate a point by waiting for all the 'AR' to be send and all the 'R' to return.
- Note: in drv_wait_output = LVM_ON mode, the wait will be done automatically in read API.

```
m_axi_env.wait_read_done;
```

QUICK EXAMPLES (ASSUME XREADY ==1)



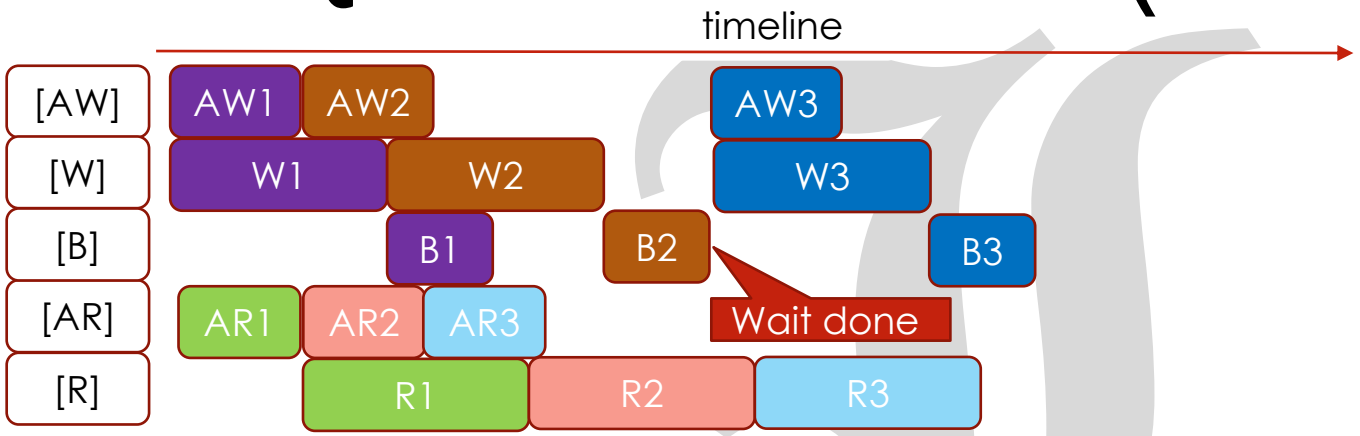

```
m_axi_env.drv_wait_output(LVM_OFF);
m_axi_env.read(...); // AR1
m_axi_env.read(...); // AR2
m_axi_env.read(...); // AR3
m_axi_env.wait_read_done(...); // wait for all AR sent done and all R return
m_axi_env.write(...); // AW1+W1
```

6. WAIT_WRITE_DONE

- This API waits for all the 'AW+W' to be send out by driver and for all the 'B' to return.
- It can be used when drv_wait_output is LVM_OFF but user wishes to gate a point by waiting for all the 'AW+W' to be send and all 'B' to return.
- Note: in drv_wait_output = LVM_ON mode, the wait will be done automatically in write API.

```
m_axi_env.wait_write_done;
```

QUICK EXAMPLES (ASSUME XREADY ==1)




```
m_axi_env.drv_wait_output(LVM_OFF);
m_axi_env.write(...); // AW1+W1
m_axi_env.write(...); // AW2+W2
m_axi_env.read(...);  // AR1
m_axi_env.read(...);  // AR2
m_axi_env.read(...);  // AR3
m_axi_env.wait_write_done(...);
m_axi_env.write(...); // AW3+W3
```

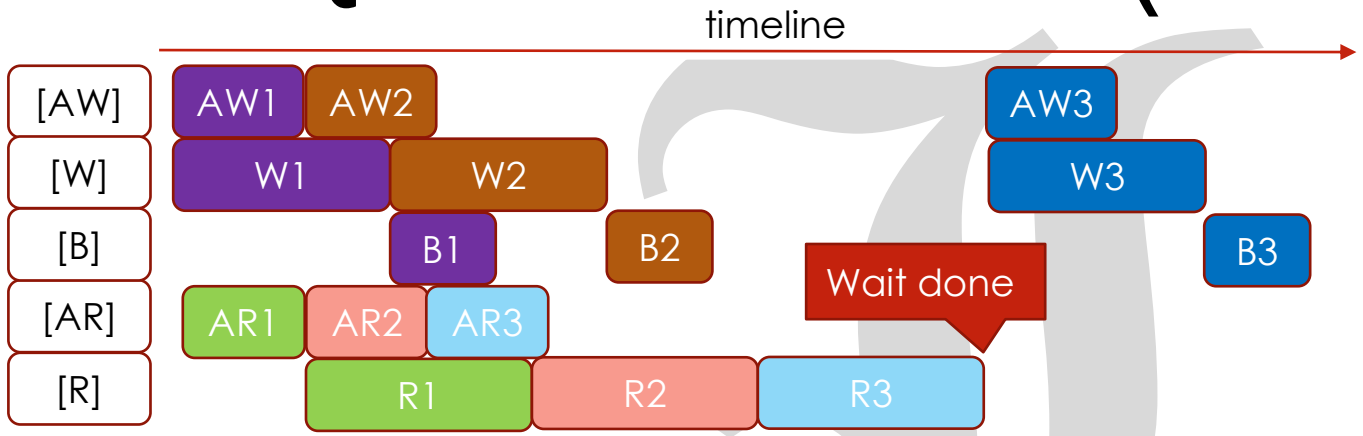
7. WAIT_ALL_DONE

- This API waits for all the AW+W+AR to be send out by driver and all 'B', 'R' to return.

```
m_axi_env.wait_all_done;
```

- It can be used when drv_wait_output is LVM_OFF but user wishes to gate a point by waiting for all the AW+W+AR to finish sending and all the B,R to return.
  - This is useful when user wants to switch mode to drv_wait_output == LVM_ON
- Note: in drv_wait_output = LVM_ON mode, the wait will be done automatically in write and read API.

QUICK EXAMPLES (ASSUME XREADY ==1)




```
m_axi_env.drv_wait_output(LVM_OFF);
m_axi_env.write(...); // AW1+W1
m_axi_env.write(...); // AW2+W2
m_axi_env.read(...);  // AR1
m_axi_env.read(...);  // AR2
m_axi_env.read(...);  // AR3
m_axi_env.wait_all_done(...);
m_axi_env.write(...); // AW3+W3
```

QUICK NOTE: END OF TEST CHECK

- AXI driver will wait for all the packets transmission to be complete.
- It gates the post_main_phase for this.
- This is to ensure all reads / writes are completed by slave.
- It can be turned OFF using the following command:

```
m_axi_env.cfg.end_of_test_check_en = 1'b0;
```

- Sample scoreboard (Ref: tb/example_axi_user_sbd.sv) also has embedded checker where it ensures total packets must not be 0.

