



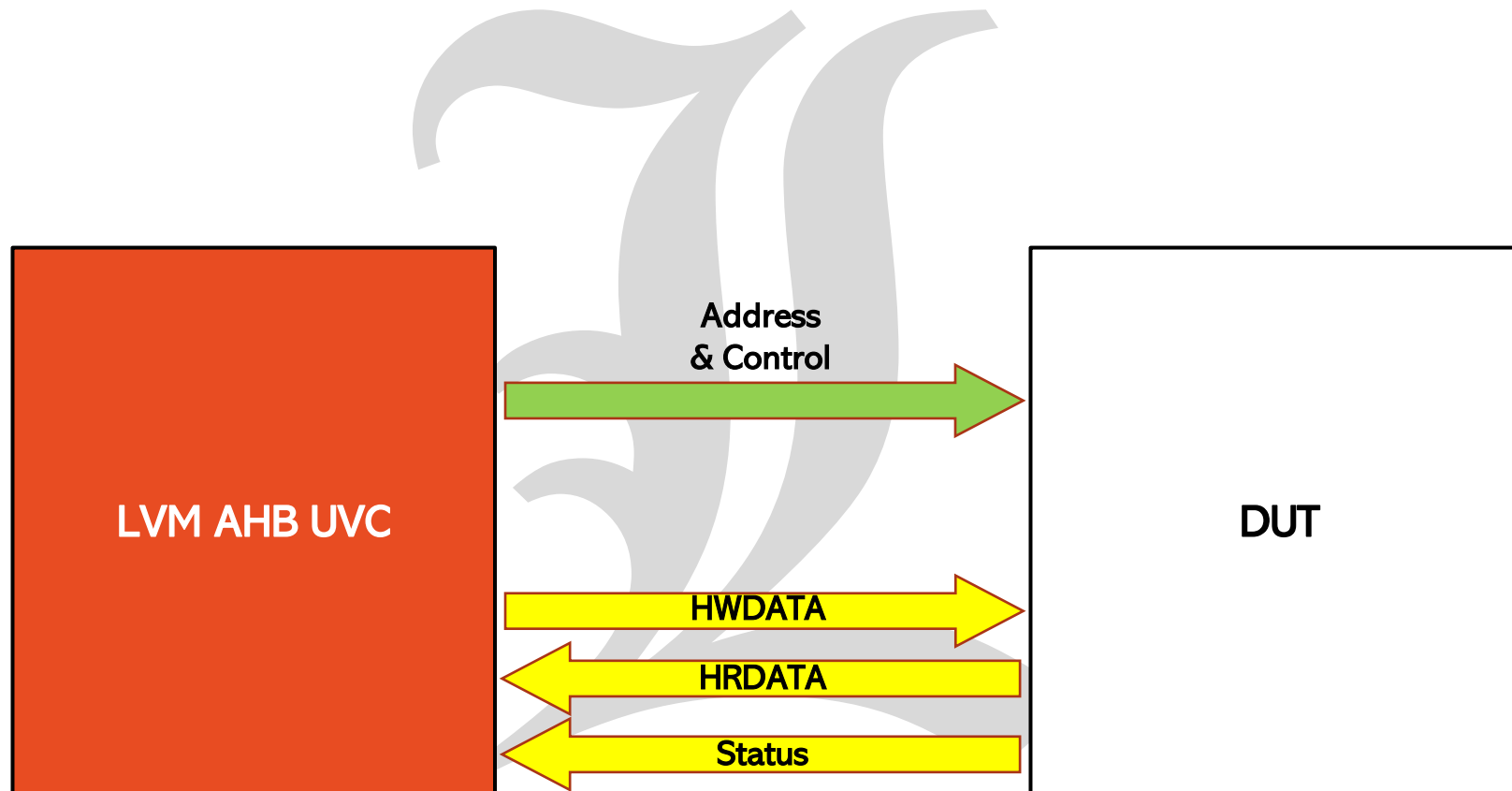
A stylized, black, gothic-style letter 'L' or 'I'.

AHB VIP

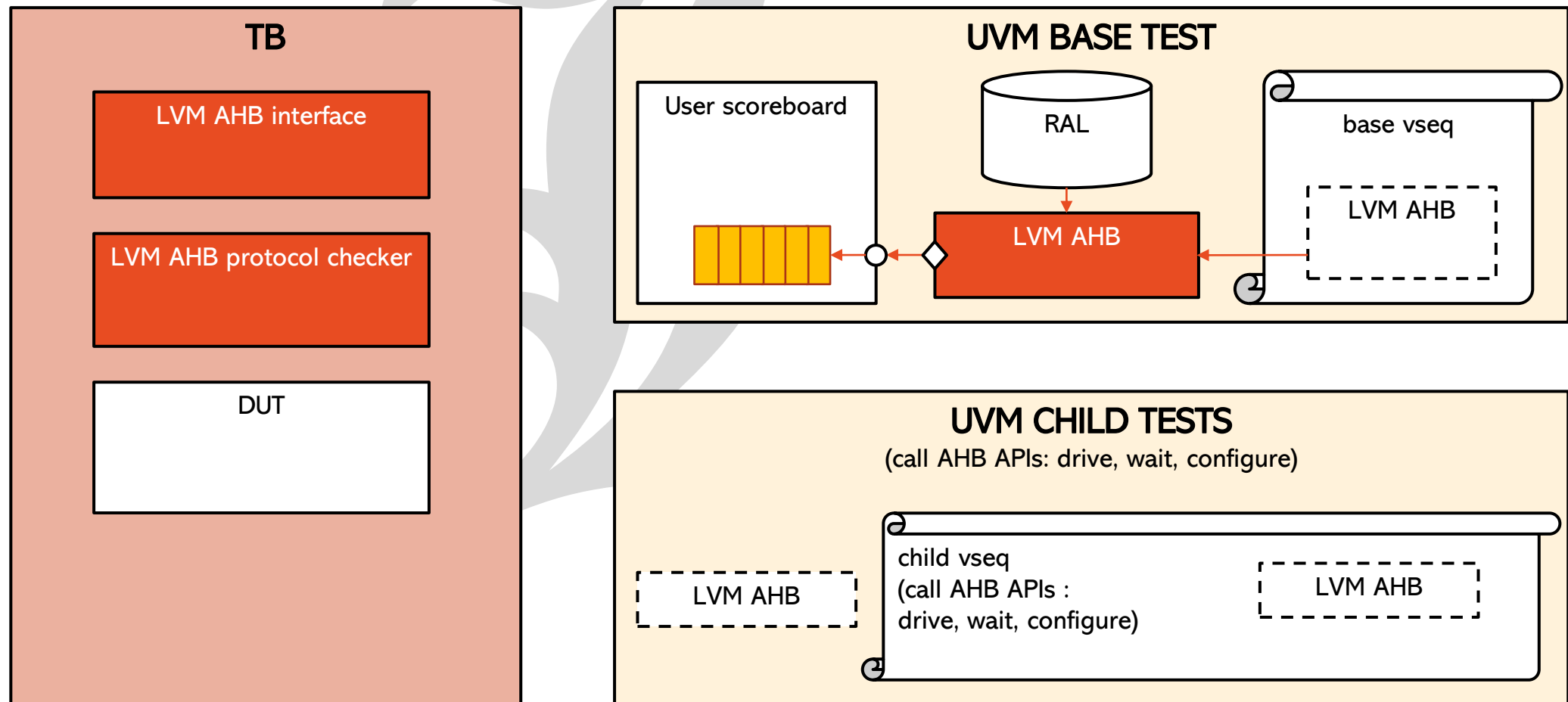
INTRODUCTION

- This is an AHB 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 AHB stimulus with ease (minimal codes).
 - To shorten the time to bring up AHB 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 AHB verification in the industry.

