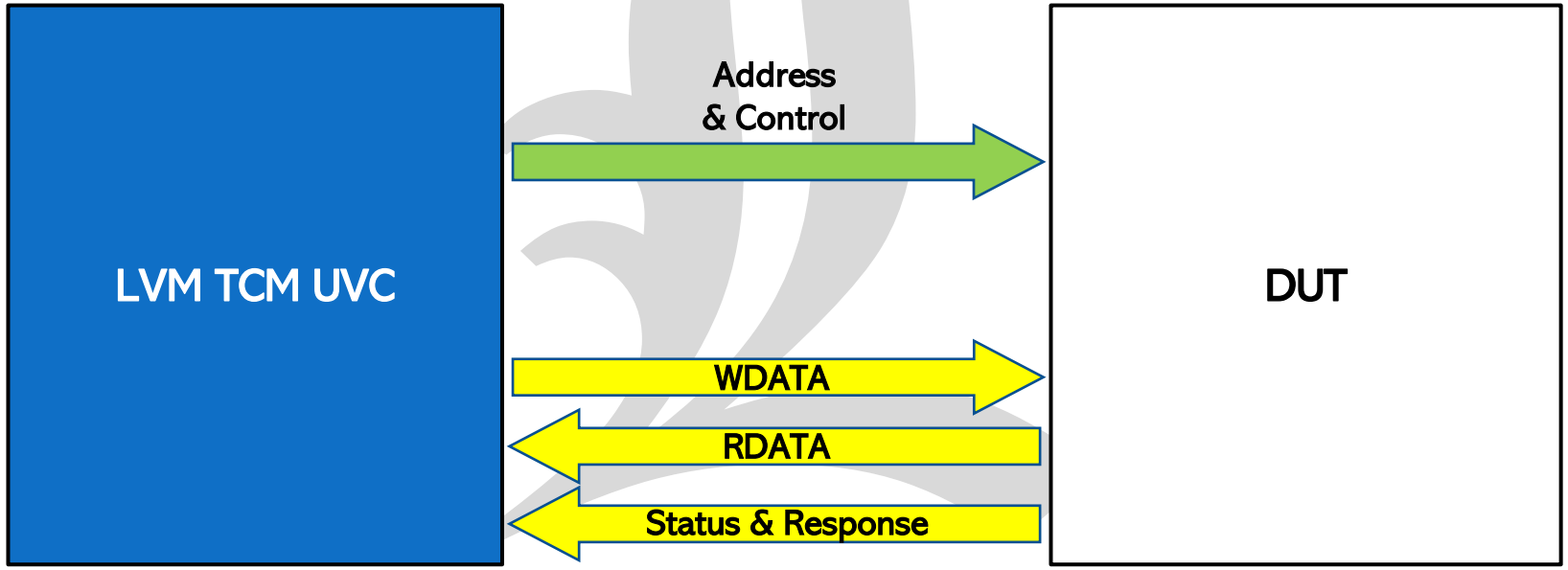
TCM VIP

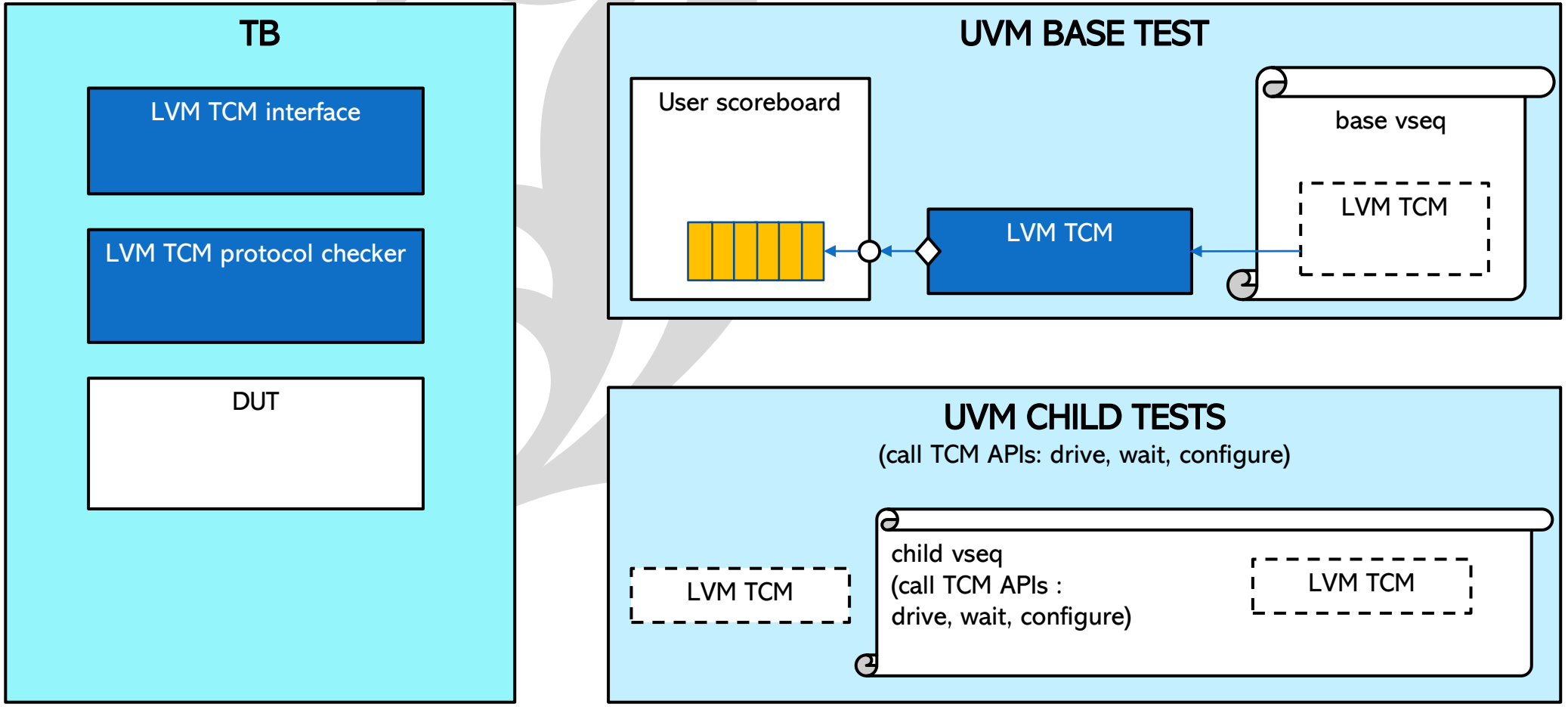
INTRODUCTION

- This is an TCM 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 TCM stimulus with ease (minimal codes).
  - 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 TCM verification in the industry.

THE LVM TCM


LVM TCM UVC
Address
& Control
WDATA
RDATA
Status & Response
DUT

EXAMPLE LVM TCM INSTALLATION



STRENGTHS

WHY CHOOSING LVM VIP?

Strengths

User friendly

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

Robust

Highly configurable.
Parameterized signal width per instance.

High debug-ability

Useful tracker log, interface signals.

Strong & Strict Checker

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

Reusable

Codes on lvm VIP is highly reusable.

Ease of Integration

Minimum steps to integrate.

Light weight

CPU efficient UVC.

Ready testsuite

Provided multiple useful tests to verify TCM slaves DUT

Reset aware

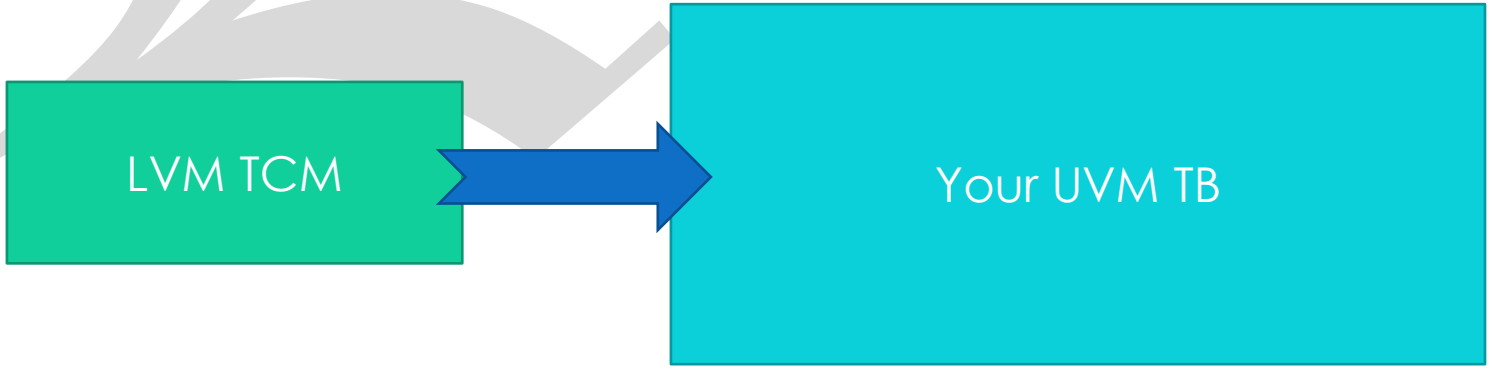
Support reset events.

USER FRIENDLY

Integration and Configuration

INTEGRATION & CONFIGURATION

- Very easy to integrate, simpler than other vendor
  - All steps are demonstrate in the self test 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 etc



DRIVING PART

EASE OF DRIVING STIMULUS

- All fully pipelined.
- API based.
- Various API to control stimulus style
  - Fully pipelined TCM traffic
  - Wait and driving capabilities.
- Most API can be done in 1 line of code.
- Full examples at self-test uvm tests.

REUSABLE CODE

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

```
m_ahb_env.s_write ( .HADDR(32'h2000_0000), .hwdata(32'h11111111) , .hresp(hresp), .hexokay(hexokay));
m_ahb_env. s_read ( .HADDR(32'h2000_0000), .hrdata(hrdata),        .hresp(hresp), .hexokay(hexokay));
`uvm_info(msg_tag, $sformatf("hrdata='h%0h hresp='h%0h, hexokay='h%0h", hrdata, hresp, hexokay),UVM_DEBUG)
```

MONITORING PART

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.

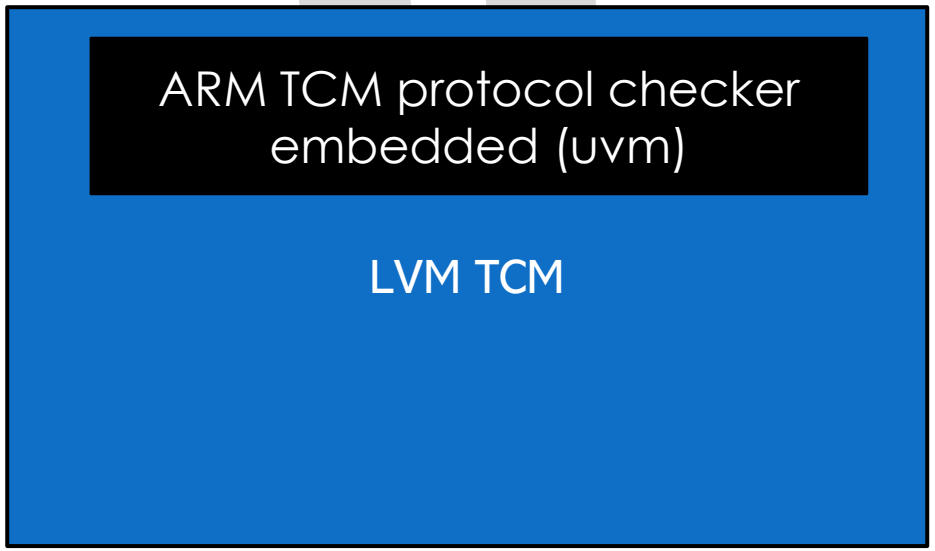
```
// Example Code
`uvm_info(msg_tag, $sformatf("Captured %0s transaction", captured_item.HWRITE?"WRITE":"READ"), UVM_MEDIUM)
`uvm_info(msg_tag, $sformatf("HBURST='h%0h  HMASTLOCK='h%0h  HPROT='h%0h  HSIZE='h%0h  HNONSEC='h%0h  HEXCL='h%0h  HMASTER='h%0h  HAUSER='h%0h  HWUSER='h%0h  HRUSER='h%0h",
    captured_item.HBURST,
    captured_item.HMASTLOCK,
    captured_item.HPROT,
    captured_item.HSIZE,
    captured_item.HNONSEC,
    captured_item.HEXCL,
    captured_item.HMASTER,
    captured_item.HAUSER,
    captured_item.HWUSER,
    captured_item.HRUSER
), UVM_MEDIUM)
```

```
// Per beat
foreach (captured_item.haddr[i]) begin
    if(captured_item.HWRITE == LVM_AHB_WRITE) begin
        `uvm_info(msg_tag, $sformatf("[Beat %-4d]  HTRANS='h%0h  HADDR='h%0h  HWDATA='h%0h  EFF='h%0h  HRESP='h%0h",
            i,
            captured_item.htrans[i],
            captured_item.haddr [i],
            captured_item.HWDATA[i],
            captured_item.effective_hwdata[i],
            captured_item.HRESP [i]
        ), UVM_MEDIUM)


        // the impact of the write for this beat
        // it is printed for every 1 addr
        `uvm_info(msg_tag, $sformatf("IMPACT"), UVM_MEDIUM)
        foreach (captured_item.mem_impact[i].addr[j])
            `uvm_info(msg_tag, $sformatf("    %h <- %2h", captured_item.mem_impact[i].addr[j], captured_item.mem_impact[i].data[j]), UVM_MEDIUM)
```