API PART 3: DIRECT DRIVING

DIRECT DRIVE: RREADY

- This API directly controls RREADY from AXI master UVC:
  - <env>.drive_RREADY(<value>);
- For example, user can delay the RREADY from the UVC and confirm DUT will hold the R.

```
m_axi_env.drv_wait_output(LVM_ON);
fork begin
    m_axi_env.read(
        .ARID       (4'h9),
        .ARADDR     (32'h9000_0000),
        .ARREGION   (4'hC),
        .ARLEN      (5),
        .RID        (RID),
        .RDATA      (RDATA),
        .RRESP      (RRESP),
        .RUSER      (RUSER)
    );
end
begin
    m_axi_env.drive_RREADY(1'b0);
    repeat($urandom_range(50000,250000)) #1ps;
    m_axi_env.drive_RREADY(1'b1);
end
join
```

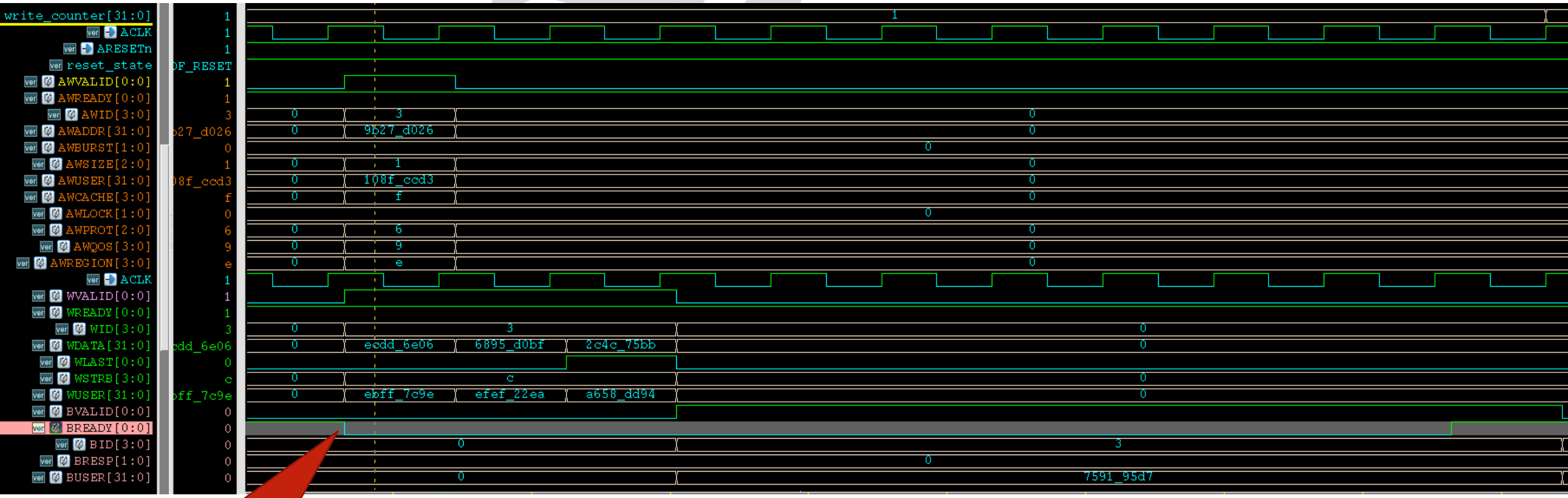
IMPACT OF RREADY == 0

DIRECT DRIVE: BREADY

- This API directly controls BREADY from AXI master UVC:
  - <env>.drive_BREADY(<value>);
- For example, user can delay the BREADY from the UVC and confirm DUT will hold the B.

```
m_axi_env.drv_wait_output(LVM_ON);
fork begin
    m_axi_env.write(
        .AWLOCK     (lock[1:0]),
        .AWLEN      (2),
        .BID        (BID),
        .BRESP      (BRESP),
        .BUSER      (BUSER)
    );
end
begin
    m_axi_env.posedge_clk(1);   // wait for some UVC clocks
    m_axi_env.drive_BREADY(1'b0);
    m_axi_env.posedge_clk(10);  // wait for some UVC clocks
    m_axi_env.drive_BREADY(1'b1);
    m_axi_env.posedge_clk(2);   // wait for some UVC clocks
end
join
```

IMPACT OF BREADY == 0

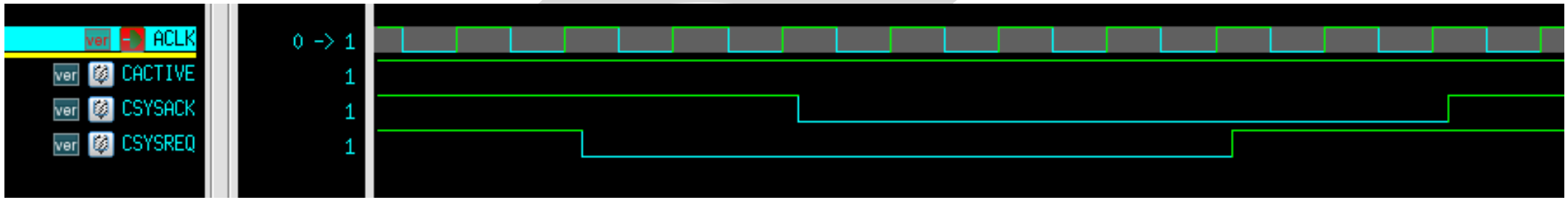


BREADY == 0

DIRECT DRIVE: CSYSREQ + WAIT FOR ACK

- This API directly controls CSYSREQ from AXI master UVC:
  - <env>.drive_CSYSREQ(<value>);
- For example, user can initiate low power request from the UVC and confirm DUT will return ACK.
- This API waits for the CSYSACK
  - <env>.drive_CSYSACK(<value>);

```
repeat($urandom_range(0    ,50000)) #1ps;
m_axi_env.drive_CSYSREQ(1'b0);
m_axi_env.wait_CSYSACK (1'b0);
repeat($urandom_range(10000,50000)) #1ps;
m_axi_env.drive_CSYSREQ(1'b1);
m_axi_env.wait_CSYSACK (1'b1);
```



API PART 4: PRINTING

1. print_AR_R
2. print_AW_W_B

PRINT_AR_R

- This API at seq item allows user to clearly print out all the AR and R elements.
- Example usage:

```
`uvm_info(msg_id, $sformatf("Captured Read  transaction\n%s", captured_item.print_AR_R)    , UVM_MEDIUM)
```

- User will get this at log file:

```
UVM_INFO @ 50.000 ns: uvm_test_top.my_sbd [example_axi_user_sbd] Captured Read  transaction
      ---------------------------------------------------------------------------------------------------------------------------------------------------------------------
      ARID=4'hf  ARADDR=32'h20000004  ARREGION=4'hf  ARLEN=8'h00  ARSIZE=LVM_AXI_4B  ARBURST=LVM_AXI_FIXED  ARLOCK=LVM_AXI_NORMAL     ARCACHE=4'h0  ARPROT=3'h0  ARQOS=4'h0  ARUSER=32'hffffffff

                         RID   ADDR          EFFECTIVE     RDATA         RUSER         RRESP  SOURCE
      [Beat   0    31: 0]  4'hf  32'h20000004  32'h13218b22  32'h13218b22  32'h40e17235  2'h0   20000007->13 20000006->21 20000005->8b 20000004->22
      ---------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

- Ref: <lvm_axi>/tb/example_axi_user_sbd.sv

PRINT_AW_W_B

- This API at seq item allows user to clearly print out all the AW, W and B elements
- Example usage:

```
`uvm_info(msg_id, $sformatf("Captured Write transaction\n%s", captured_item.print_AW_W_B) , UVM_MEDIUM)
```

- User will get this:

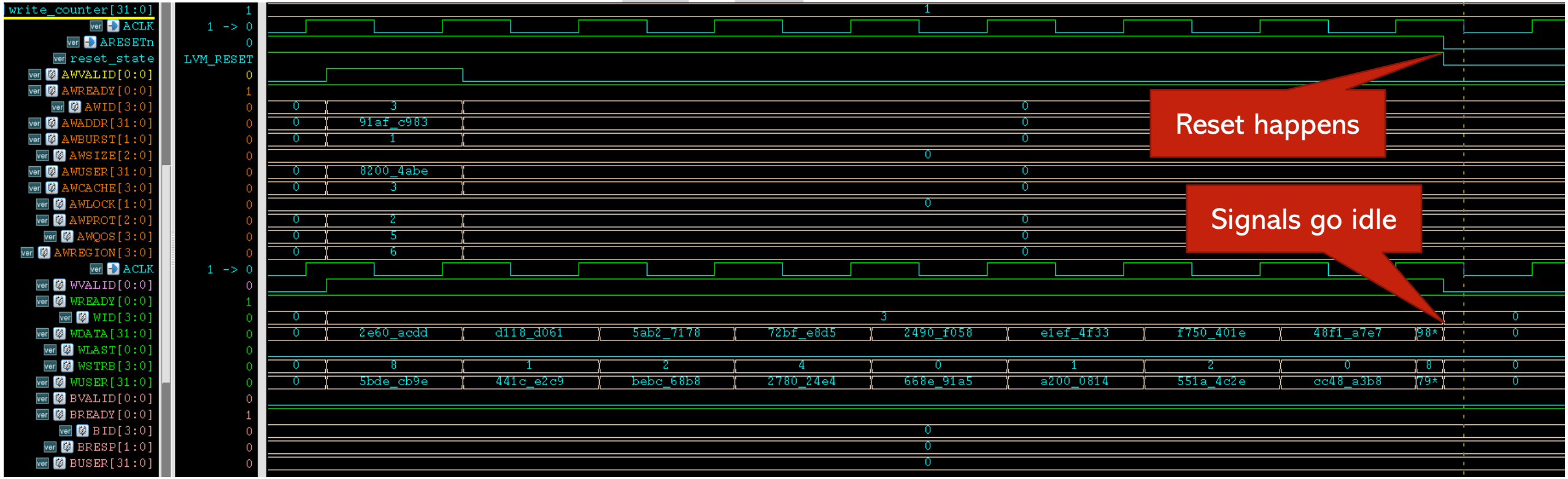
```
UVM_INFO @ 130.000 ns: uvm_test_top.my_sbd [example_axi_user_sbd] Captured Write transaction
      ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
      AWID=4'hf  AWADDR=32'h20000008  AWREGION=4'hf  AWLEN=8'h00  AWSIZE=LVM_AXI_4B  AWBURST=LVM_AXI_FIXED  AWLOCK=LVM_AXI_NORMAL     AWCACHE=4'h0  AWPROT=3'h0  AWQOS=4'h0  AWUSER=32'hffffffff

                         ADDR          EFFECTIVE     WDATA         WSTRB  WUSER          IMPACT
      [Beat   0   31: 0]  32'h20000008  32'h12345678  32'h12345678  4'hf   32'hffffffff   2000000b<-12 2000000a<-34 20000009<-56 20000008<-78

      BID=4'hf  BRESP=2'h0  BUSER=32'hf1620bad
      ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

- Ref: <lvm_axi>/tb/example_axi_user_sbd.sv

RESET AWARE UVC

RESET AWARE UVC

- This UVC is reset aware.
- When ARESETn goes active, all the components will go to passive mode automatically.
- During this time, driver will not response to incoming sequence item.



SEQUENCE ITEM & SEQUENCE WRITING

SEQ ITEM VARIABLE NAMES

- LVM AXI sequence item consists of standard AXI bus signals.
- Red colour font means they are array.

AW
AWID
AWADDR
AWREGION
AWLEN
AWSIZE
AWBURST
AWLOCK
AWCACHE
AWPROT
AWQOS
AWVALID
AWUSER
AWREADY

W
WID
WDATA
WSTRB
WLAST
WVALID
WUSER
WREADY

B
BID
BRESP
BVALID
BUSER
BREADY

AR
ARID
ARADDR
ARREGION
ARLEN
ARSIZE
ARBURST
ARLOCK
ARCACHE
ARPROT
ARQOS
ARVALID
ARUSER
ARREADY

R
RID
RDATA
RRESP
RLAST
RVALID
RUSER
RREADY

Power
CSYSREQ
CSYSACK
CACTIVE

SEQ ITEM VARIABLE NAMES

- Besides, there are some variables that can ease the verification work of the user.
- Red colour font means they are array.

DEBUG

op

ADDR

msb

lsb

effective_wdata

effective_rdata

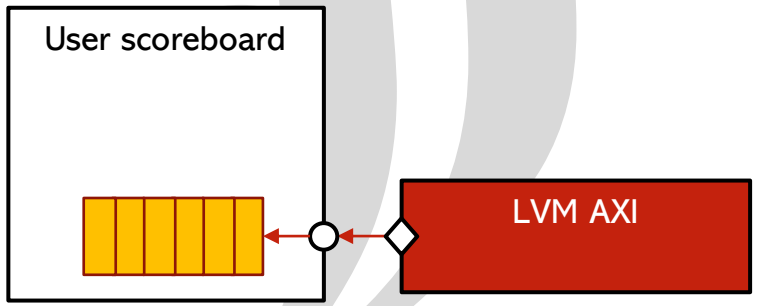
mem_impact

No	Variable name	type	Purpose
1	op	lvm_axi_op_em	To tell the type of the item, for example, it is a READ / WRITE / WRITE_ADDR / WRITE_DATA
2	ADDR	logic (array)	Stores effective addresses for each beat (marked as “1” below)
3	msb	logic (array)	Stores effective data MSB for each beat (marked as “2” below)
4	lsb	logic (array)	Stores effective data LSB for each beat (marked as “3” below)
5	effective_wdata	logic (array)	Stores the effective write data for each beat (marked as “4” below)
6	effective_rdata	logic (array)	Stores the effective read data for each beat
7	mem_impact	array of addr,data	Stores the effective write/read data for each beat, like “impact” below (marked as “5” below)

```
[Packet 4  : WRITE 3  ]       510.000 ns
      ------------------------------------------------------------------------------------------------------------------------
      AWID=4'h2  AWADDR=32'h7eae34af  AWLEN=8'h05  AWSIZE=LVM_AXI_2B  AWBURST=LVM_AXI_INCR   AWLOCK=LVM_AXI_LOCKED     AWCACHE=4'h3  AWPROT=3'h0

                        WID   ADDR          EFFECTIVE     WDATA         WSTRB  IMPACT
      [Beat   0   31:24]  4'h2  32'h7eae34af  16'h88        32'h8879a92d  4'h8   7eae34af<-88
      [Beat   1   15: 0]  4'h2  32'h7eae34b0  16'h3d3e      32'hd2d03d3e  4'h3   7eae34b1<-3d 7eae34b0<-3e
      [Beat   2   31:16]  4'h2  32'h7eae34b2  16'h5ce5      32'h5ce5b610  4'hc   7eae34b3<-5c 7eae34b2<-e5
      [Beat   3   15: 0]  4'h2  32'h7eae34b4  16'hc8f9      32'h4cd5c8f9  4'h3   7eae34b5<-c8 7eae34b4<-f9
      [Beat   4   31:16]  4'h2  32'h7eae34b6  16'h58bf      32'h58bfbb84  4'hc   7eae34b7<-58 7eae34b6<-bf
      [Beat   5   15: 0]  4'h2  32'h7eae34b8  16'h2872      32'he6d32872  4'h3   7eae34b9<-28 7eae34b8<-72

      BID=4'h2  BRESP=2'h0
      ------------------------------------------------------------------------------------------------------------------------
```