THE LVM AHB



EXAMPLE LVM AHB 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.

Performance Analyzer

Performance analyser

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

Reusable

Codes on lvm VIP is highly reusable.

Ease of Integration

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

Reset aware

Support reset events.

Light weight

CPU efficient UVC.

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

Ready testsuite

Provided multiple useful tests to verify AHB slaves DUT

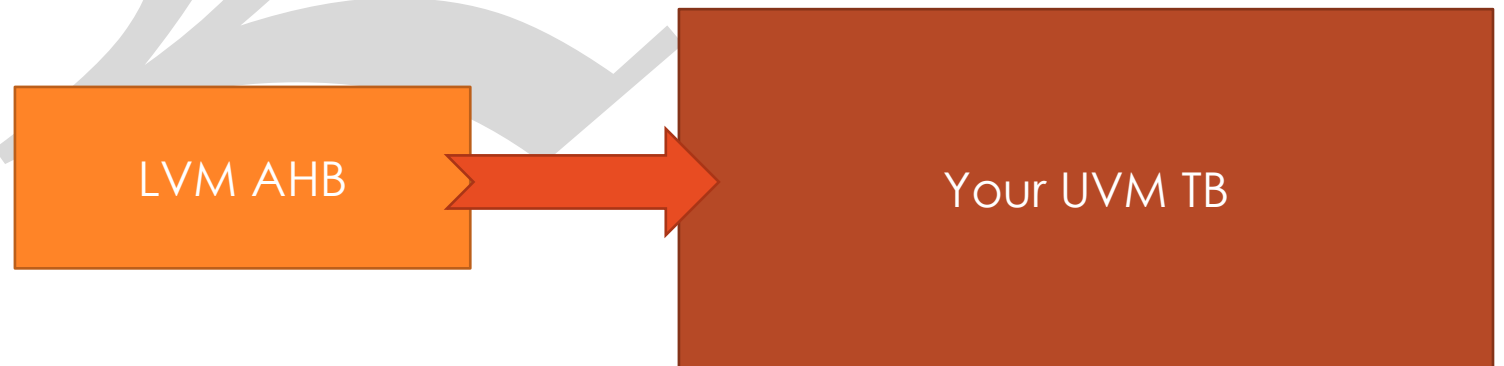
USER FRIENDLY

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, adaptor, predictor etc

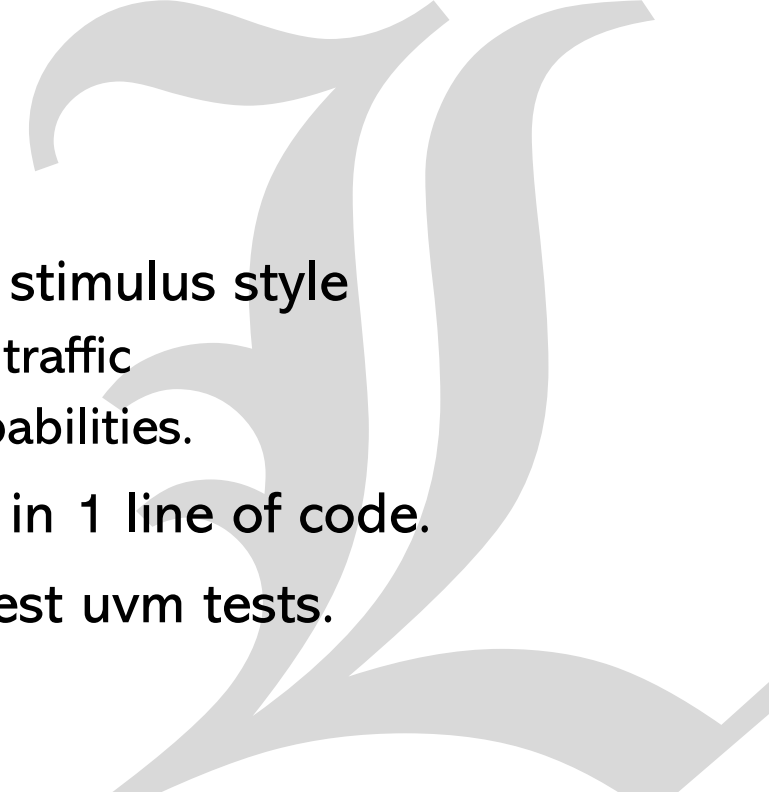


DRIVING PART



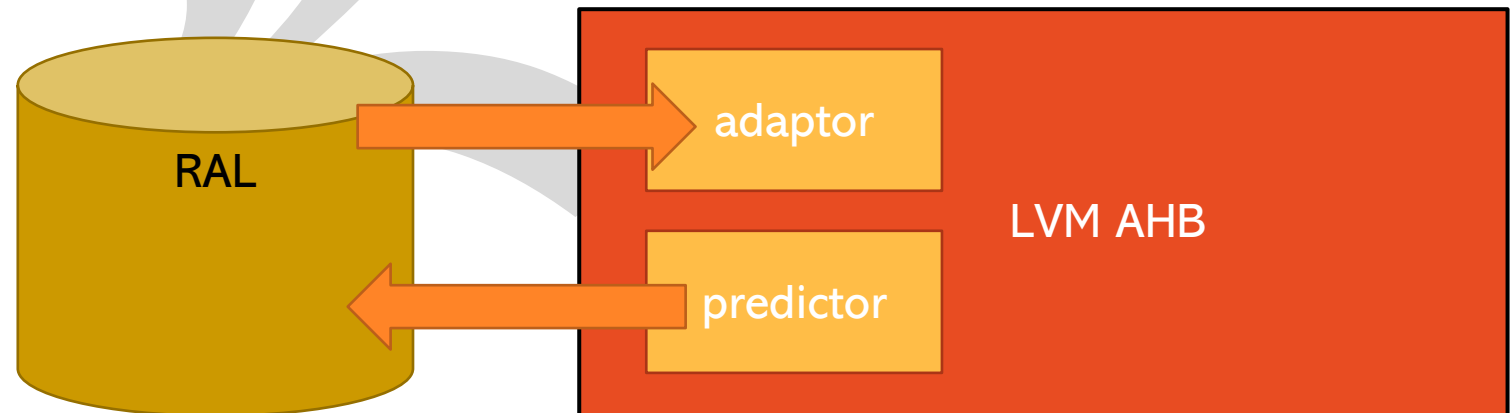


EASE OF DRIVING STIMULUS

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

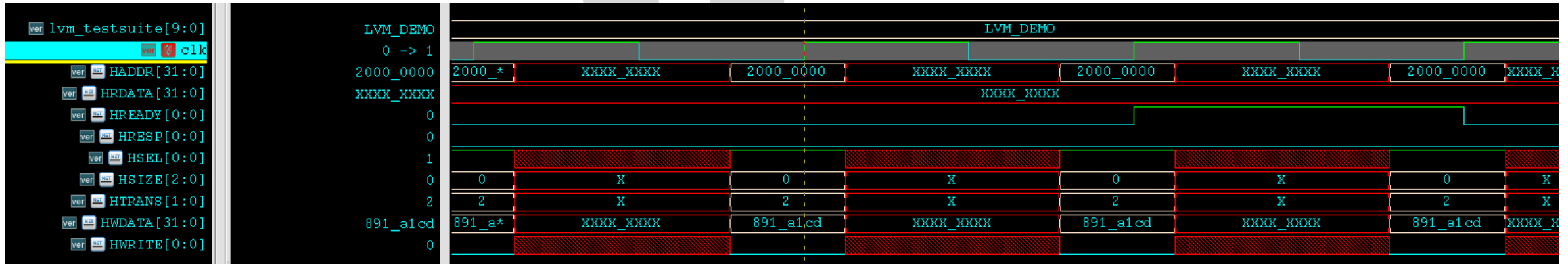