CHECKING PART

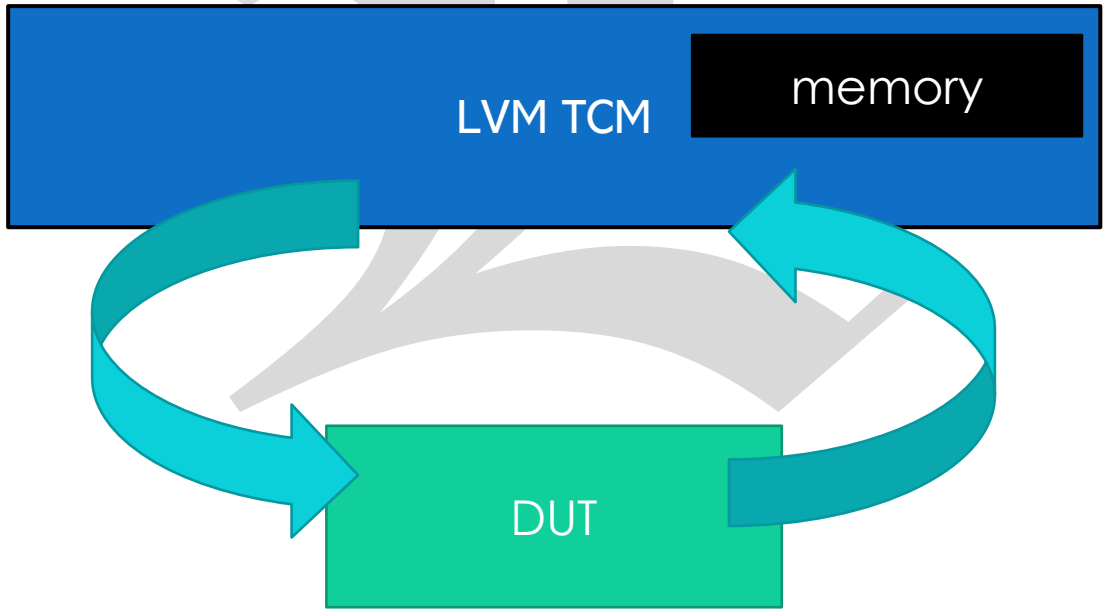
EMBEDDED ARM PROTOCOL CHECKER

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



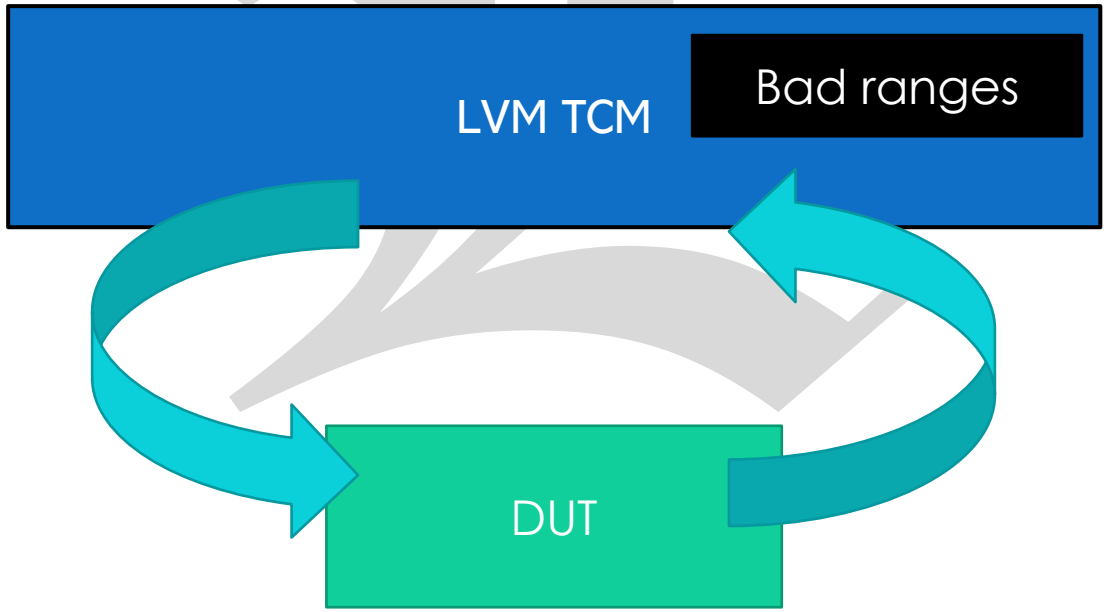
EMBEDDED MEMORY CHECKING

- 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)



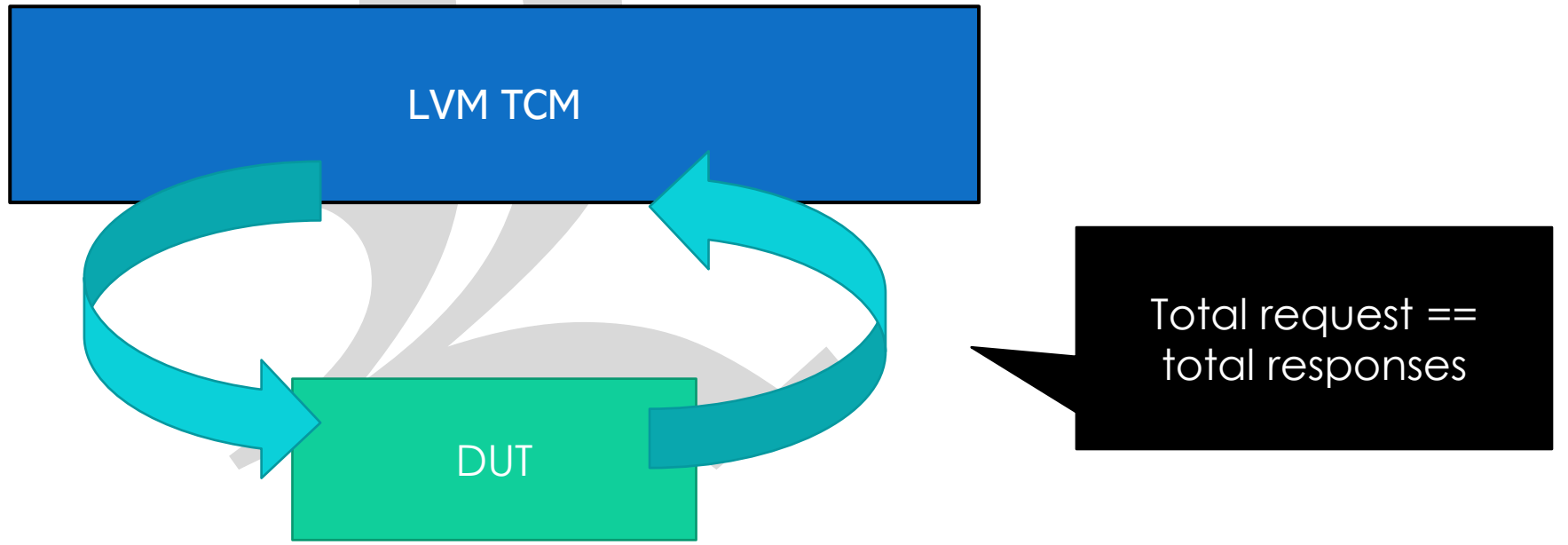
OUT OF BOUND ADDRESS CHECKING

- Built in out of bound address verification scoreboard, where
  - all writes outside valid address range will get HRESP = ERR (default value, user configurable)
  - All reads outside valid address range will get HRESP = ERR (default value, user configurable)



AUTO ENSURE SLAVE COMPLETES ALL REQUESTS

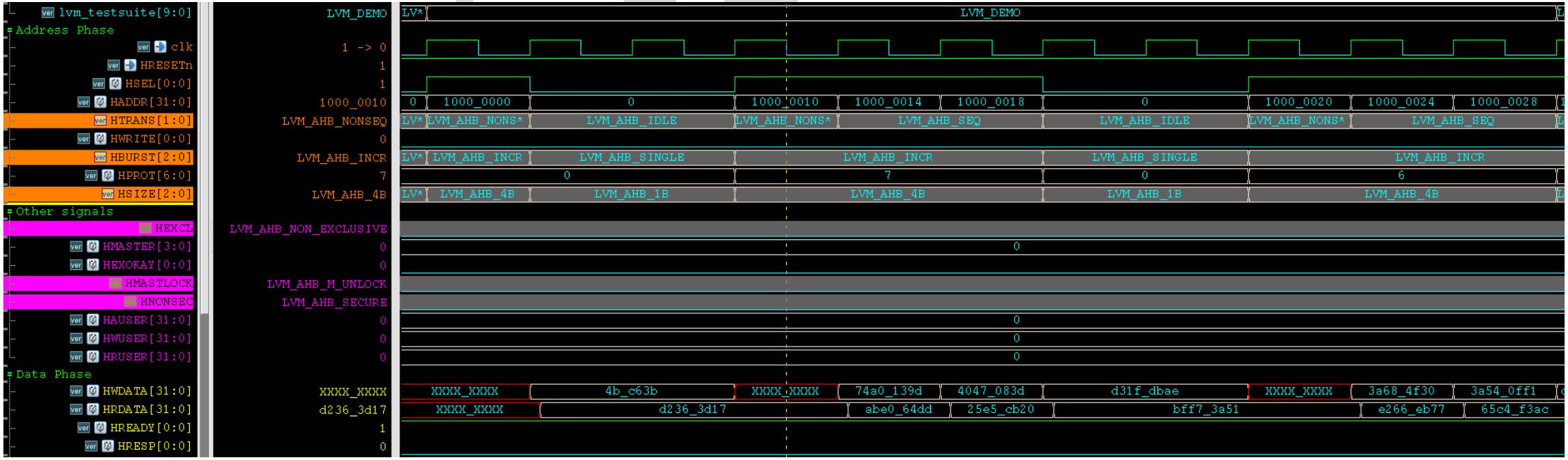
- At the end of test, UVC will ensure slave does not missed out any read / write.



DEBUG PART

ENUM & DEBUG SIGNALS

- Signals are designed in enum, to ease user to read the waveform



COMPREHENSIVE TRACKER

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

```
[Packet 130: READ  2  ]      2890.000 ns
       ------------------------------------------------------------------------------------------------------------------------------------------------------------------------
       CMD= LVM_AHB_READ  HBURST=LVM_AHB_INCR     HMASTLOCK=LVM_AHB_M_UNLOCK  HPROT=7'h07  HSIZE=LVM_AHB_4B

                            HTRANS           HADDR         EFFECTIVE     HRDATA        HRESP                SOURCE
       [Beat 0       31: 0]  LVM_AHB_NONSEQ   32'h10000010  32'habe064dd  32'habe064dd  LVM_AHB_OK           10000013->ab 10000012->e0 10000011->64 10000010->dd
       [Beat 1       31: 0]  LVM_AHB_SEQ      32'h10000014  32'h25e5cb20  32'h25e5cb20  LVM_AHB_OK           10000017->25 10000016->e5 10000015->cb 10000014->20
       [Beat 2       31: 0]  LVM_AHB_SEQ      32'h10000018  32'hbff73a51  32'hbff73a51  LVM_AHB_OK           1000001b->bf 1000001a->f7 10000019->3a 10000018->51

       +-------------------------------------------------------------------------------------------------+
       |   HPROT[3]=0             HPROT[2]=1                HPROT[1]=1             HPROT[0]=1             |
       |   LVM_AHB_UNMODIFIABLE   LVM_AHB_BUFFERABLE        LVM_AHB_PRIVILEGED     LVM_AHB_DATA_ACCESS    |
       +-------------------------------------------------------------------------------------------------+


[Packet 164: WRITE 136]      5450.000 ns
       ------------------------------------------------------------------------------------------------------------------------------------------------------------------------
       CMD=LVM_AHB_WRITE  HBURST=LVM_AHB_INCR     HMASTLOCK=LVM_AHB_M_UNLOCK  HPROT=7'h00  HSIZE=LVM_AHB_2B

                            HTRANS           HADDR         EFFECTIVE     HWDATA        HRESP                IMPACT
       [Beat 0       15: 0]  LVM_AHB_NONSEQ   32'h10000040  16'hed73      32'hxxxxed73  LVM_AHB_OK           10000041<-ed 10000040<-73
       [Beat 1       31:16]  LVM_AHB_SEQ      32'h10000042  16'h6c6a      32'h6c6axxxx  LVM_AHB_OK           10000043<-6c 10000042<-6a
       [Beat 2       15: 0]  LVM_AHB_SEQ      32'h10000044  16'haeef      32'hxxxxaeef  LVM_AHB_OK           10000045<-ae 10000044<-ef
       [Beat 3       31:16]  LVM_AHB_SEQ      32'h10000046  16'h1bd       32'h01bdxxxx  LVM_AHB_OK           10000047<-01 10000046<-bd

       +-------------------------------------------------------------------------------------------------+
       |   HPROT[3]=0             HPROT[2]=0                HPROT[1]=0             HPROT[0]=0             |
       |   LVM_AHB_UNMODIFIABLE   LVM_AHB_NON_BUFFERABLE    LVM_AHB_UNPRIVILEGED   LVM_AHB_INSTRUCTION     |
       +-------------------------------------------------------------------------------------------------+
```