RETRIEVING SEQ ITEM FROM ENV

- As shown below, this UVC provides a port at its env.
- User can retrieve the seq item as shown in tb/example_axi_user_sbd.sv



USER DEFINED UVM SEQUENCE

- Unlike conventional UVCs , users need not code the UVM sequence manually when using LVM AXI UVC.
- However, if the existing API cannot support what user wants, the user might need to code the sequence.
- To best illustrate this, an example code has been added into the VIP package:
  - Pls refer to tb/example_axi_user_seq.sv
- Please take note that the debug signals are not meant to be used in sequence writing.
- Recommendation: use API instead.
- If user has any situation that cannot be supported by current API, please email LVM

AXI4 TESTSUITES

AXI4 DEMO TESTCASES AND TESTSUITES

- In this VIP, the self-test testbench and testcases are ready for the user to try on.
- The aim for demo testcases is to:
  - get to know the UVC.
  - know how to utilize the APIs.
  - try to simulate and observe the waveforms.
  - try to modify the testcases and start applying the API to create new stimulus.
- The AXI testsuite is designed to
  - verify the DUT as slave
- User can easily add new testcases by adding into the <lvm_axi>/tests/testlist.sv and adding the runcmd into <lvm_axi>/sim/makefile

DEMO TESTCASES

Testname	Purpose
lvm_axi_01_ral_access_test	RAL access examples with drv_wait_output = LVM_ON and LVM_OFF.
lvm_axi_02_ral_access_cfg_test	RAL access examples with runtime configurable AW, W and AR signal values like: AxID, AxREGION, AxPROT etc. Also demonstrate predictor working for various packets.
lvm_axi_03_basic_rw_test	Demonstrates simple write and read API usage: User can provide all arguments or minimum number of arguments.
lvm_axi_04_basic_b2b_rw_test	Demonstrates how pipelined transactions can be achieved. (AW's+W's+AR's)
lvm_axi_05_exclusive_rw_test	Example on exclusive read write.
lvm_axi_05a_locked_rw_test	Example on locked read write.
lvm_axi_06_mixed_mode_drv_test	Switching driver mode on the fly and demonstrates the usage of wait_write_done API.
lvm_axi_07_w_then_aw_test	Pipelined W's then pipelined AW's, 2 variants: B come in order and out of order
lvm_axi_08_aw_then_w_test	Pipelined AW's then pipelined W's, 2 variants: B come in order and out of order

DEMO TESTCASES

Testname	Purpose
lvm_axi_09_ar_then_r_test	Send b2b AR, then all the R come back after that. Demonstrate the usage example of wait_AR_sent and wait_read_done, 2 variants: R come in order and out of order
lvm_axi_09a_aw_w_then_b_test	Holding up BREADY, while sending both AW and W
lvm_axi_10_direct_drive_test	Direct driving of RREADY, BREADY and CSYSREQ examples.
lvm_axi_11_write_orders_test	Demonstrate different AW and W order
lvm_axi_12_more_writes_test	More write examples.
lvm_axi_13_more_reads_test	More read examples.
lvm_axi_15_single_access_test	Single access test using s_write and s_read API
lvm_axi_17_stalled_read_test	Read is stalled while write is on going
lvm_axi_18_stalled_write_test	Write is stalled while read is on going

DEMO TESTCASES

Testname	Purpose
lvm_axi_20_user_seqs_test	Example of test that starts the user defined sequence.
lvm_axi_21_b2b_unaligned_test	Run b2b random unaligned transfers
lvm_axi_30_no_component_test	Demonstrates how to turn OFF a component completely, they are prd, adp and also mon, UVC will not instantiate the component.
lvm_axi_31_mid_reset_test	Demonstrates the robustness of UVC where it is reset aware. Even get reset in the middle of transactions, after out of reset, it can get back to being operational.
lvm_axi_32_monitor_off_test	Demonstrates the robustness of UVC where monitor can be OFF in the middle of transactions and then recovers after getting turned ON.
lvm_axi_33_predictor_off_test	Demonstrates the effect of RAL predictor, where user can observe the impact of turning off RAL predictor in the env.
lvm_axi_40_x_setup_hold_test	Demonstrates the effect of UVC driving X for setup and hold value verification.
lvm_axi_50_illegal_burst_test	Demonstrate the illegal AxBURST=2'b11 sending procedure

TESTSUITES

- LVM prepared a lot of quality testsuites, that is specially designed to achieve high quality coverage to verify user's DUT
- The scenario can be seen at the log file, for example

```
[Scenario   3]================================================================================
[Scenario   3] - 1 read or write to a target (if writable) then wait for all done, then move to next target
[Scenario   3] - Repeat for all targets
[Scenario   3] - Repeat above for n times
[Scenario   3]
```

- The runcmd is by providing the plusarg, like below:

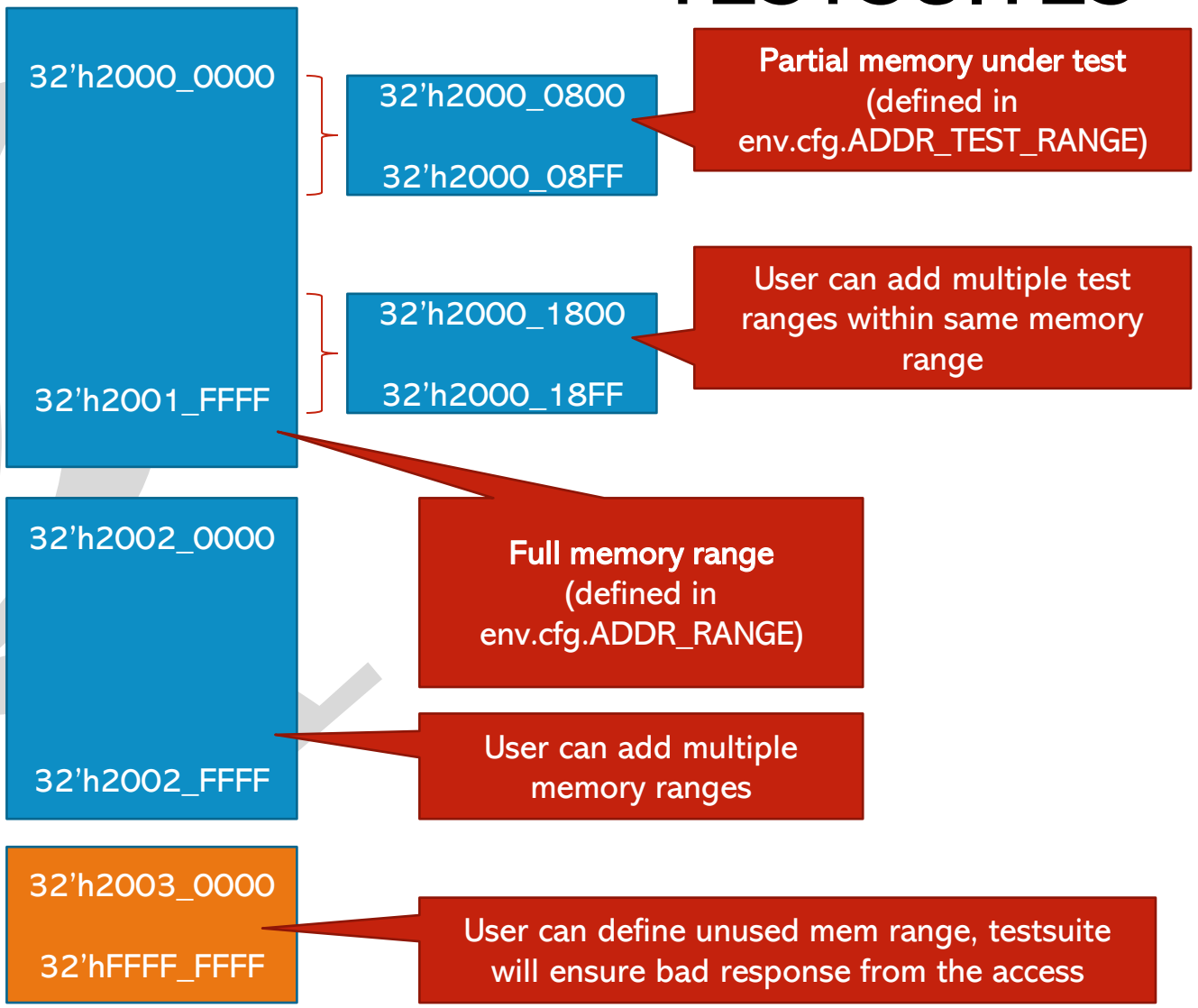
```
make run t=lvm_axi_master_testsuites_test  plusarg="+lvm_axi_total_pkt=150 +lvm_axi_scenario=1"
```