ADVANCED TECH WITH RAL

- Predictor and Adaptor are built in and 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



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_ahb_env.s_write ( .HADDR(32'h2000_0000), .hdata(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 AHB protocol checker.
- Enhanced to become UVM_ERROR when assertions fail.



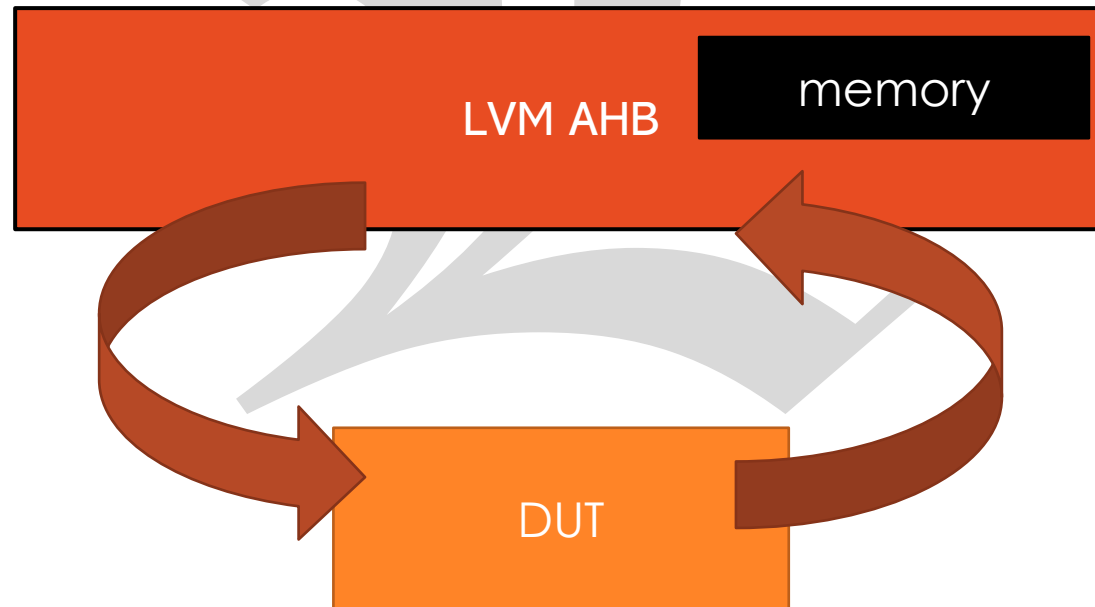
ARM AHB protocol checker
embedded (uvm)

The diagram consists of a large orange rectangle representing the 'LVM AHB' block. Inside this rectangle, at the top, is a smaller black rectangle containing the text 'ARM AHB protocol checker embedded (uvm)' in white. This visualizes the checker as a component embedded within the LVM AHB hardware block.

LVM AHB

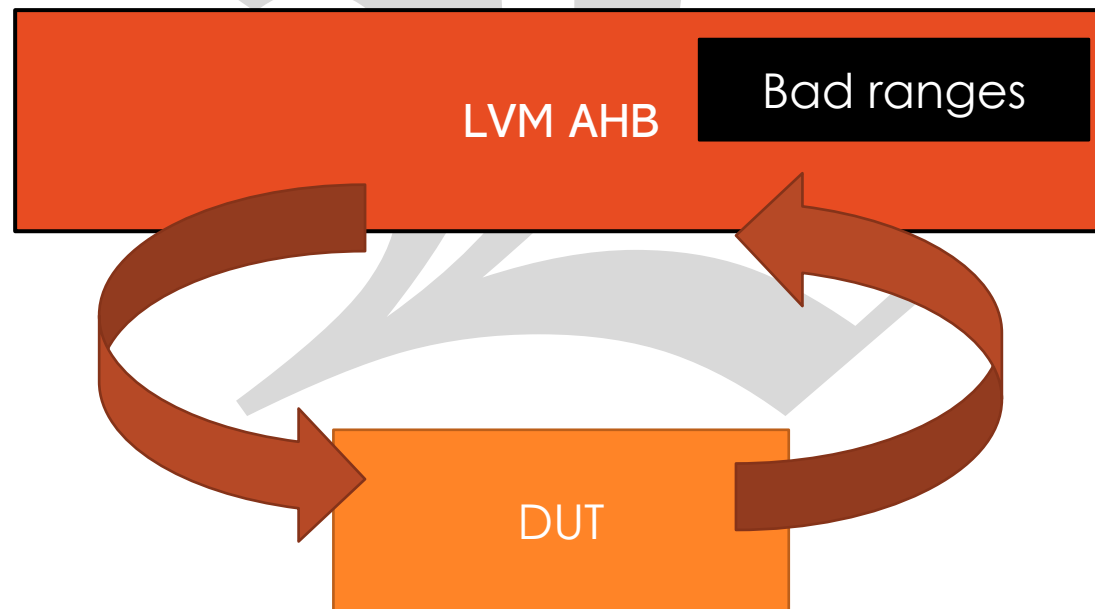
EMBEDDED MEMORY CHECKING

- 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)