END OF TEST MEMORY PRINTER
END OF TEST SCOREBOARD PRINTER

- It prints total read and total write, ensure test is not empty.

```
--------------------------------
     AHB Scoreboard Report
--------------------------------
    Total Reads      :     129
    Total Writes     :     129
    Total Checked    :     129
--------------------------------
```

TESTSUITE

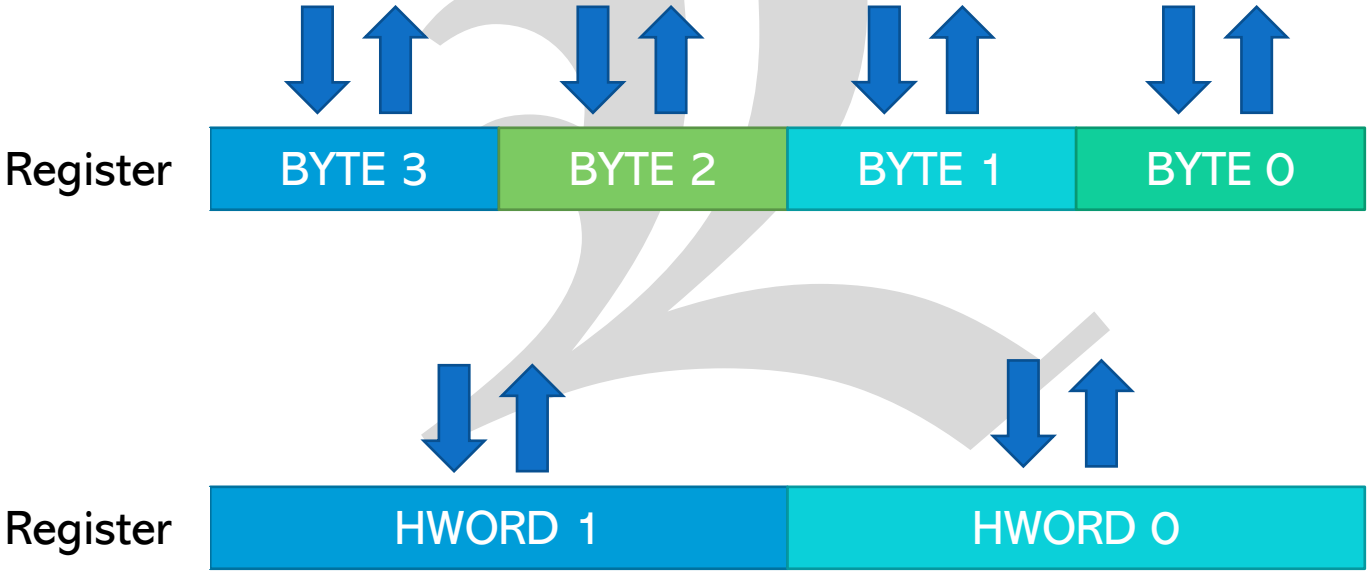
THE BENEFITS FROM TESTSUITE

- Can verify various aspects, for example:
  - Verify the data integrity, with full blast of randomized TCM packets
  - Verify the response of every packet to meet expected value.

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



STRENGTH SUMMARY

- Robust
  - Easy-to-control packet sending style: pipelined or gated styles.
  - 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 TCM packet.
- High debug-ability
  - Tracker file: TCM transactions can be traced in the log file.
  - Interface provides some debug signals.
- Strong checkers
  - Equipped with industry standard protocol checks on the TCM 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 HRESETn 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

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 logic [2:0] {
        LVM_TCM_1B                      = 0,
        LVM_TCM_2B                      = 1,
        LVM_TCM_4B                      = 2,
        LVM_TCM_8B                      = 3,
        LVM_TCM_16B                     = 4,
        LVM_TCM_32B                     = 5,
        LVM_TCM_64B                     = 6,
        LVM_TCM_128B                    = 7
} lvm_ahb_hsize_em;
```

```
typedef enum logic  {
        LVM_TCM_READ                    = 0,
        LVM_TCM_WRITE                   = 1
} lvm_ahb_op_em;
```

```
typedef enum bit  {
        LVM_TCM_OK                      = 0,
        LVM_TCM_HRESP_ERR               = 1
} lvm_ahb_hresp_em;
```

```
typedef enum logic [2:0] {
        LVM_TCM_SINGLE                  = 0,
        LVM_TCM_INCR                    = 1,
        LVM_TCM_WRAP4                   = 2,
        LVM_TCM_INCR4                   = 3,
        LVM_TCM_WRAP8                   = 4,
        LVM_TCM_INCR8                   = 5,
        LVM_TCM_WRAP16                  = 6,
        LVM_TCM_INCR16                  = 7
} lvm_ahb_hburst_em;
```

TCM UVC COMPONENTS

UVC HIERARCHY PATHS

- If user instantiates the LVM TCM UVC under test, the following paths will be valid:
  - uvm_test_top.m_ahb_env
  - uvm_test_top.m_ahb_env.agt
  - uvm_test_top.m_ahb_env.agt.sqr
  - uvm_test_top.m_ahb_env.agt.drv
  - uvm_test_top.m_ahb_env.agt.mon
  - uvm_test_top.m_ahb_env.cfg

TCM 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 TCM 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.

TCM UVC COMPONENTS

- 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 TCM 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 HRESETn fall)
2	exit_reset	<env>.exit_reset; // This makes the UVC to exit reset state (auto done when HRESETn 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
[m_ahb_cfg] clock period = 20.000 ns
[m_ahb_cfg] window of 1  = 10.000 ns
[m_ahb_cfg] window of 0  = 10.000 ns
[m_ahb_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

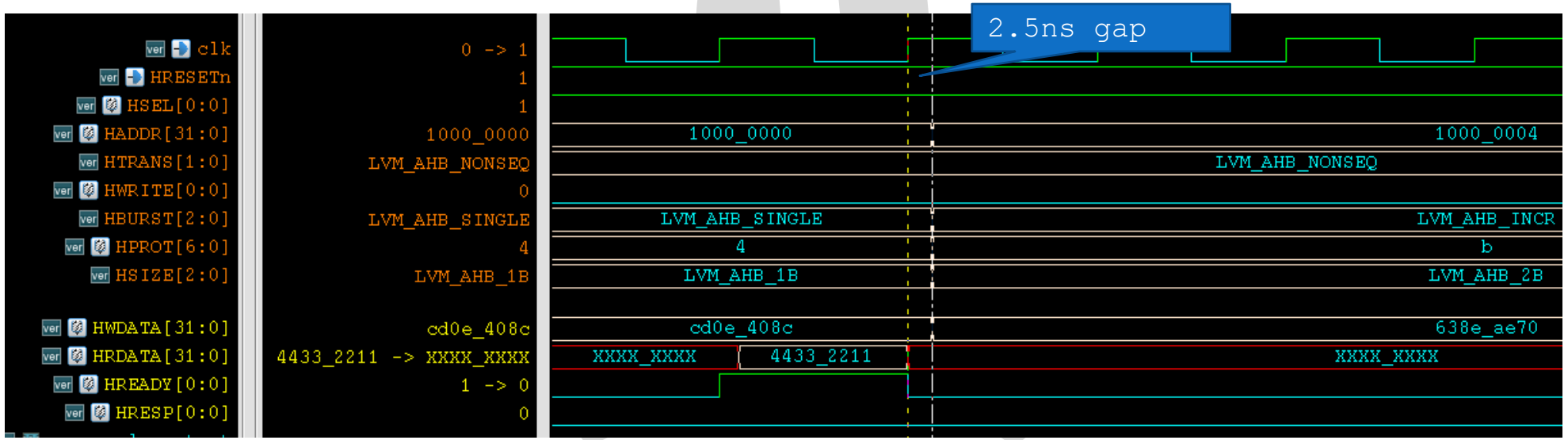
- User can configure TCM5 properties for this UVC, using the following cmd:

```
ahb_env.cfg.TCM5_Extended_Memory_Types   = LVM_TRUE;
ahb_env.cfg.TCM5_Secure_Transfers        = LVM_TRUE;
ahb_env.cfg.TCM5_Exclusive_Transfers     = LVM_TRUE;
ahb_env.cfg.TCM_User_Signals             = LVM_TRUE;
```

- Default it is full TCM5 protocol.

CLOCKING BLOCK CONFIGURATION

- This UVC implements the standard clocking blocks.
- When configuring output (VIP → DUT) to be 2.5ns, the impact is as follows:

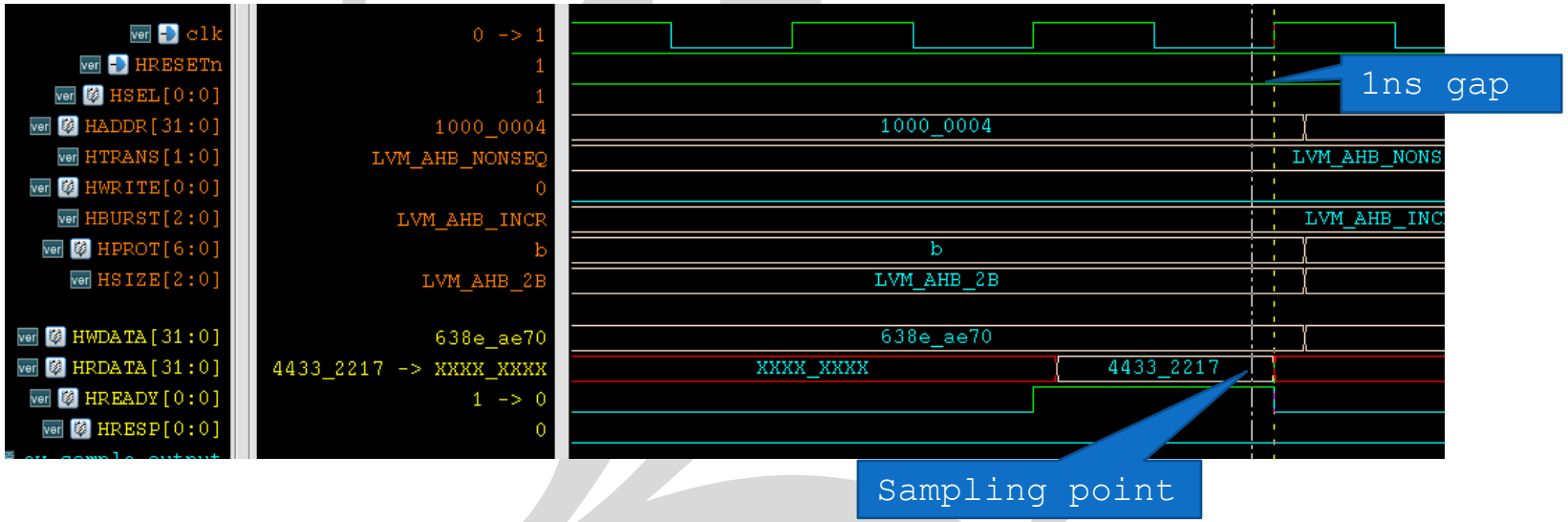


- The cmd:

```
m_ahb_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:

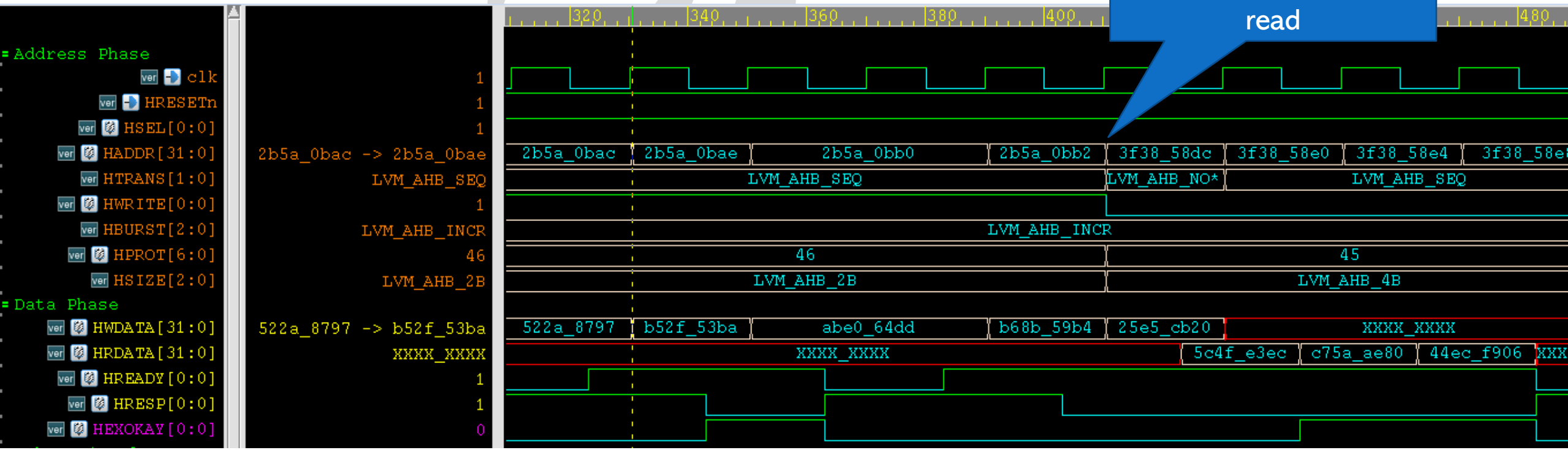
```
m_ahb_env.set_setup_time(1     ,"ns");
```

DRIVER WAIT STYLE

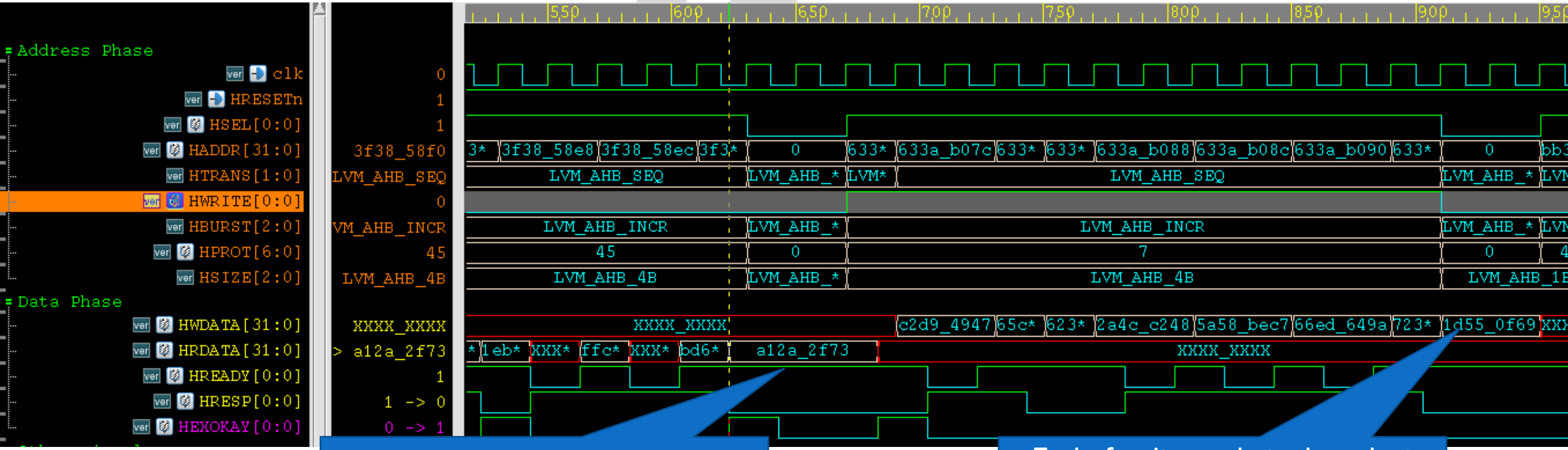
- The driver can be configured by user to wait for the slave response after it has sent a packet. For example,
  - After TCM read is sent, drv can be made to wait for all HRDATAs to come back with responses.
  - After TCM write are sent, drv can be made to wait for all responses to come back.

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 TCM slave.

PIPELINED TRANSACTION DRV_WAIT_OUTPUT(LVM_OFF)



WAIT FOR RESPONSE TRANSACTION DRV_WAIT_OUTPUT(LVM_ON)



End of read packet HRDATA is back, only proceed to next write

End of write packet where last HWDATA is transferred, only proceed to next packet

CARE_BOUNDARY

- This bit will define master UVC to follow the ADDR_TEST_RANGE (more details at “TCM 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_ahb_env.cfg.care_boundary = 1'b0; // default value is 1
```

CHANCE_OF_BUSY & MAX_BUSY

- chance_of_busy is percentage number that define the probability of master UVC to inject busy state during TCM packet sending.
- Default is 0 (will not inject busy)
- To enable the BUSY state

```
m_ahb_env.cfg.chance_of_busy = 10; // a value in percentage
```

- max_busy define maximum clock number of master to be busy.
- Only effective when chance_of_busy != 0
- Default is 5 (5 clks max for busy state, randomized from 1-5)

```
m_ahb_env.cfg.max_busy = 5;
```

CANCEL_ON_ERROR

- This bit controls master UVC to cancel a stimulus during when it sees HRESP==ERROR
- Default is 0 (will not cancel)
- To enable the cancel mode

```
m_ahb_env.cfg.cancel_on_error = 1'b1;
```

LVM TCM 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

```
#
# Note:
#
# -------------------+---------+-------------------------------------------------------------------------+---------+
# Plusargs           : Target  : When they are 1                                                         : Default :
# -------------------+---------+-------------------------------------------------------------------------+---------+
# ever_ready         :  Slave  : Slave drives ready signals = 1 all time                                 : random  :
# bad_hresp          :  Slave  : Percentage for slave to return hresp as LVM_AHB_HRESP_ERR                : random  :
# not_ready_clks     :  Slave  : Total clks for HREADY=0 Slave, if ever_ready==1, it will be made 0       :    5    :
# rand_not_ready     :  Slave  : When not_ready_clks is not set, this bit control not_ready_clks randomization   :    1    :
```

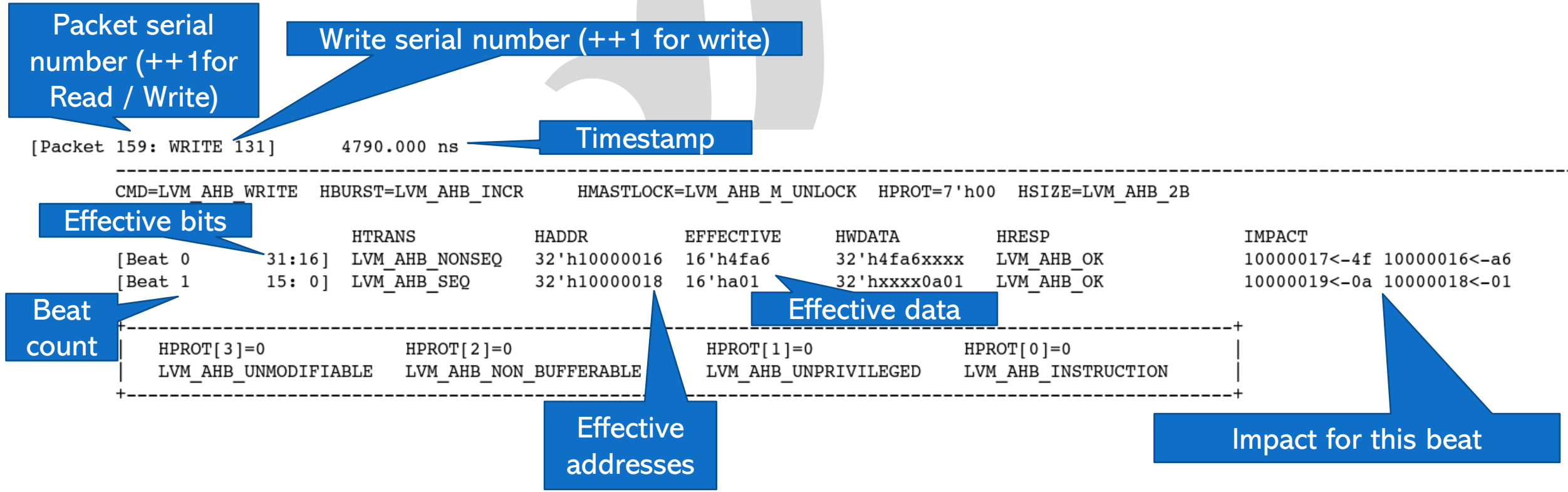
HIGH DEBUG-ABILITY

HIGH DEBUG-ABILITY UVC

- To speed up the debug process, the UVC provides its user with high visibility of TCM bus traffic.
- The simulation with monitor = ON & tracker = ON will enable the tracker file dump
  - It is a file that contains all the TCM 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 HSIZE, HADDR 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 HSIZE, HADDR etc.).

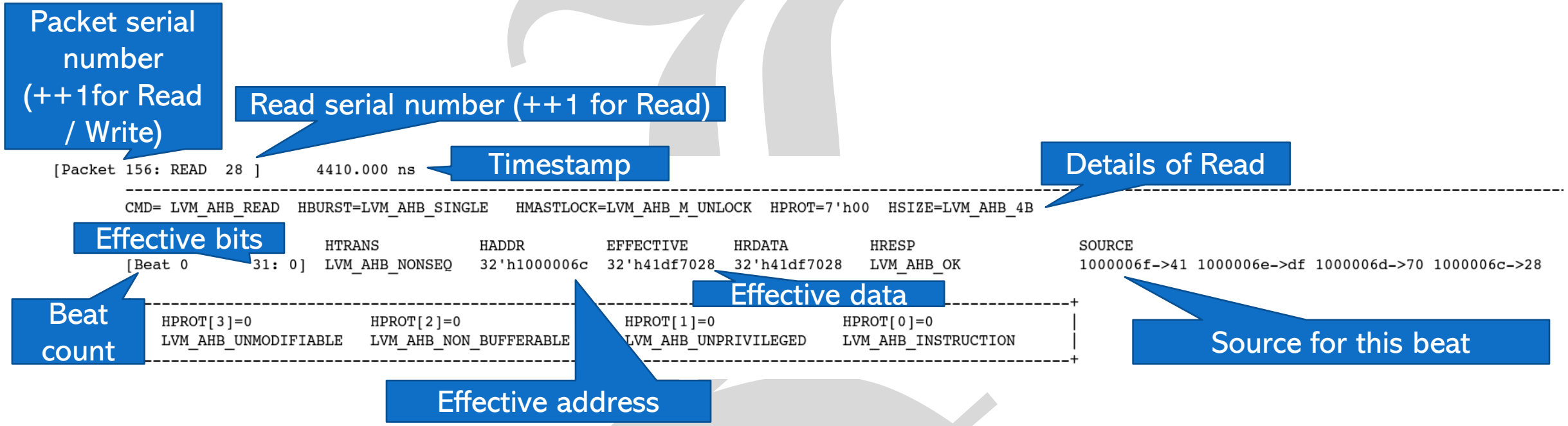
TRACKER LOG

- LVM TCM provides high debugability by preparing log file for all TCM 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 HPROT will be further presented in more readable form:

```
[Packet 156: READ  28 ]      4410.000 ns
        ------------------------------------------------------------------------------------------------------------------------------------------------------------------------
        CMD= LVM_AHB_READ  HBURST=LVM_AHB_SINGLE   HMASTLOCK=LVM_AHB_M_UNLOCK  HPROT=7'h00  HSIZE=LVM_AHB_4B

                             HTRANS           HADDR         EFFECTIVE     HRDATA         HRESP                SOURCE
        [Beat 0       31: 0] LVM_AHB_NONSEQ   32'h1000006c  32'h41df7028  32'h41df7028   LVM_AHB_OK           1000006f->41 1000006e->df 1000006d->70 1000006c->28