TESTSUITES

- Configurable memory range:
  - For master, if the AxADDR are within the ranges
    - it will track the writes and update internal memory
    - It will check the data for read is matching its internal memory
  - For slave, if the AxADDR are within the ranges
    - It will store the data from the writes into internal memory
    - It will use the data stored as RDATA when it receives read packets
- Configurable memory testing range:
  - The memory sometime is very large that not efficient to cover all addresses in 1 test
  - Thus, user can define multiple smaller ranges of memory

TESTSUITES

- User add the info for “Full memory range” via the add_ADDR_RANGE api.
- add_ADDR_TEST_RANGE api is to add info for partial memory under test
- Detail on the usage is coded at lvm’s base test



TESTSUITES

- User can configurable outstanding transactions, example like below:

```
m_axi_env.cfg.add_ADDR_RANGE(
    .start_addr(32'h8000_0000),     .end_addr(32'h8000_0FFF),
    .read_only(1'b0), .ahb_lite(1'b0),
    .total_outstanding_AR  (5),     .total_outstanding_AW  (15),
    .total_outstanding_W   (15),    .total_outstanding_AW_W(15)
);
```

Argument name	Purpose
start_addr	memory block start address
end_addr	memory block end address
read_only	Whether this memory block cannot be written
ahb_lite	Is this memory resides under AXI-AXI bridge, as AXI lite slave?
total_outstanding_AR	How many outstanding AR (targeting this memory) can be presented on AXI bus without R coming back?
total_outstanding_AW	How many outstanding AW (targeting this memory) can be presented on AXI bus without presenting W?
total_outstanding_W	How many outstanding W (targeting this memory) can be presented on AXI bus without presenting AW?
total_outstanding_AW_W	How many outstanding AW+W (targeting this memory) can be presented on AXI bus without B coming back?

TESTSUITES

- User can configurable out of bound memory range using the following code:

```
// Example unused address ranges
m_axi_env.cfg.add_BAD_ADDR_RANGE(
    .start_addr   (32'h8000_1000),
    .end_addr     (32'hFFFF_FFFF),
    .expected_resp({LVM_AXI_SLVERR})
);
```

Argument name	Purpose
start_addr	Out of bound start address
end_addr	Out of bound end address
expected_resp	Expected response for access to these ranges

TESTSUITES

- User can configurable out of bound memory access to be limited to 1 beat (if needed) using the following code:

```
// Example to constrain all bad packets are 1 beat access
m_axi_env.cfg.bad_packet_in_1_beat = 1'b1;
```

STEP BY STEP INTEGRATION GUIDE

DOWNLOADING THE LVM AXI VIP

1. Request a copy of the LVM AXI by emailing LVM and specify which EDA tool you are using (Synopsys VCS, Questa sim, or Cadence xrun).
2. Unzip the package and copy to unix designated location, for example:

```
/project/home/verification/lvm_axi
```

3. Now set the UNIX variable to point to this path.

```
setenv LVM_AXI_PATH /project/home/verification/lvm_axi
```

4. For first sanity check, launch this cmd and make sure it is working, where you shall see a passing status. This proves that the LVM AXI copy is correct.

```
cd $LVM_AXI_PATH/sim
make sim
```

ADDING FILELIST AND SWITCHES

1. Add LVM VIP filelist into top of your testbench compilation filelist, by referring to <lvm_axi>/sim/top.f

```
+incdir+$LVM_AXI_PATH/tb
+incdir+$LVM_AXI_PATH/../lvm_class
// lvm class
$LVM_AXI_PATH/../lvm_class/lvm_macros.svp
$LVM_AXI_PATH/../lvm_class/lvm_pkg.svp
// LVM_AXI UVC filelist
-f $LVM_AXI_PATH/src/lvm_axi.f
...
```

2. Now copy the compilation keys and runtime keys from <lvm_axi>/sim/makefile to your compile and runtime command, respectively.

```
+define+LVM_LICENSE_FOR_<given name> +define+LVM_LICENSE_SINCE_<date>
```

Compile options

```
+LVM_SINCE_<date> +LVM_<name>_<date>  +seed=<random seed>
```

Runtime options

IMPORT PACKAGES DEFINE SIGNAL WIDTH

3. Import the package in your test, virtual sequence package / module top. (Ref: <lvm_axi>/tb/top.sv)

```
import uvm_pkg::*; // your uvm package import
import lvm_pkg::*;
// lvm_axi UVC
import lvm_axi_pkg::*;
```

4. Now determine your signal widths for the AXI signals and add into the top of your base test, or your define file. (Ref: <lvm_axi>/tests/lvm_axi_base_test.sv)

Change LOCK_WIDTH to 2 for AXI3

```
`define LVM_AXI_VALUES #(.WID_WIDTH(4),.RID_WIDTH(4),
.ADDR_WIDTH(`UVM_REG_ADDR_WIDTH),.REGION_WIDTH(4),.LEN_WIDTH(8),.SIZE_WIDTH(3),.BURST_WIDTH(2),.LOCK_WIDTH(1),
.CACHE_WIDTH(4),.PROT_WIDTH(3),.QOS_WIDTH(4),.VALID_WIDTH(1),.AWUSER_WIDTH(32),.READY_WIDTH(1),.DATA_WIDTH(`UVM_R
EG_DATA_WIDTH),.STRB_WIDTH(`UVM_REG_DATA_WIDTH/8),.LAST_WIDTH(1),.WUSER_WIDTH(32),.RRESP_WIDTH(2)
,.BRESP_WIDTH(2),.BUSER_WIDTH(32),.ARUSER_WIDTH(32),.RUSER_WIDTH(32))
```