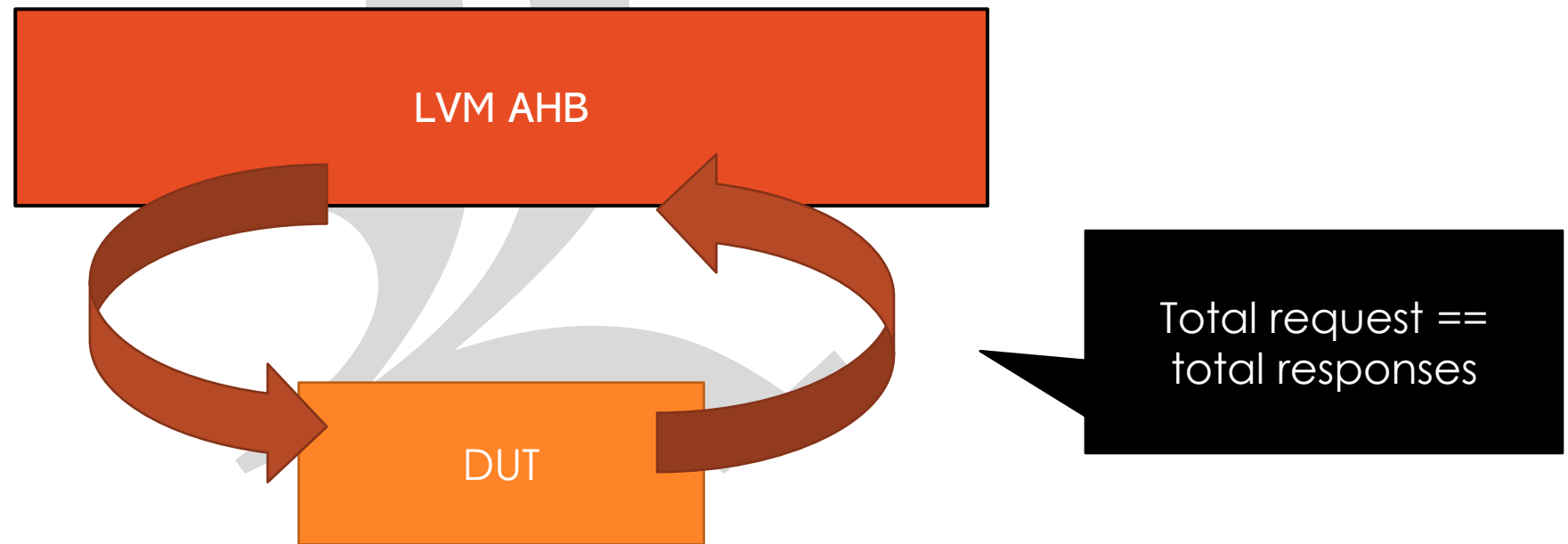
OUT OF BOUND ADDRESS CHECKING

- 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.

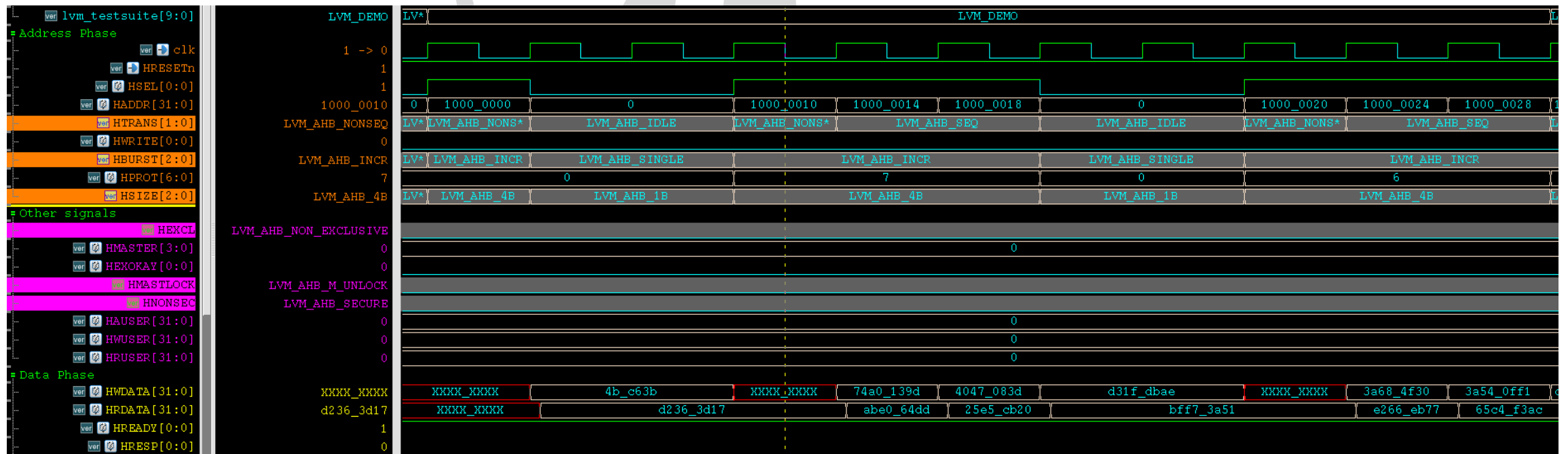


DEBUG PART



ENUM & DEBUG SIGNALS

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



COMPREHENSIVE TRACKER

- Full info for each packet in the AHB 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'hbffa73a51	32'hbffa73a51	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

- Print 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 AHB packets
 - Verify the response of every packet to meet expected value.

PERFORMANCE ANALYZER



PERFORMANCE ANALYZER

- Performance

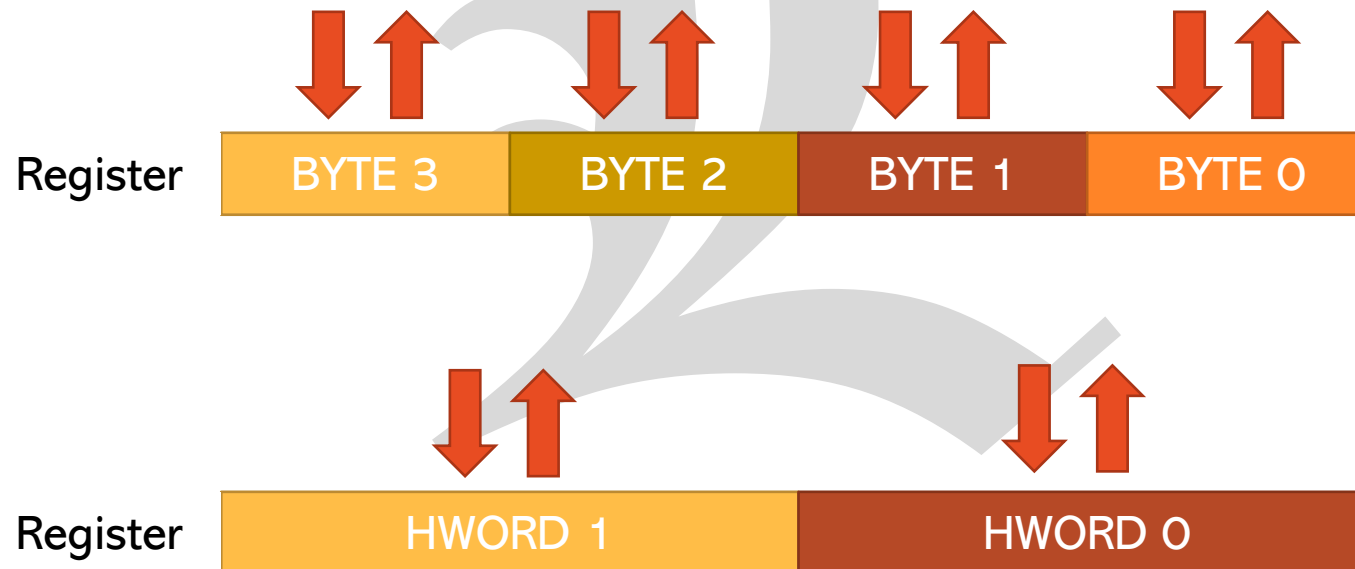
Performance Report					
Chnl	Total thru	Total waits	Average wait wait / beat	% Bus utilization	Performance
AHB	272	1350	4.963	17	WORST