        +-------------------------------------------------------------------------------------------------------+
        |   HPROT[3]=0             HPROT[2]=0                 HPROT[1]=0             HPROT[0]=0                |
        |   LVM_AHB_UNMODIFIABLE   LVM_AHB_NON_BUFFERABLE     LVM_AHB_UNPRIVILEGED   LVM_AHB_INSTRUCTION        |
        +-------------------------------------------------------------------------------------------------------+
```

Signal & value

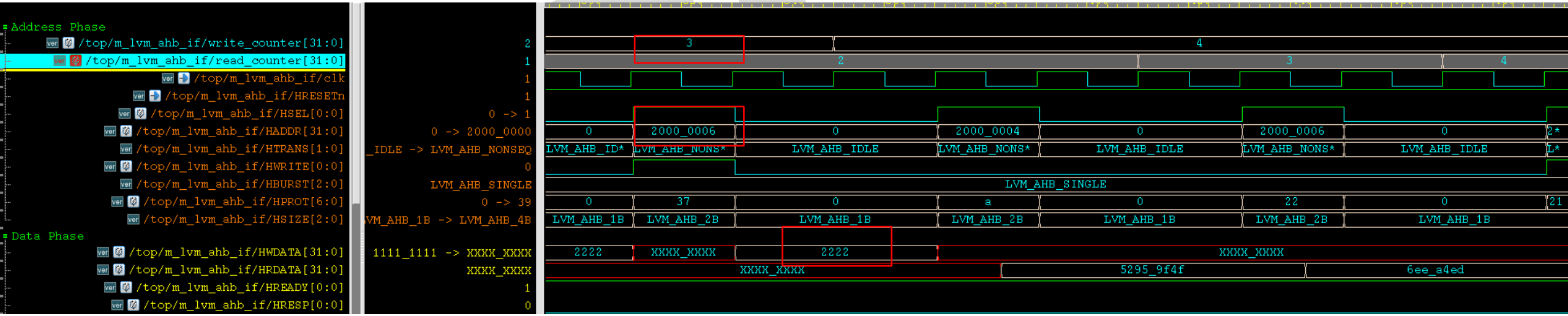
Clear name

READ AND WRITE COUNTER

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

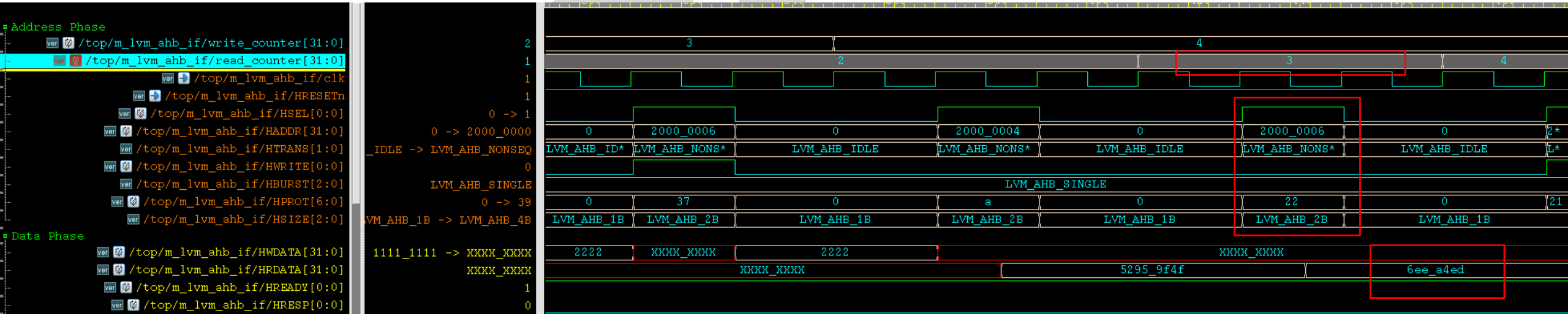
WRITE COUNTER

```
[Packet 4   :  WRITE 3  ]      350.000 ns
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
CMD=LVM_AHB_WRITE  HBURST=LVM_AHB_SINGLE   HMASTLOCK=LVM_AHB_M_UNLOCK  HPROT=7'h37  HSIZE=LVM_AHB_2B  HNONSEC=1'h1  HEXCL=1'h1  HMASTER=4'h1  HAUSER=32'h40761955  HWUSER=32'hab29dded  HRUSER=32'hb6432125

                    HTRANS           HADDR        EFFECTIVE    HWDATA/HRDATA  HRESP            HEXOKAY          IMPACT/SOURCE
[Beat 0      31:16] LVM_AHB_NONSEQ   32'h20000006 16'h0        32'h00002222   LVM_AHB_HRESP_OK LVM_AHB_HEX_ERR  20000007<-00 20000006<-00
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```



READ COUNTER

```
[Packet 6   :  READ  3  ]      470.000 ns
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
CMD= LVM_AHB_READ  HBURST=LVM_AHB_SINGLE   HMASTLOCK=LVM_AHB_M_UNLOCK  HPROT=7'h22  HSIZE=LVM_AHB_2B  HNONSEC=1'h0  HEXCL=1'h1  HMASTER=4'he  HAUSER=32'hf371d044  HWUSER=32'hca6bbad3  HRUSER=32'h3156dab7

                        HTRANS          HADDR          EFFECTIVE     HWDATA/HRDATA     HRESP             HEXOKAY           IMPACT/SOURCE
[Beat 0       31:16]    LVM_AHB_NONSEQ  32'h20000006   16'h6ee       32'h06eea4ed      LVM_AHB_HRESP_OK  LVM_AHB_HEX_ERR   20000007->06 20000006->ee
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```



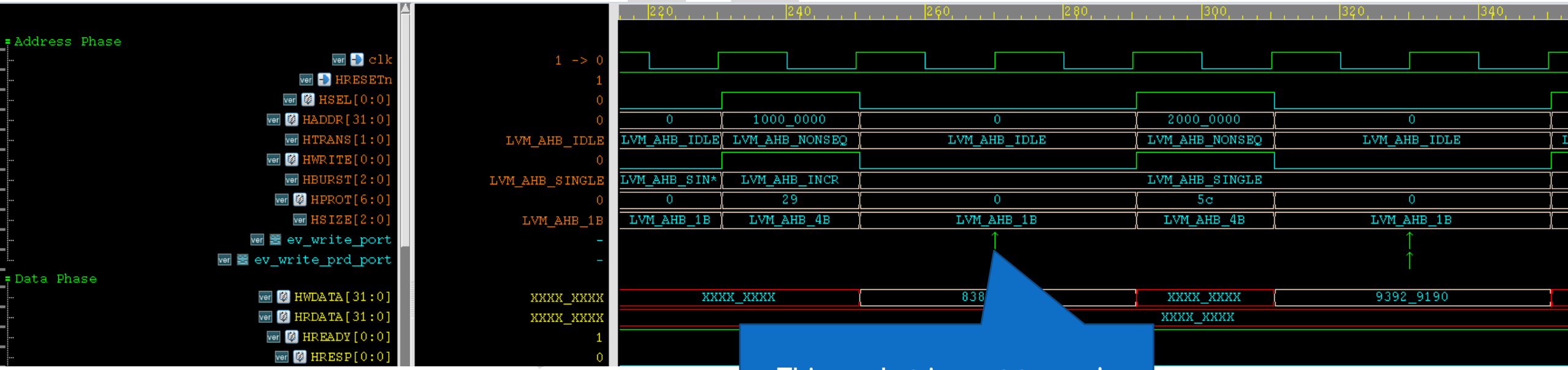
PORT WRITE EVENTS

- 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 TCM read packet is written into env's TLM port
2	ev_write_port	Marks the time where TCM write packet is written into env's TLM port

PORT WRITE EVENTS

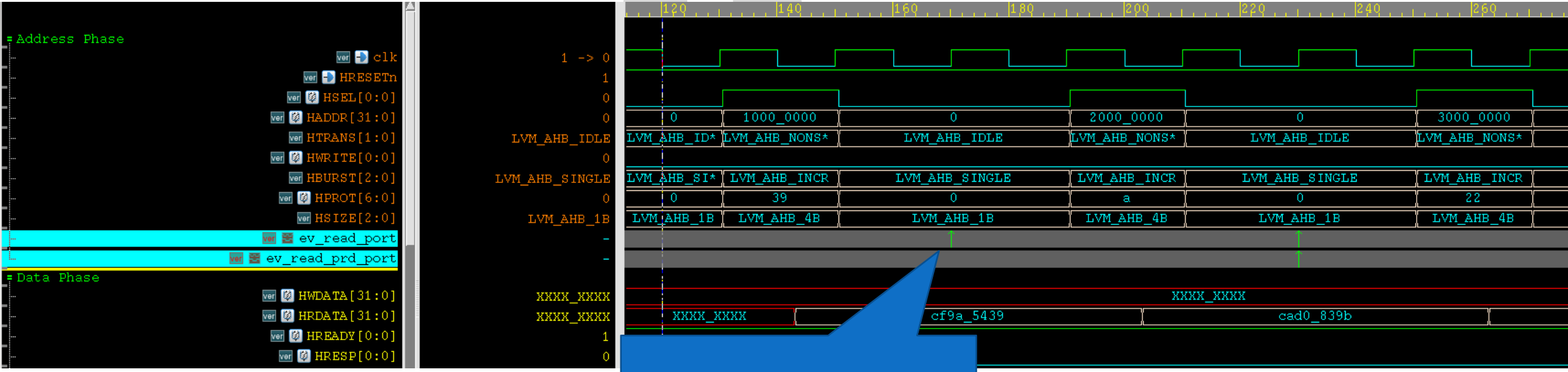
- Events for write packets



This packet is sent to env's port

PORT WRITE EVENTS

- Events for read packets



This packet is sent to env's port

BACKGROUND TCM READ DATA CHECK

BACKGROUND TCM 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 HADDR, HSIZE etc.
- For example, user set UVC to be as such:

```
m_ahb_env.cfg.got_mem        = 1'b1;  // this bit turn on the background data check mechanism
m_ahb_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 TCM write 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 TCM 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_ahb_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 TCM memory, user commonly will initialize the memory with random value, via frontdoor write.
- The API below will initialize the memory within the range with TCM writes (multiple single beat TCM write)

```
m_ahb_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_ahb_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_ahb_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_ahb_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

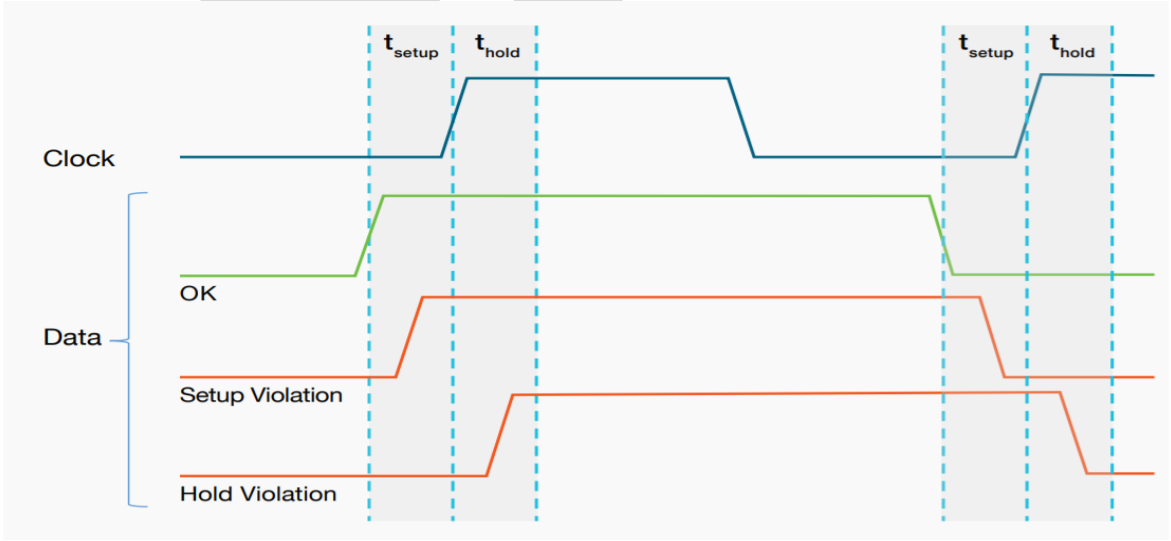
- User can retrieve the current stored data from memory database using the following API as well.

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

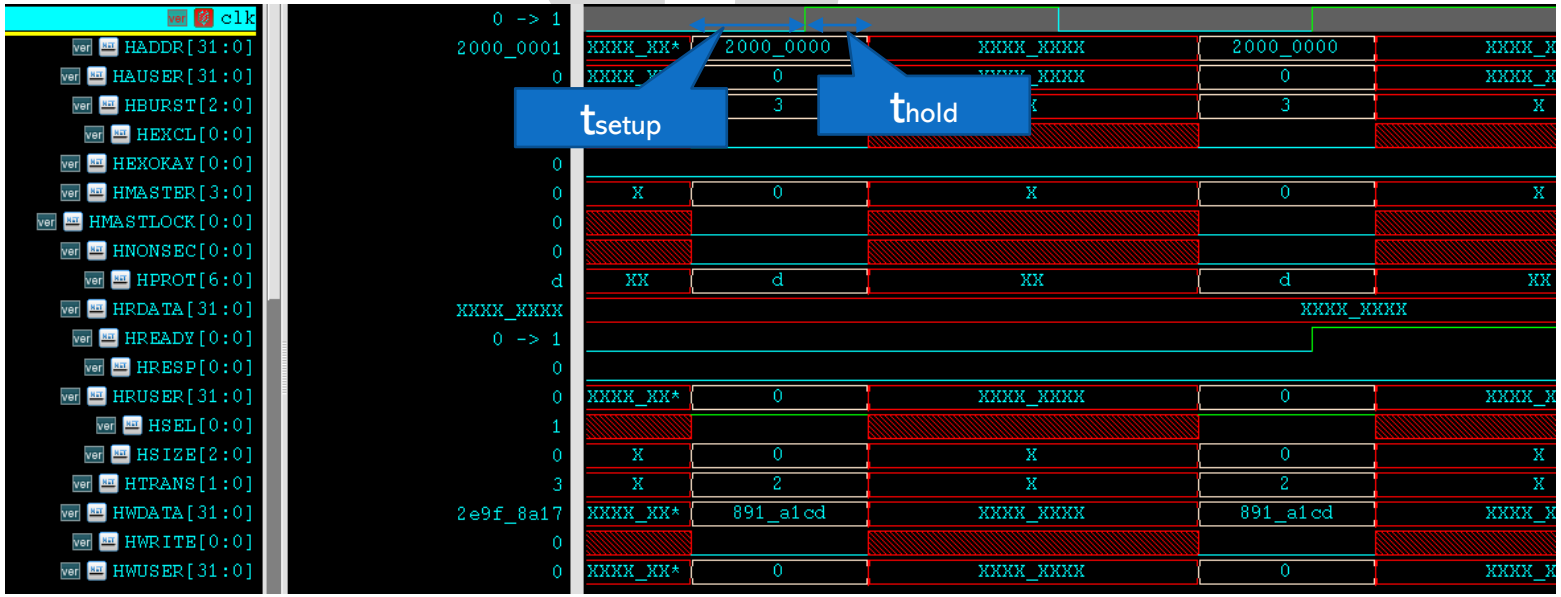


Ref:

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

X INJECTION

- LVM TCM 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_ahb_env.cfg.setup_hold = LVM_X_INJECTION;
m_ahb_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_TCM_CLK_STABLE 3
```