5. Now adjust the parameter of the AXI protocol checker at<lvm_axi>/tests/lvm_axi_base_test.sv

```
`define LVM_AXI_SVA_VALUES
#(.DATA_WIDTH(`UVM_REG_DATA_WIDTH),.WID_WIDTH(4),.RID_WIDTH(4),.AWUSER_WIDTH(32),.WUSER_WIDTH(32),.BUSER_WIDTH(32
),.ARUSER_WIDTH(32),.RUSER_WIDTH(32),.MAXWBURSTS(50),.MAXWAITS(16),.PROTOCOL(`AXI4PC_AMBA_AXI4),.ADDR_WIDTH(`UVM_
REG_ADDR_WIDTH),.EXMON_WIDTH(4))
```

Details of the parameter are available at the src/sva/Axi4PC.svp

DECLARATION AND CONSTRUCTION

6. Declaration and build_phase at base test (Ref: <lvm_axi>/tests/lvm_axi_base_test.sv)

```
lvm_axi_env `LVM_AXI_VALUES          m_axi_env;
uvm_status_e                         status;        // UVM status
uvm_reg_data_t                       myrdata;       // Read Data
// API outputs
`LVM_AXI_API_OUTPUT
// Other variables
int                                  lock;
logic   [3:0]                        id = 'h0;
```

```
// At build phase, construct the lvm_axi UVC
function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    m_axi_env =     lvm_axi_env `LVM_AXI_VALUES::type_id::create("m_axi_env", this);
    ...
endfunction
```

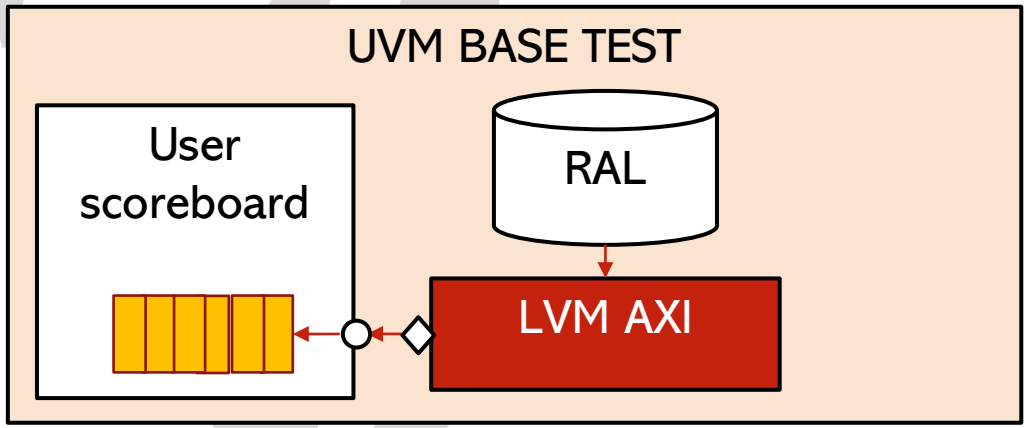
CONNECTING & CONFIGURING THE UVC

7. Connect phase at base test (Ref: <lvm_axi>/tests/lvm_axi_base_test.sv)

```
function void connect_phase (uvm_phase phase);
    super.connect_phase(phase);
    // Connect to your RAL's default map
    if(m_axi_env.cfg.has_prd)
        m_axi_env.prd.map      = urm.default_map;

    if(m_axi_env.cfg.has_adp)
        urm.default_map.set_sequencer(m_axi_env.agt.sqr,
        m_axi_env.adp);
    ...
```

```
    if(m_axi_env.cfg.has_prd)
        urm.default_map.set_auto_predict(0);
    else
        urm.default_map.set_auto_predict(1);

    // Connection to your SBD / Coverage
    m_axi_env.port.connect(<your sbd export>);
    m_axi_env.port.connect(<your cov export>);
    // Choose your working protocol
    m_axi_env.cfg.protocol = LVM_AXI4; // LVM_AXI3
endfunction
```

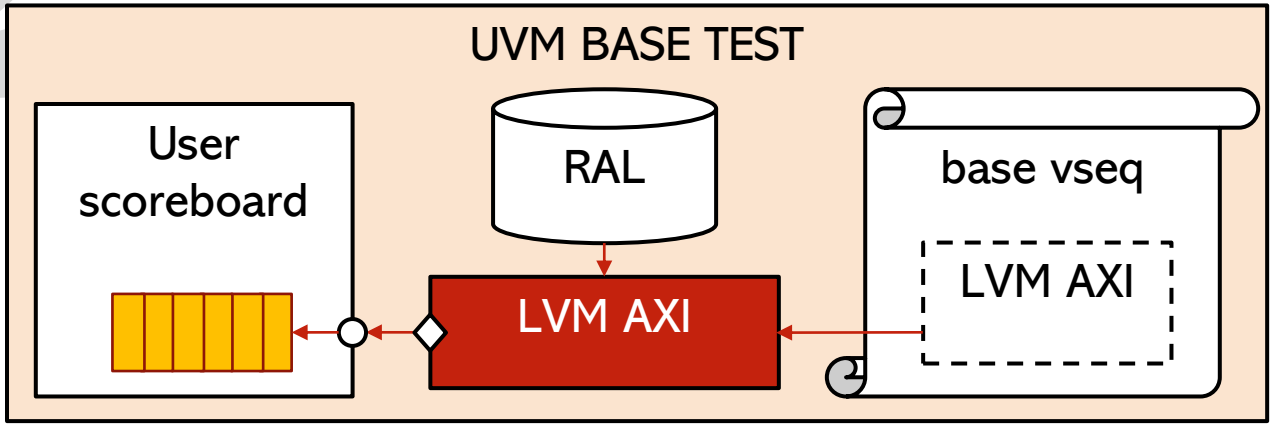
WHAT YOU HAVE NOW



OPTIONAL STEP

8. [Optional] In case you need to add lvm_axi env handle at your virtual sequence, you can repeat the declarations in step 5, inside your virtual sequence.
   - After that, go to base test and add the connection:

```
function void connect_phase (uvm_phase phase);
    super.connect_phase(phase);
    <vseq>.axi_env = m_axi_env;
endfunction
```



CREATE WIRES, SUPPLY CLOCKS, INSTANTIATING INTERFACE, SVA AND CONNECTIONS

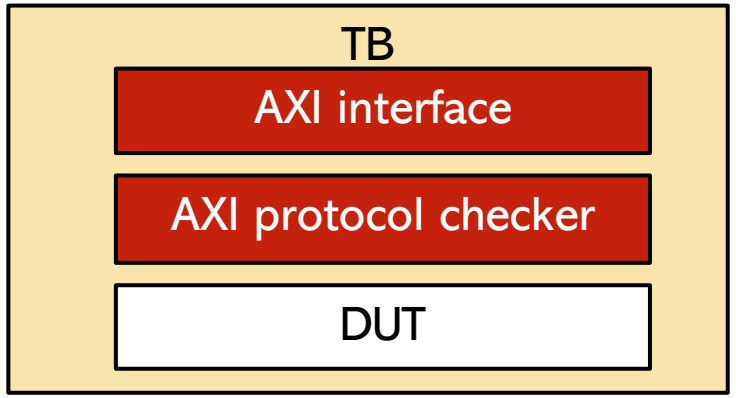
9. Declare all AXI wires in module top, and supply clock and reset to the wires:
   (Ref: <lvm_axi>/tb/top.sv)

```
// User defined interface input signals
wire                        ACLK                ;
wire                        ARESETn             ;
// User defined interface signals
wire [3:0]                  AWID                ;
wire [31:0]                 AWADDR              ;
...
wire                        CACTIVE             ;
// Supply clock and reset
assign ACLK    = <your clock supply>;
assign ARESETn = <your axi reset signal>;
```