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

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 AHB packet.
 - It is RAL ready.
- **High debug-ability**
 - Tracker file: AHB transactions can be traced in the log file.
 - Interface provides some debug signals.
- **Strong checkers**
 - Equipped with industry standard protocol checks on the AHB 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_AHB_1B           = 0,
    LVM_AHB_2B           = 1,
    LVM_AHB_4B           = 2,
    LVM_AHB_8B           = 3,
    LVM_AHB_16B          = 4,
    LVM_AHB_32B          = 5,
    LVM_AHB_64B          = 6,
    LVM_AHB_128B         = 7
} lvm_ahb_hsize_em;
```

```
typedef enum logic {
    LVM_AHB_READ         = 0,
    LVM_AHB_WRITE        = 1
} lvm_ahb_op_em;
```

```
typedef enum bit {
    LVM_AHB_OK           = 0,
    LVM_AHB_HRESP_ERR    = 1
} lvm_ahb_hresp_em;
```

```
typedef enum logic [2:0] {
    LVM_AHB_SINGLE       = 0,
    LVM_AHB_INCR          = 1,
    LVM_AHB_WRAP4        = 2,
    LVM_AHB_INCR4         = 3,
    LVM_AHB_WRAP8        = 4,
    LVM_AHB_INCR8         = 5,
    LVM_AHB_WRAP16       = 6,
    LVM_AHB_INCR16       = 7
} lvm_ahb_hburst_em;
```

AHB UVC COMPONENTS

UVC HIERARCHY PATHS

- If user instantiates the LVM AHB 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`
 - `uvm_test_top.m_ahb_env.prd`
 - `uvm_test_top.m_ahb_env.adp`

AHB 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 AHB 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.

AHB UVC COMPONENTS

- RAL adaptor (adp)
 - Ready-to-use adaptor to convert user's RAL access commands into AHB transactions.
- RAL predictor (prd)
 - Ready-to-use predictor to convert monitored AHB 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 AHB 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	<pre><env>.enter_reset; // This makes the UVC to enter reset state (auto done when HRESETn fall)</pre>
2	exit_reset	<pre><env>.exit_reset; // This makes the UVC to exit reset state (auto done when HRESETn rise)</pre>
3	deactivate_UVC	<pre><env>.deactivate_UVC; // Driver, monitor etc. stop their operation (called by enter_reset)</pre>
4	activate_UVC	<pre><env>.activate_UVC; // Driver, monitor etc. back to being operational (called by exit_reset)</pre>
5	posedge_clk	<pre><env>.posedge_clk; // Wait for next posedge of UVC clock</pre>
6	negedge_clk	<pre><env>.negedge_clk; // Wait for next negedge of UVC clock</pre>
7	recompute_clk_period	<pre><env>. recompute_clk_period; // Restart the UVC clock frequency calculation</pre>

```
// 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

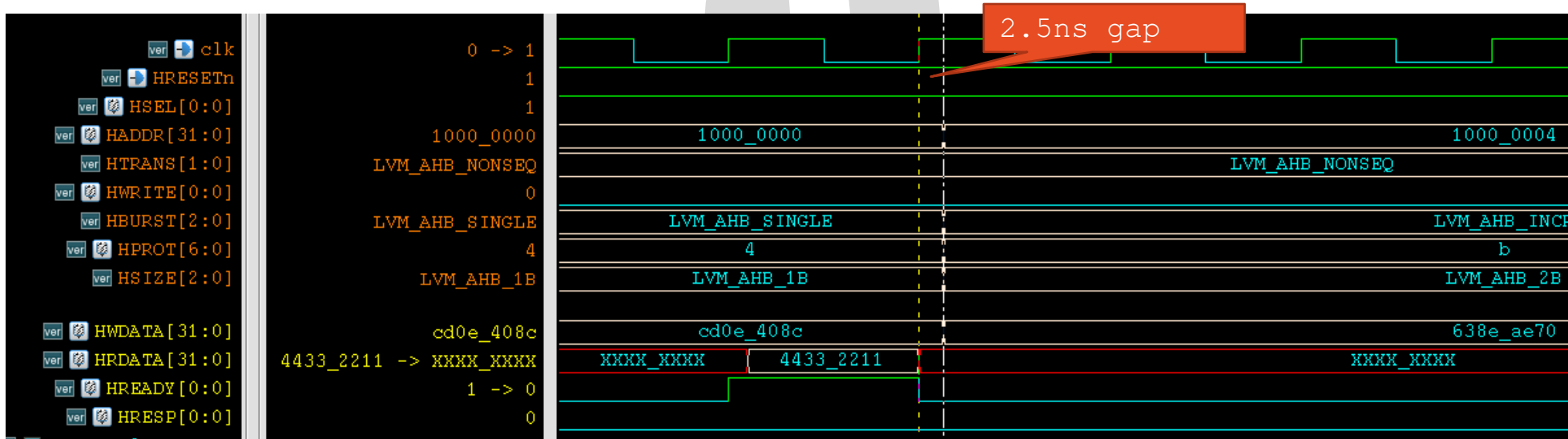
- User can configure AHB5 properties for this UVC, using the following cmd:

```
ahb_env.cfg.AHB5_Extended_Memory_Types = LVM_TRUE;  
ahb_env.cfg.AHB5_Secure_Transfers      = LVM_TRUE;  
ahb_env.cfg.AHB5_Exclusive_Transfers   = LVM_TRUE;  
ahb_env.cfg.AHB_User_Signals           = LVM_TRUE;
```

- Default it is full AHB5 protocol.
- To enter AHB LITE protocol mode, these variable can be turned OFF.