  - Note: this macro is located at src/lvm_ahb_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. s_read
4. s_write
5. write_16
6. read_16
7. write_8
8. read_8
9. idle

1. READ

- UVC sends an TCM 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:
  - HADDR
  - HBURST
  - HMASTLOCK
  - HPROT
  - HSIZE
  - HNONSEC
  - HEXCL
  - HMASTER
  - HWRITE
  - HSEL
  - HAUSER
  - HWUSER
  - HRUSER
  - beats

Outputs:
HRDATA []
HRESP []
HEXOKAY[]

```
logic   [31:0]                  HRDATA [];
logic                           HRESP  [];
logic                           HEXOKAY[];

m_ahb_env.read(
        .HADDR      (32'h1000_004C),
        .HBURST     (LVM_TCM_WRAP4),
        .HSIZE      (LVM_TCM_2B),
        .HMASTLOCK  (LVM_TCM_M_UNLOCK),

        .HRDATA     (HRDATA),
        .HRESP      (HRESP),
        .HEXOKAY    (HEXOKAY)
);

// the parameter not mentioned will be randomized
// For INCR, beats can be defined to tell how much to
// read, else it will be randomized
```

1. READ

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

```
foreach(HRDATA[i]) begin
    hresp   = lvm_ahb_hresp_em  '(HRESP  [i]);
    hexokay = lvm_ahb_hexokay_em'(HEXOKAY[i]);
    `uvm_info(msg_tag, $sformatf("print_read_return [Index %2d] HRESP=%0s, HRDATA=32'h%0h,
              HEXOKAY=%0s", i, hresp.name, HRDATA[i], hexokay.name), UVM_DEBUG)
end
```

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

2. WRITE

- UVC to send a write packet.
- 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:
  - HADDR
  - HBURST
  - HMASTLOCK
  - HPROT
  - HSIZE
  - HNONSEC
  - HEXCL
  - HMASTER
  - HWRITE
  - HWDATA
  - HSEL
  - HAUSER
  - HWUSER
  - HRUSER
  - beats

Outputs:
HRESP []
HEXOKAY[]

```
HWDATA    = new[4];
HWDATA[0] = 32'h0000_1111; // User to mark the data based on alignment
HWDATA[1] = 32'h2222_0000;
HWDATA[2] = 32'h0000_3333;
HWDATA[3] = 32'h4444_0000;

m_ahb_env.write(.HADDR(32'h2000_0010),.HWDATA(HWDATA),.HSIZE(LVM_TCM_2B),
   .HMASTLOCK(LVM_TCM_M_UNLOCK),.HRESP(HRESP),.HEXOKAY(HEXOKAY));

// the parameter not mentioned will be randomized
```

3. S_READ

- It will launch bus size read for 1 beat, same like read, but hard-coded beat and simpler API output
- Arguments accepted:
  - HADDR
  - HBURST
  - HMASTLOCK
  - HPROT
  - HSIZE
  - HNONSEC
  - HEXCL
  - HMASTER
  - HWRITE
  - HSEL
  - HAUSER
  - HWUSER
  - HRUSER
  - beats

Outputs:
hrdata,
hresp ,
hexokay

```
m_ahb_env. s_read (
    .HADDR  (32'h2000_0000),
    .hrdata (hrdata)       ,
    .hresp  (hresp),
    .hexokay(hexokay)
);
```

4. S_WRITE

- It will launch bus size write for 1 beat, same like write, but hard-coded beat and simpler API output
- Arguments accepted:
  - HADDR
  - HBURST
  - HMASTLOCK
  - HPROT
  - HSIZE
  - HNONSEC
  - HEXCL
  - HMASTER
  - HWRITE
  - **hwdata**
  - HSEL
  - HAUSER
  - HWUSER
  - HRUSER
  - beats

Outputs:
hresp ,
hexokay

```
m_ahb_env.s_write (
    .HADDR  (32'h2000_0000),
    .hwdata (32'h11111111),
    .hresp  (hresp),
    .hexokay(hexokay)
);
```

WRITE_16, WRITE_8, READ_16, READ_8

- It will launch bus size write / read for 1 beat, same like read / write, but
  - hard-coded beat,
  - fixed access size (HSIZE), where *_16 means hword, *_8 means byte
  - simpler hwdata entry (no need manual alignment)
  - simpler API output

```
m_ahb_env.write_16( .HADDR(32'h2000_0006), .hwdata(32'h3333)      , .hresp(hresp), .hexokay(hexokay));

m_ahb_env. read_16( .HADDR(32'h2000_0004), .hrdata(hrdata)        , .hresp(hresp), .hexokay(hexokay));
`uvm_info(msg_tag, $sformatf("hrdata=32'h%0h hresp=2'h%0h, hexokay=32'h%0h", hrdata, hresp, hexokay), UVM_DEBUG)


m_ahb_env.write_8 ( .HADDR(32'h2000_000b), .hwdata(32'h44)        , .hresp(hresp), .hexokay(hexokay));

m_ahb_env. read_8 ( .HADDR(32'h2000_0008), .hrdata(hrdata)        , .hresp(hresp), .hexokay(hexokay));
`uvm_info(msg_tag, $sformatf("hrdata=32'h%0h hresp=2'h%0h, hexokay=32'h%0h", hrdata, hresp, hexokay), UVM_DEBUG)
```

9. IDLE

- It will launch bus size read for 1 beat, same like read, but hard-coded htrans == IDLE
- Arguments accepted:
  - HADDR
  - HBURST
  - HMASTLOCK
  - HPROT
  - HSIZE
  - HNONSEC
  - HEXCL
  - HMASTER
  - HWRITE
  - HSEL
  - HAUSER
  - HWUSER
  - HRUSER
  - beats
  - HWDATA[]

Outputs:
HRESP []
HEXOKAY[]

```
m_ahb_env.idle(
                .beats        (1),
                .HRESP        (HRESP),
                .HEXOKAY      (HEXOKAY),
                .HMASTLOCK    (LVM_TCM_M_UNLOCK)
);
```

SUMMARY: IMPACT OF DRV_WAIT_OUTPUT

API name	drv_wait_output=LVM_ON	drv_wait_output=LVM_OFF
read/s_read	[WAIT] The read will be sent and gated until the read data is returned, only then next API is executed. Output arguments like HRDATA, HRESP etc can be used after the API.	[NO WAIT] The read will be sent. Without waiting for HRDATA, HRESP etc, next line after the API will get executed.
write/s_write	[WAIT] The write will be sent and gated until the write response is returned, only then next API is executed. Output arguments like HRESP etc can be used after the API.	[NO WAIT] The write will be sent. Without waiting for HRESP etc, next line after the API will get executed.

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_ahb_env.s_write (
      .HADDR   (32'h2000_0000),
      .hwdata  (32'h11111111),
      .hresp   (hresp),
      .hexokay(hexokay)
);
```

Other variables like HMASTER, HPROT etc are fully randomized based on protocol constraint.

API PART 2: PACKET WAITING

1. wait_all_done

1. WAIT_ALL_DONE

- This API waits for all the TCM packet to be send out by driver and response to return.

```
m_ahb_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 TCM packet to finish sending and all the response 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 NOTE: END OF TEST CHECK

- TCM 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_ahb_env.cfg.end_of_test_check_en = 1'b0;
```

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

API PART 3: PRINTING

1. print_ahb

PRINT_AHB

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

```
`uvm_info(msg_tag, $sformatf("Captured %0s transaction\n%s", captured_item.op.name, captured_item.print_ahb), UVM_MEDIUM)
```

- User will get this at log file:

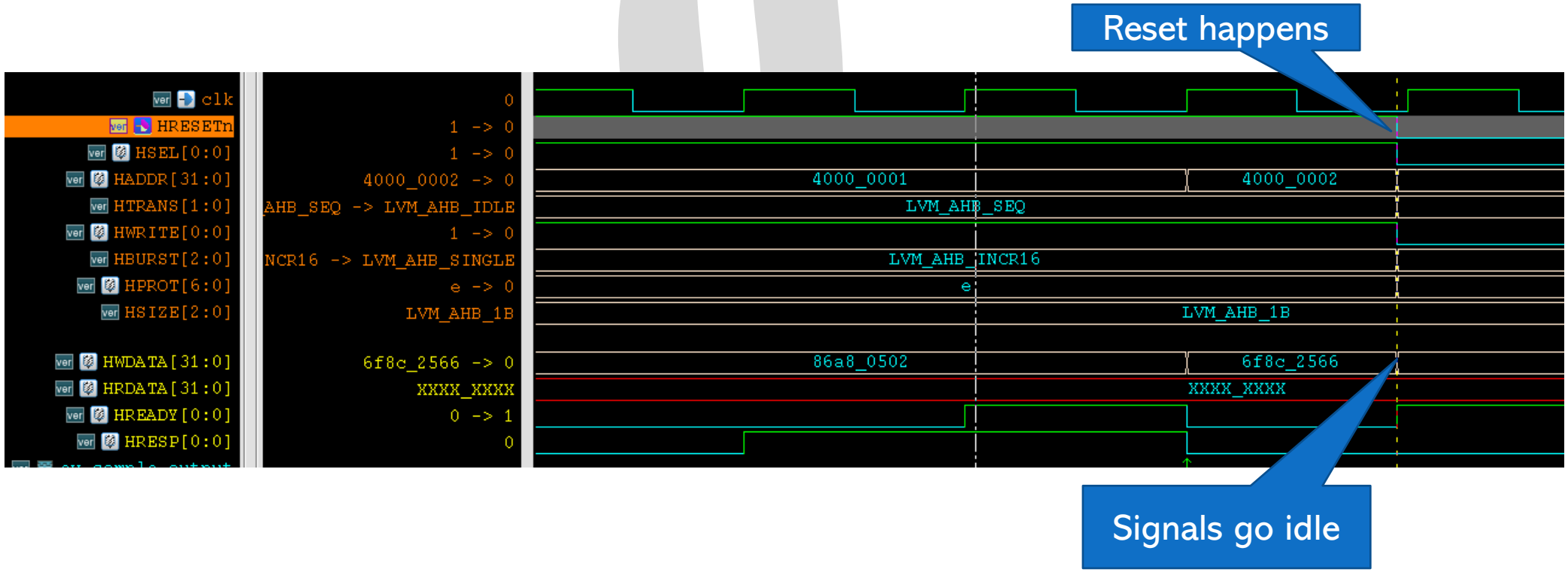
```
UVM_INFO @ 16010.000 ns: uvm_test_top.ahb_sbd [example_ahb_user_sbd] Captured  transaction
      ------------------------------------------------------------------------------------------------------------------------
      CMD= LVM_AHB_READ  HBURST=LVM_AHB_INCR4    HMASTLOCK=LVM_AHB_M_UNLOCK  HPROT=7'h0d  HSIZE=LVM_AHB_1B