CREATE WIRES, SUPPLY CLOCKS, INSTANTIATING INTERFACE, SVA AND CONNECTIONS

10. Include the lvm_axi_connection.sv (mostly need modifications)

```
// Instantiate the interface, set ConfigDB, connecting wire to interface

`include "lvm_axi_connection.sv"


// Example Connections of wires to your DUT

// assuming DUT port list are the same as wire name, else use explicit
connection to replace ".*" below

<your RTL module>     dut(.*);
```



SVA INSTANTIATION SELECTION

11. [No action if it is AXI4] Change to instantiate AXI3 PC in lvm_axi_connection.sv

```
/* Comment out AXI4
Axi4PC `LVM_AXI_SVA_VALUES lvm_axi4_sva (
    ...
);
*/

AxiPC `LVM_AXI3_SVA_VALUES lvm_axi_sva (
    ...
);
```

CREATING YOUR FIRST TEST

12. Now create a new testcase and add the following API (Ref: <lvm_axi>/tests/lvm_axi_03_basic_rw_test.sv)

```
// at main phase
m_axi_env.drv_wait_output(LVM_ON);
m_axi_env.read(
    .ARLOCK     ('0),
    // output cannot be skipped
    .RID        (RID),
    .RDATA      (RDATA),
    .RRESP      (RRESP),
    .RUSER      (RUSER)
);
```

```
m_axi_env.write(
    .AWLOCK     ('0),
    // output cannot be skipped
    .BID        (BID),
    .BRESP      (BRESP),
    .BUSER      (BUSER)
);
```

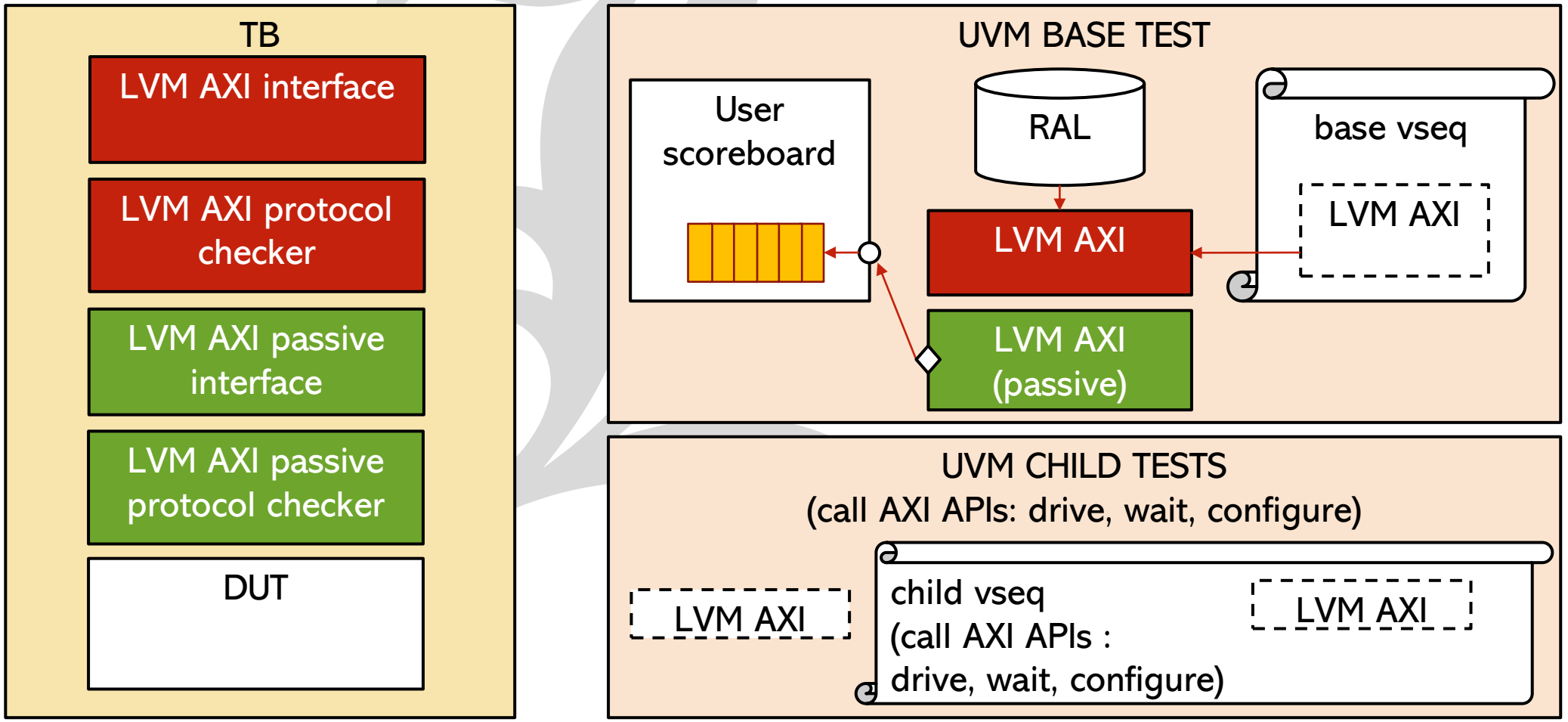
INSTALLING LVM AXI AS PASSIVE UVC

PASSIVE AXI UVC

- LVM AXI UVC can be installed as a passive UVC also.
- This is when user has existing AXI master UVC, and wishes to evaluate the LVM AXI UVC.
- To enable this “mode”, which has been embedded in the self-test testbench, the user just needs to add passive_axi=on in the run command. For example:

```
make sim passive_axi=on t=lvm_axi_01_ral_access_test
```

EXAMPLE OF TWO LVM AXI IN THE TESTBENCH



INTEGRATION GUIDE

1. Follow steps 1 to 5 from previous guide.
2. Declaration and build_phase at base test (Ref: <lvm_axi>/tests/lvm_axi_base_test.sv)

```
lvm_axi_env `LVM_AXI_VALUES         p_axi_env;
```

```
// At build phase, construct the lvm_axi UVC
function void build_phase (uvm_phase phase);
    ...
    uvm_config_db#(uvm_active_passive_enum)::set(this, "*p_axi_env*", "is_active", UVM_PASSIVE);
    p_axi_env   =     lvm_axi_env `LVM_AXI_VALUES::type_id::create("p_axi_env", this);
    // Turning off master env's predictor (if you are using LVM's AXI UVC)
    uvm_config_db#(bit)::set(this, "*m_axi_env*", "has_prd", 1'b0);
endfunction
```