CLOCKING BLOCK CONFIGURATION

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

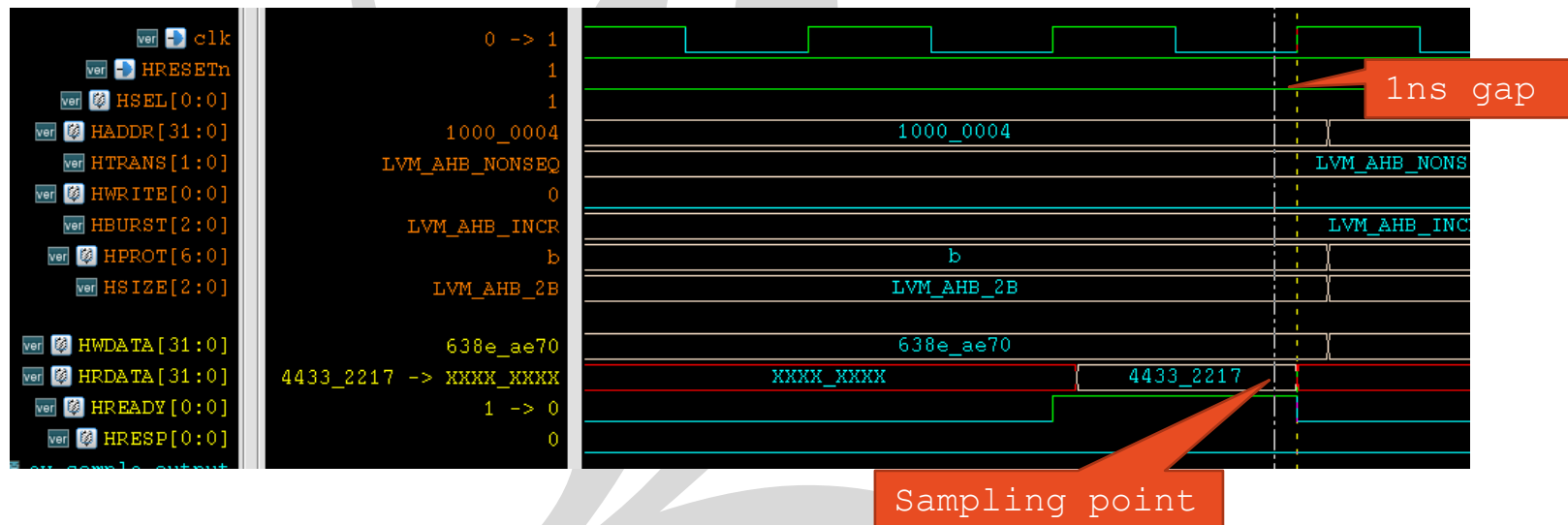


- The cmd:

```
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:

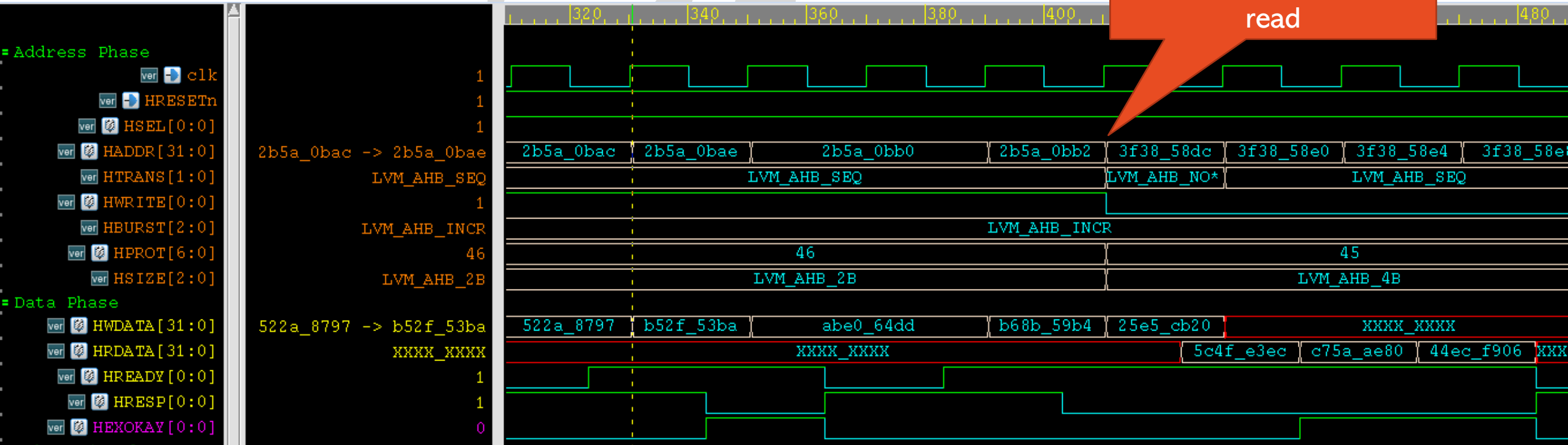
```
m_ahb_env.set_setup_time(1, "ns");
```

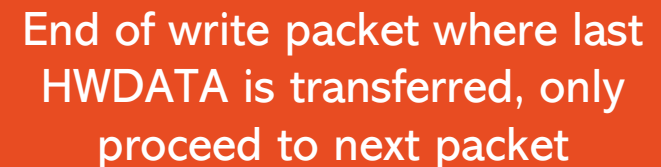
DRIVER WAIT STYLE

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

PIPELINED TRANSACTION DRV_WAIT_OUTPUT(LVM_OFF)





CARE_BOUNDARY

- This bit will define master UVC to follow the ADDR_TEST_RANGE (more details at “AHB test suites” 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 AHB 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 int is probability of 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 = 50; // 50% chance to cancel
```

LVM AHB 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       :
```

LVM AHB SLAVE CONFIGURATION

- `<slave_env>.cfg.auto_reply`
 - Default value = 1'b1
 - Slave VIP will be returning HREADY, HRESP per ever_ready, not_ready_clks, bad_hresp, rand_not_ready, inject_error.
 - when `<slave env>.cfg.auto_reply == 1'b0`, slave VIP will not be returning HREADY, HRESP until user give value via API
`<slave env>.drive_HREADY(<user value: LVM_AHB_READY / LVM_AHB_NOT_RDY>);`

HIGH DEBUG-ABILITY

HIGH DEBUG-ABILITY UVC

- To speed up the debug process, the UVC provides its user with high visibility of AHB bus traffic.
- The simulation with monitor = ON & tracker = ON will enable the tracker file dump
 - It is a file that contains all the AHB 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 AHB provides high debugability by preparing log file for all AHB 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>;`

Packet serial
number (++1 for
Read / Write)

Write serial number (++1 for write)

Timestamp

[Packet 159: WRITE 131]

4790.000 ns

CMD=LVM_AHB_WRITE HBURST=LVM_AHB_INCR HMASTLOCK=LVM_AHB_M_UNLOCK HPROT=7'h00 HSIZE=LVM_AHB_2B

Effective bits

	HTRANS	HADDR	EFFECTIVE	HWDATA	HRESP	IMPACT
[Beat 0 31:16]	LVM_AHB_NONSEQ	32'h10000016	16'h4fa6	32'h4fa6xxxx	LVM_AHB_OK	10000017<-4f 10000016<-a6
[Beat 1 15: 0]	LVM_AHB_SEQ	32'h10000018	16'ha01	32'hxxxx0a01	LVM_AHB_OK	10000019<-0a 10000018<-01

Beat
count

Effective data

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

Effective
addresses

Impact for this beat

TRACKER LOG

- For read packet:

Packet serial
number
(++1 for Read
/ Write)

Read serial number (++1 for Read)

Timestamp

Details of Read

[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

Effective bits

[Beat 0 31: 0]

HTRANS

LVM_AHB_NONSEQ

HADDR

32'h1000006c

EFFECTIVE

32'h41df7028

HRDATA

32'h41df7028

HRESP

LVM_AHB_OK

SOURCE

1000006f->41 1000006e->df 1000006d->70 1000006c->28

Effective data

Beat
count

HPROT[3]=0

LVM_AHB_UNMODIFIABLE

HPROT[2]=0

LVM_AHB_NON_BUFFERABLE

HPROT[1]=0

LVM_AHB_UNPRIVILEGED

HPROT[0]=0

LVM_AHB_INSTRUCTION

Effective address

Source for this beat

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 AHB 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

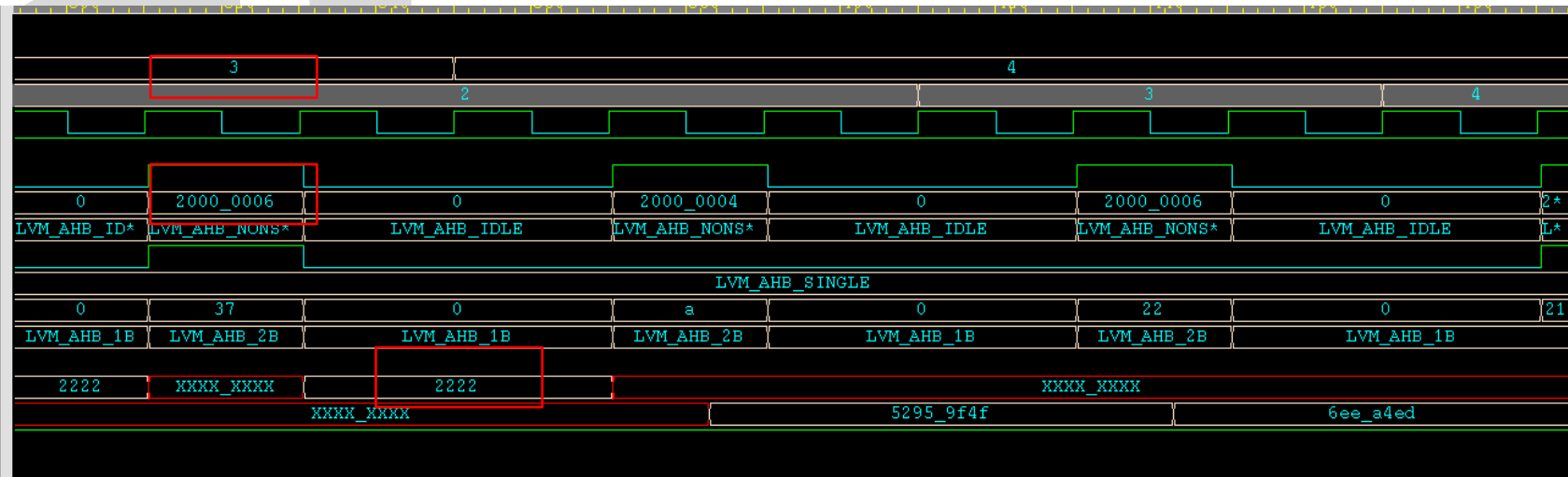
CMD=LVM_AHB_WRITE HBURST=LVM_AHB_SINGLE HMASTLOCK=LVM_AHB_M_UNLOCK HPROT=7'h37 HSIZE=LVM_AHB_2B HNONSEC=1'h1 HEXCL=1'h1 HMASTER=4'h1 HAUSER=32'h40761955 HWUSER=32'hab29dded HRUSER=32'hb6432125

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