                           HTRANS           HADDR         EFFECTIVE     HRDATA         HRESP                SOURCE
      [Beat 0        7: 0]  LVM_AHB_NONSEQ   32'h20000000  8'h1d         32'hxxxxxx1d   LVM_AHB_OK           20000000->1d
      [Beat 1       15: 8]  LVM_AHB_SEQ      32'h20000001  8'h95         32'hxxxx951d   LVM_AHB_OK           20000001->95
      [Beat 2       23:16]  LVM_AHB_SEQ      32'h20000002  8'h53         32'hxx53951d   LVM_AHB_OK           20000002->53
      [Beat 3       31:24]  LVM_AHB_SEQ      32'h20000003  8'hb1         32'hb153951d   LVM_AHB_OK           20000003->b1

      +-------------------------------------------------------------------------------------------------------+
      |   HPROT[3]=1             HPROT[2]=1                  HPROT[1]=0             HPROT[0]=1               |
      |   LVM_AHB_MODIFIABLE     LVM_AHB_BUFFERABLE          LVM_AHB_UNPRIVILEGED   LVM_AHB_DATA_ACCESS      |
      +-------------------------------------------------------------------------------------------------------+
```

- Ref: <lvm_ahb>/tb/example_ahb_user_sbd.sv

RESET AWARE UVC

RESET AWARE UVC

- This UVC is reset aware.
- When HRESETn 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

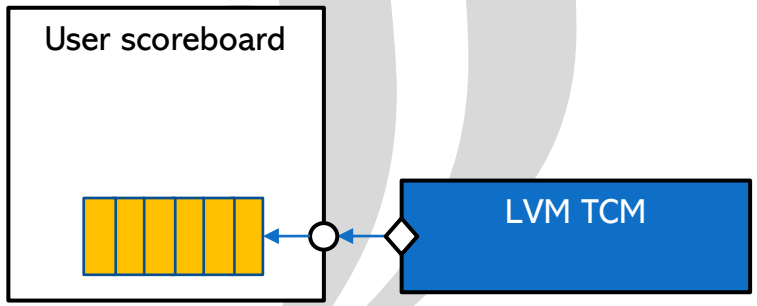
SEQ ITEM VARIABLE NAMES

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

HADDR	HSEL
HBURST	HAUSER
HMASTLOCK	HWUSER
HPROT	HRUSER
HSIZE	HWDATA []
HNONSEC	HRDATA []
HEXCL	HREADY
HMASTER	HRESP []
HTRANS	HEXOKAY[]
HWRITE	beats

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_ahb_user_sbd.sv



USER DEFINED UVM SEQUENCE

- Unlike conventional UVCs , users need not code the UVM sequence manually when using LVM TCM 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_ahb_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 to LVM

TCM TESTSUITES

TCM 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 TCM testsuite is designed to
  - verify the DUT as slave
- User can easily add new testcases by adding into the <lvm_ahb>/tests/testlist.sv and adding the runcmd into <lvm_ahb>/sim/makefile

DEMO TESTCASES

Testname	Purpose
lvm_ahb_02_single_accesses_test	s_write, s_read, write_16, read_16, write_8, read_8 usage examples.
lvm_ahb_03_basic_write_test.sv	Demonstrates simple write API usage: User can provide all arguments or minimum number of arguments.
lvm_ahb_04_basic_read_test.sv	Demonstrates simple read API usage: User can provide all arguments or minimum number of arguments.
lvm_ahb_05_more_write_test.sv	More write examples
lvm_ahb_06_more_read_test.sv	More read examples
lvm_ahb_07_locked_rw_test.sv	TCM lock transaction demonstration
lvm_ahb_08_read_write_test.sv	Mixed traffics
lvm_ahb_09_read_write_others_test.sv	Mixed traffics

DEMO TESTCASES

Testname	Purpose
lvm_ahb_10_rand_read_write_test.sv	More random style read writes
lvm_ahb_11_rw_idle_test.sv	Sending idle
lvm_ahb_20_user_seqs_test.sv	Example for manual uvm sequences
lvm_ahb_30_no_component_test.sv	Do not instantiate some components
lvm_ahb_31_mid_reset_test.sv	Mid reset scenario
lvm_ahb_32_monitor_off_test.sv	Turning OFF monitor
lvm_ahb_33_predictor_off_test.sv	Turning OFF predictor
lvm_ahb_40_x_setup_hold_test.sv	X injection tests
lvm_ahb_master_testsuites_test.sv	TCM testsuite

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:

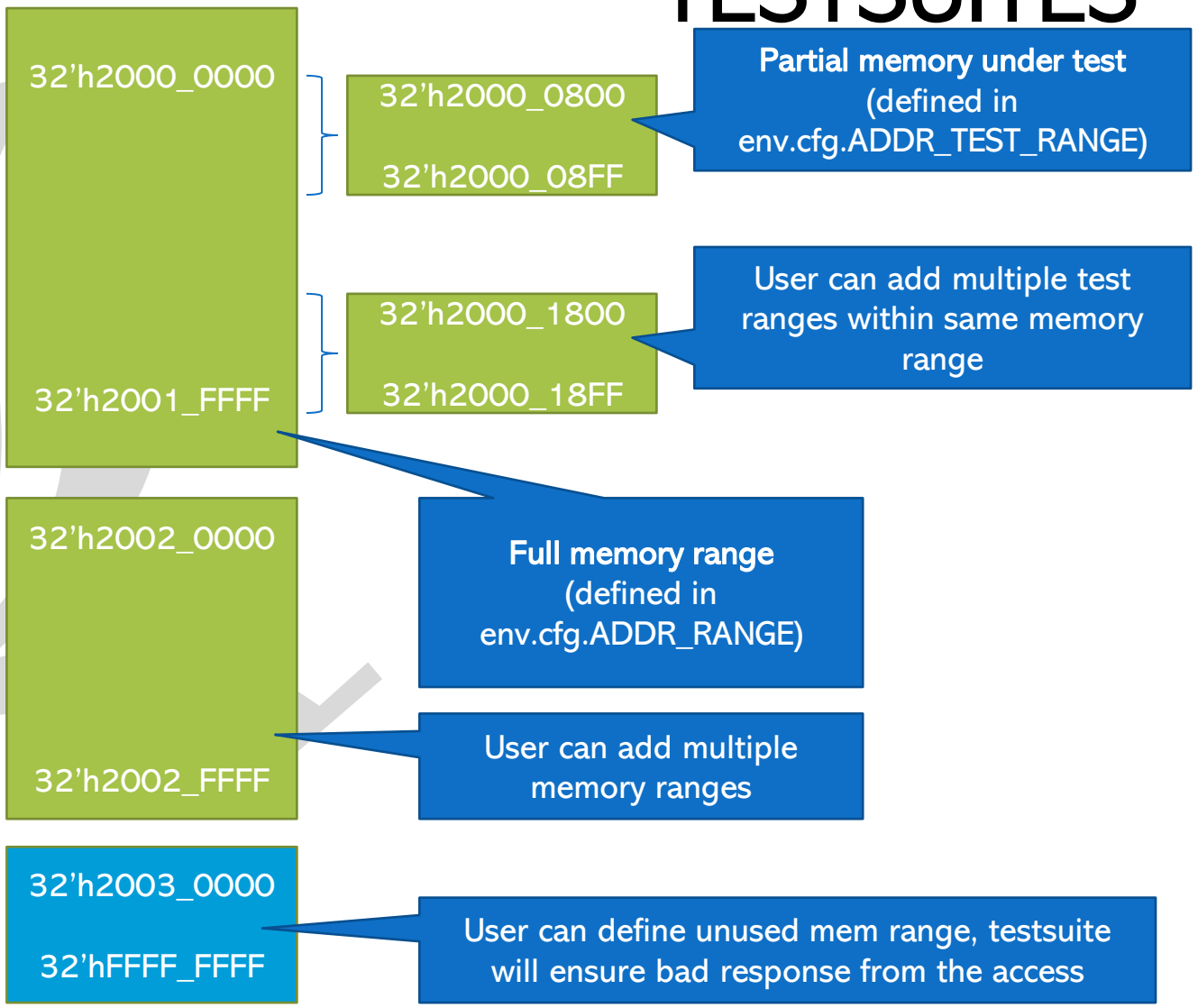
```
make run t=lvm_ahb_master_testsuites_test  plusarg="+lvm_ahb_total_pkt=150 +lvm_ahb_scenario=1"
```

TESTSUITES

- Configurable memory range:
  - For master, if the HADDR 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 HADDR are within the ranges
    - It will store the data from the writes into internal memory
    - It will use the data stored as HRDATA 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 configure ADDR RANGEs as such:

```
m_ahb_env.cfg.add_ADDR_RANGE(
    .start_addr(32'h8000_0000),
    .end_addr  (32'h8000_0FFF),
    .read_only (1'b0),
    .ahb_lite  (1'b0)
);
```

Argument name	Purpose
start_addr	Out of bound start address
end_addr	Out of bound end address
read_only	Whether this memory block cannot be written
ahb_lite	Is this memory resides under TCM-TCM bridge, as TCM lite slave?

TESTSUITES

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

```
// Example unused address ranges
m_ahb_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	memory block start address
end_addr	memory block 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_ahb_env.cfg.bad_packet_in_1_beat = 1'b1;
```

STEP BY STEP INTEGRATION GUIDE

DOWNLOADING THE LVM TCM VIP

1. Request a copy of the LVM TCM 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_ahb
```

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

```
setenv LVM_TCM_PATH /project/home/verification/lvm_ahb
```

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 TCM copy is correct.

```
cd $LVM_TCM_PATH/sim
make sim
```

ADDING FILELIST AND SWITCHES

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

```
+incdir+$LVM_TCM_PATH/tb
+incdir+$LVM_TCM_PATH/../lvm_class
// lvm class
$LVM_TCM_PATH/../lvm_class/lvm_macros.svp
$LVM_TCM_PATH/../lvm_class/lvm_pkg.svp
// LVM_TCM UVC filelist
-f $LVM_TCM_PATH/src/lvm_ahb.f
...
```

2. Now copy the compilation keys and runtime keys from <lvm_ahb>/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_ahb>/tb/top.sv)

```
import uvm_pkg::*; // your uvm package import

import lvm_pkg::*;

// lvm_ahb UVC

import lvm_ahb_pkg::*;
```

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

```
`define LVM_TCM_VALUES
#(.HADDR_WIDTH(`UVM_REG_ADDR_WIDTH),.HBURST_WIDTH(3),.HMASTLOCK_WIDTH(1),.HPROT_WIDTH(7),.HSIZE_WIDTH(3),.HNONSEC
_WIDTH(1),.HEXCL_WIDTH(1),.HMASTER_WIDTH(4),.HTRANS_WIDTH(2),.HWDATA_WIDTH(`UVM_REG_DATA_WIDTH),.HWRITE_WIDTH(1),
.HRDATA_WIDTH(`UVM_REG_DATA_WIDTH),.HREADY_WIDTH(1),.HRESP_WIDTH(1),.HEXOKAY_WIDTH(1),.HSEL_WIDTH(1),.HAUSER_WIDT
H(32),.HWUSER_WIDTH(32),.HRUSER_WIDTH(32))
```

DECLARATION AND CONSTRUCTION

5. Declaration and build_phase at base test (Ref: <lvm_ahb>/tests/lvm_ahb_base_test.sv)

```
lvm_ahb_env `LVM_TCM_VALUES          m_ahb_env;
uvm_status_e                         status;         // UVM status
uvm_reg_data_t                       myrdata;        // Read Data
// API outputs
`LVM_TCM_API_OUTPUT
```

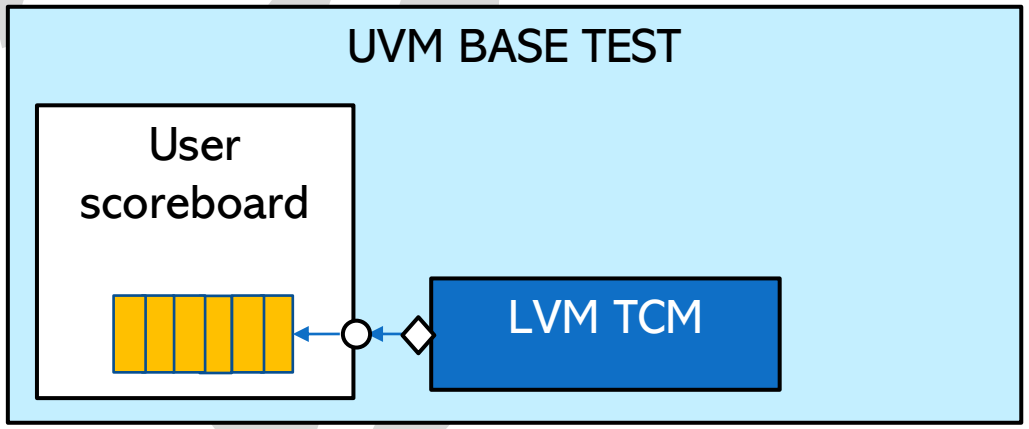
```
// At build phase, construct the lvm_ahb UVC
function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    m_ahb_env =     lvm_ahb_env `LVM_TCM_VALUES::type_id::create("m_ahb_env", this);
    ...
endfunction
```