CONNECTING THE PASSIVE UVC

3. Connect phase at base test (Ref: <lvm_axi>/tests/lvm_axi_base_test.sv)

```
function void connect_phase (uvm_phase phase);

        super.connect_phase(phase);

        // Connect to your RAL's default map

        if(p_axi_env.cfg.has_prd)

            p_axi_env.prd.map      = urm.default_map;

        // Connection to your SBD / Coverage

        p_axi_env.port.connect(my_sbd.user_export);

        // Copying the master memory range settings

        p_axi_env.cfg.copy_RAL_ADDR_RANGE       (m_axi_env.cfg.RAL_ADDR_RANGE       );

        p_axi_env.cfg.copy_RAL_RSVD_ADDR_RANGE  (m_axi_env.cfg.RAL_RSVD_ADDR_RANGE  );

        p_axi_env.cfg.copy_ADDR_RANGE           (m_axi_env.cfg.ADDR_RANGE           );

        p_axi_env.cfg.copy_ADDR_TEST_RANGE      (m_axi_env.cfg.ADDR_TEST_RANGE      );

        p_axi_env.cfg.copy_BAD_ADDR_RANGE       (m_axi_env.cfg.BAD_ADDR_RANGE       );

    endfunction
```

CREATE WIRES, SUPPLY CLOCKS, INSTANTIATING INTERFACE, SVA AND CONNECTIONS

4. By reusing all AXI wires in module top, including the clock supply and reset, connect them to AXI passive interface:

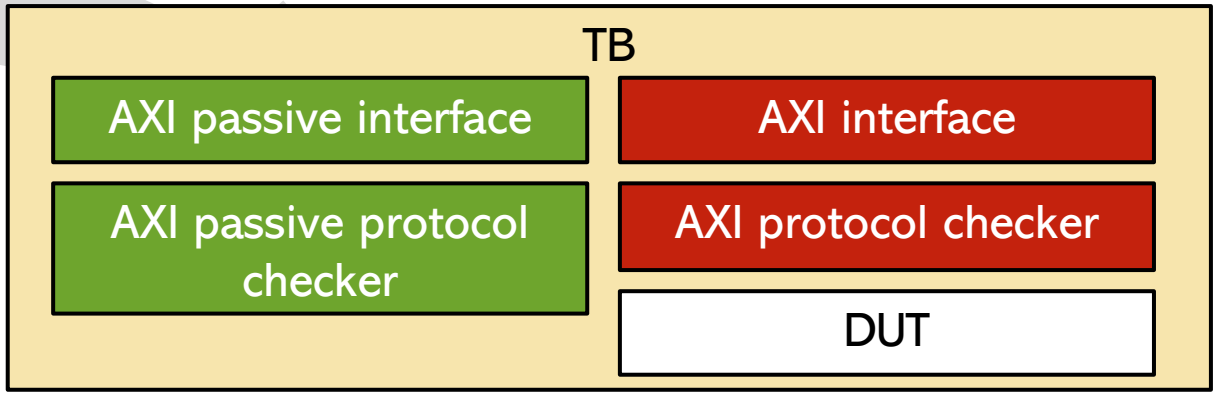
   (Ref: <lvm_axi>/tb/top.sv)

```
// User defined interface input signals
wire                        ACLK                ;
wire                        ARESETn             ;
// User defined interface signals
wire [3:0]                  AWID                ;
wire [31:0]                 AWADDR              ;
...
wire                        CACTIVE             ;
// Supply clock and reset
assign ACLK    = <your clock supply>;
assign ARESETn = <your axi reset signal>;
```

CREATE WIRES, SUPPLY CLOCKS, INSTANTIATING INTERFACE, SVA AND CONNECTIONS

5. Include the lvm_p_axi_connection.sv (mostly need modifications)

```
// Instantiate the interface, set ConfigDB, connecting wire
to interface

`include "lvm_p_axi_connection.sv"


// Example Connections of wires to your DUT

// assuming DUT port list are the same as wire name, else use
explicit connection to replace ".*" below

<your RTL module>     dut(.*);
```



PASSIVE UVC

- With this UVC the user will be able to:
  - Dump AXI traffic into a file.
  - Start checking the AXI protocol on the signals.
  - Start writing the UVC's analysis TLM port.

TERMS AND CONDITIONS

IMPORTANT NOTICE

LVM VIP reserves the right to make changes without further notice to any product or specifications herein. LVM VIP does not assume any responsibility for use of any its products for any particular purpose, nor does LVM VIP assume any liability arising out of the application or use of any its products.

Reference

- https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification?lang=en

Revision History

Rev.	Date	Description	Author
2.00	14 March 2021	V2.00 User guide	LVM VIP
2.01	15 March 2021	Added printing API user guide	LVM VIP
2.02	3 April 2021	Added strength note, added write API new arguments: order and delay. Added passive UVC mode. Added watermark, page number. Added note on setup and hold time	LVM VIP
2.03	20 June 2021	Major bug fix at tracker. AWSIZE and ARSIZE consideration. Enhancement for effective data printing in tracker and waveform	LVM VIP
2.04	3 July 2021	Printing fix at tracker. Enhanced tracker and item for effective_data Added info on new events for debug-ability. Add 2 more new tests.	LVM VIP

Revision History

Rev.	Date	Description	Author
2.05	17 August 2021	- Added field access capability of the UVC - Added AXI4-lite support - Update setup and hold time user configuration API	LVM VIP
2.06	25 Dec 2021	- Bug fixes on WID - Added arburst, awburst, arsize, awsize for enum signals	LVM VIP
2.07	29 March 2022	- Added info on impact/source - Fixed tracker display - Added capability to turn off AxQOS, xUSER, AxREGION printing at tracker - Added AxPROT and AxCACHE banner at tracker - Added simplified write (s_write) and read (s_read) API for quick bus size read write API for user, and test 15 as demo	LVM VIP
2.08	2 April 2022	- Added end of test memory content printing - Added out of order B and R scenarios - Busiest traffic scenario	LVM VIP

Revision History

Rev.	Date	Description	Author
2.09	23 April 2022	- Added effective araddr and effective awaddr - Added slave configuration note - Added strength slides	LVM VIP
2.10	8 June 2022	- Added test 17, 18, 19, 21, 22, 23	LVM VIP
2.11	12 June 2022	- Added test 24,25,26 - Enhancement: inactive_lane_X_data -> X injection feature for RDATA for slave UVC and WDATA for master UVC - Added bresp and rresp enum signal to ease debug	LVM VIP
2.11.1	18 June 2022	- Added new APIs rand_write, rand_read, rand_access, rand_1B_access to rand_8B_access, unaligned_write, unaligned_read	LVM VIP
2.11.2	21 June 2022	- Added more tests, AXI3 PC, and enhanced testsuite, and debug signals	LVM VIP
2.12	29 June 2022	- Added care_boundary features and testsuite explanations	LVM VIP
2.12.1	8 July 2022	- Restructure whole testsuite - Enhanced the testsuite messages, add more scenarios	LVM VIP

Revision History

Rev.	Date	Description	Author
2.12.2	23 Sept 2022	- Enhanced the testsuite messages, add more scenarios	LVM VIP
2.12.3	2 Oct 2022	- Enhanced the testsuite messages, add more scenarios	LVM VIP
2.13.0	11 Oct 2022	- Added note on API usage.	LVM VIP
2.13.1	23 Oct 2022	- Added RAL burst access verification	LVM VIP
2.15	2 Feb 2024	- Cleaning up	LVM VIP