```

= Address Phase
ver 0 /top/m_lvm_ahb_if/write_counter[31:0] 2
ver 0 /top/m_lvm_ahb_if/read_counter[31:0] 1
ver 0 /top/m_lvm_ahb_if/clk 1
ver 0 /top/m_lvm_ahb_if/HRESETn 1
ver 0 /top/m_lvm_ahb_if/HSEL[0:0] 0 -> 1
ver 0 /top/m_lvm_ahb_if/HADDR[31:0] 0 -> 2000_0000
ver 0 /top/m_lvm_ahb_if/HTRANS[1:0] IDLE -> LVM_AHB_NONSEQ
ver 0 /top/m_lvm_ahb_if/HWRITE[0:0] 0
ver 0 /top/m_lvm_ahb_if/HBURST[2:0] LVM_AHB_SINGLE
ver 0 /top/m_lvm_ahb_if/HPROT[6:0] 0 -> 39
ver 0 /top/m_lvm_ahb_if/HSIZE[2:0] LVM_AHB_1B -> LVM_AHB_4B

= Data Phase
ver 0 /top/m_lvm_ahb_if/HWDATA[31:0] 1111_1111 -> XXXX_XXXX
ver 0 /top/m_lvm_ahb_if/HRDATA[31:0] XXXX_XXXX
ver 0 /top/m_lvm_ahb_if/HREADY[0:0] 1
ver 0 /top/m_lvm_ahb_if/HRESP[0:0] 0
  
```



READ COUNTER

[Packet 6 **READ 3**] 470.000 ns

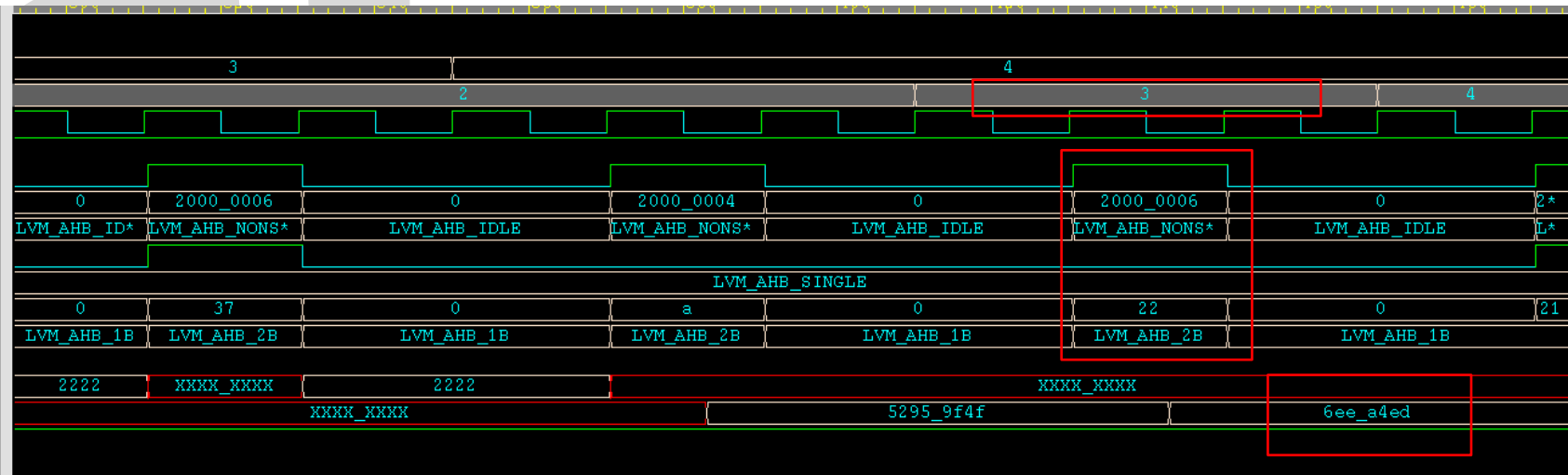
CMD= LVM_AHB_READ HBURST=LVM_AHB_SINGLE HMASTLOCK=LVM_AHB_M_UNLOCK HPROT=7'h22 HSIZE=LVM_AHB_2B HNONSEC=1'h0 HEXCL=1'h1 HMASTER=4'he HAUSER=32'hf371d044 HWUSER=32'hca6bbad3 HRUSER=32'h3156dab7

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