CONNECTING & CONFIGURING THE UVC

6. Connect phase at base test (Ref: <lvm_ahb>/tests/lvm_ahb_base_test.sv)

```
function void connect_phase (uvm_phase phase);
    super.connect_phase(phase);
     // Connection to your SBD / Coverage
    m_ahb_env.port.connect(<your sbd export>);
    m_ahb_env.port.connect(<your cov export>);
endfunction
```

WHAT YOU HAVE NOW

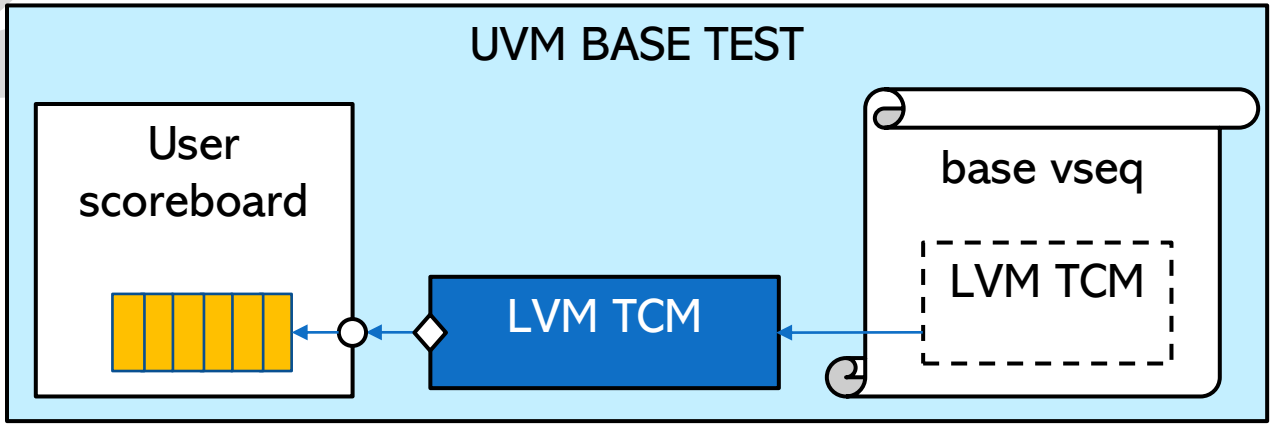


OPTIONAL STEP

7. [Optional] In case you need to add lvm_ahb 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>.TCM_env = m_ahb_env;
endfunction
```



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

8. Declare all TCM wires in module top, and supply clock and reset to the wires:

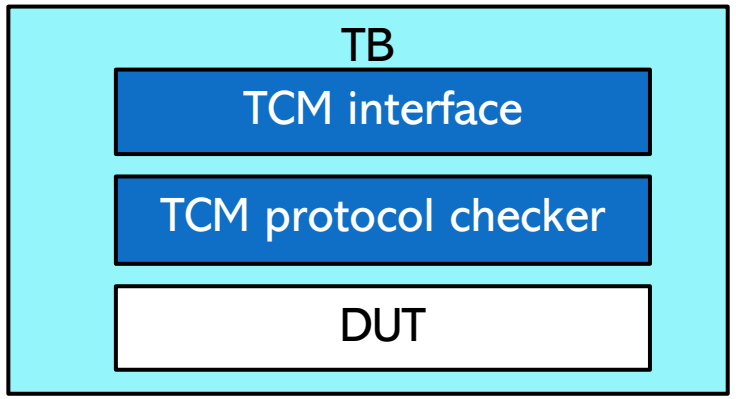
(Ref: <lvm_ahb>/tb/top.sv)

```
// User defined interface signals
   wire [31:0]                HADDR                ;
   wire [2:0]                 HBURST               ;
   wire [0:0]                 HMASTLOCK            ;
   wire [6:0]                 HPROT                ;
   ...
   // Supply clock and reset
   assign HCLK    = <your clock supply>;
   assign HRESETn = <your TCM reset signal>;
```

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

9. Include the lvm_ahb_connection.sv (mostly need modifications)

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

`include "lvm_ahb_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(.*);
```



CREATING YOUR FIRST TEST

10. Now create a new testcase and add the following API (Ref: <lvm_ahb>/tests/ lvm_ahb_03_basic_rw_test.sv)

```
// at main phase
m_ahb_env.drv_wait_output(LVM_ON);
m_ahb_env.read(
        .HADDR      (32'h1000_008C),
        .HBURST     (LVM_TCM_WRAP8),
        .HSIZE      (LVM_TCM_2B),
        .HMASTLOCK  (LVM_TCM_M_UNLOCK),

        .HRDATA     (HRDATA),
        .HRESP      (HRESP),
        .HEXOKAY    (HEXOKAY)
    );
```

```
m_ahb_env.write(
        .HADDR      (32'h1000_00F8),
        .HBURST     (LVM_TCM_INCR),
        .HSIZE      (LVM_TCM_4B),
        .HMASTLOCK  (LVM_TCM_M_UNLOCK),

        .HRESP      (HRESP),
        .HEXOKAY    (HEXOKAY)
    );
```

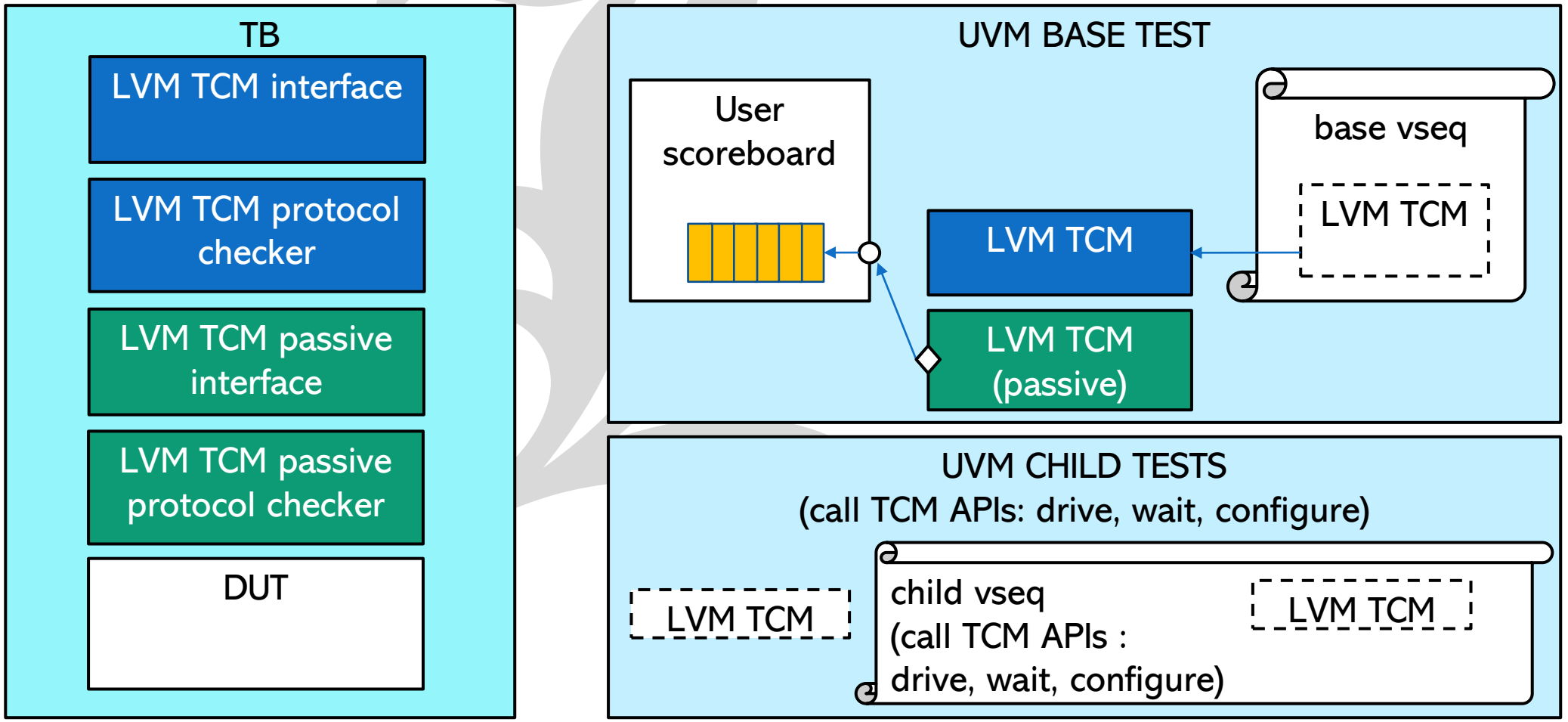
INSTALLING LVM TCM AS PASSIVE UVC

PASSIVE TCM UVC

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

```
make sim dual_TCM=on t=lvm_tcm_02_basic_test
```

EXAMPLE OF TWO LVM TCM IN THE TESTBENCH



INTEGRATION GUIDE

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

```
lvm_ahb_env `LVM_TCM_VALUES          m_lvm_ahb_passive;
```

```
// At build phase, construct the lvm_ahb UVC
function void build_phase (uvm_phase phase);
    ...
    uvm_config_db#(uvm_active_passive_enum)::set(this, "*m_lvm_ahb_passive*", "is_active", UVM_PASSIVE);
    m_lvm_ahb_passive    =     lvm_ahb_env `LVM_TCM_VALUES::type_id::create("m_lvm_ahb_passive", this);
    // Turning off master env's monitor (if you are using LVM's TCM UVC)
    uvm_config_db#(bit)::set(this, "*m_ahb_env*", "has_mon", 1'b0);
endfunction
```

CONNECTING THE PASSIVE UVC

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

```
    function void connect_phase (uvm_phase phase);
        super.connect_phase(phase);

        // Connection to your SBD / Coverage
        m_lvm_ahb_passive.port.connect(my_sbd.user_export);

endfunction
```

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

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

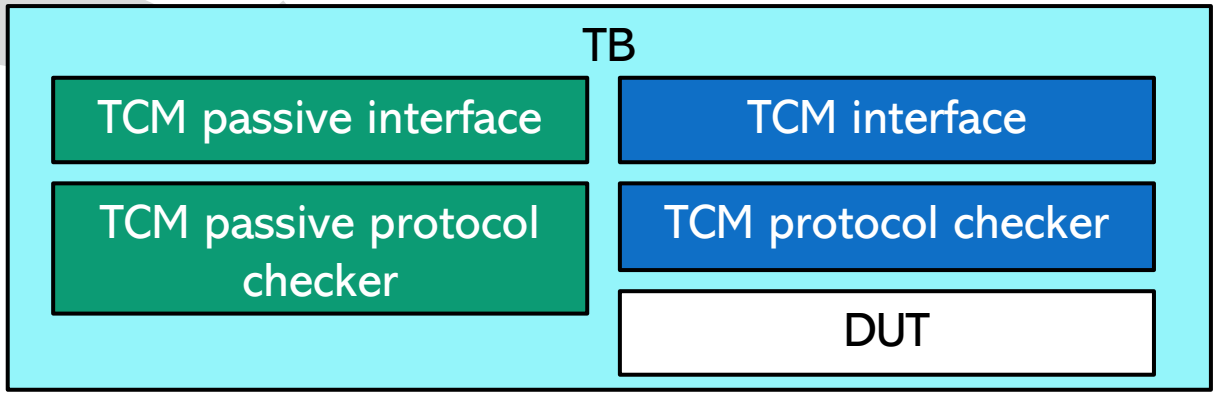
   (Ref: <lvm_ahb>/tb/top.sv)

```
// User defined interface signals
wire [31:0]                HADDR                ;
wire [2:0]                 HBURST               ;
wire [0:0]                 HMASTLOCK            ;
wire [6:0]                 HPROT                ;
...
// Supply clock and reset
assign HCLK    = <your clock supply>;
assign HRESETn = <your TCM reset signal>;
```

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

5. Include the lvm_ahb_connection2.sv (mostly need modifications)

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

`include "lvm_ahb_connection2.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 TCM traffic into a file.
  - Start checking the TCM 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.