```

= Address Phase
ver 0 /top/m_lvm_ahb_if/write_counter[31:0] 2
ver 0 /top/m_lvm_ahb_if/read_counter[31:0] 1
ver 0 /top/m_lvm_ahb_if/clk 1
ver 0 /top/m_lvm_ahb_if/HRESETn 1
ver 0 /top/m_lvm_ahb_if/HSEL[0:0] 0 -> 1
ver 0 /top/m_lvm_ahb_if/HADDR[31:0] 0 -> 2000_0000
ver 0 /top/m_lvm_ahb_if/HTRANS[1:0] IDLE -> LVM_AHB_NONSEQ
ver 0 /top/m_lvm_ahb_if/HWRITE[0:0] 0
ver 0 /top/m_lvm_ahb_if/HBURST[2:0] LVM_AHB_SINGLE
ver 0 /top/m_lvm_ahb_if/HPROT[6:0] 0 -> 39
ver 0 /top/m_lvm_ahb_if/HSIZE[2:0] LVM_AHB_1B -> LVM_AHB_4B

= Data Phase
ver 0 /top/m_lvm_ahb_if/HWDATA[31:0] 1111_1111 -> XXXX_XXXX
ver 0 /top/m_lvm_ahb_if/HRDATA[31:0] XXXX_XXXX
ver 0 /top/m_lvm_ahb_if/HREADY[0:0] 1
ver 0 /top/m_lvm_ahb_if/HRESP[0:0] 0
  
```



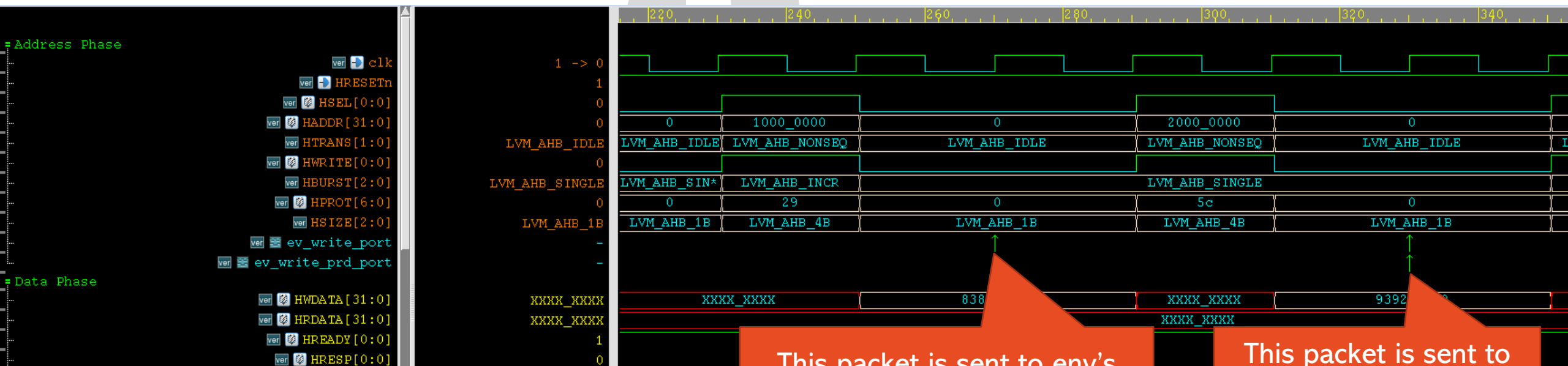
PORT WRITE EVENTS

- 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 AHB read packet is written into env's TLM port
2	ev_read_prd_port	Marks the time where AHB read packet is sent to RAL predictor
3	ev_write_port	Marks the time where AHB write packet is written into env's TLM port
4	ev_write_prd_port	Marks the time where AHB write packet is sent to RAL predictor

PORT WRITE EVENTS

- Events for write packets

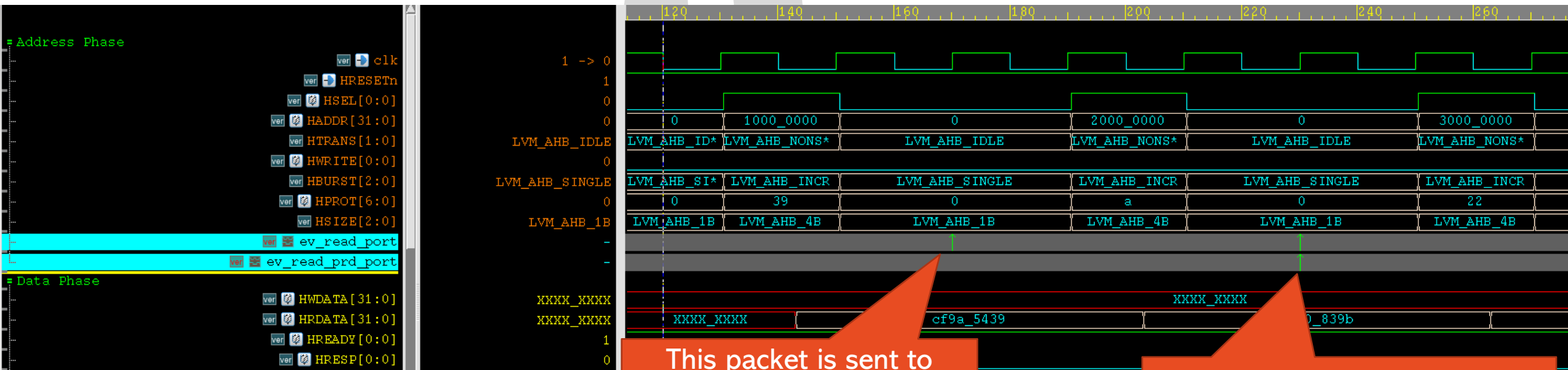


This packet is sent to env's port only, because the address is outside the range of RAL

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

PORT WRITE EVENTS

- Events for read packets



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

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

WORKING WITH RAL

RAL READY AHB UVC

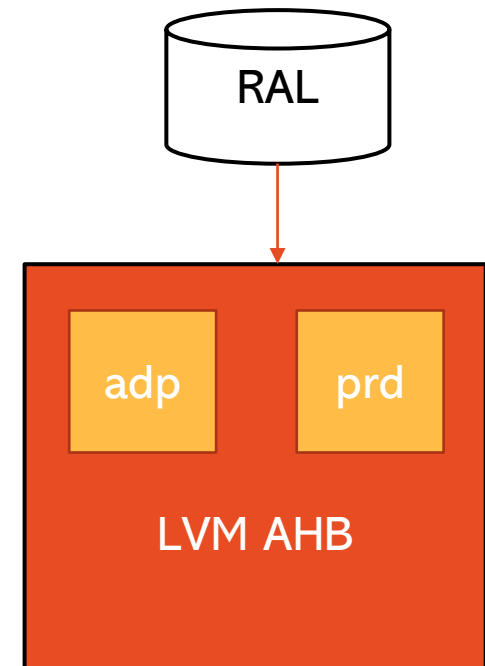
- The LVM's AHB is ready to work with RAL.
- It has built-in RAL adaptor and RAL predictor.
- To connect the AHB UVC with RAL, the following is to be done at connect phase:

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

if(m_ahb_env.cfg.has_prd)
    m_ahb_env.prp.map      = urm.default_map;

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

Ref: tests/lvm_ahb_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("HRESP=%h", m_ahb_env.cfg.ral_HRESP[0]), UVM_NONE) // retrieving HRESP

urm.<regB>.write(status, mydata);
`uvm_info(msg_tag, $sformatf("HRESP=%h", m_ahb_env.cfg.ral_HRESP[0]), UVM_NONE) // retrieving HRESP
// 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_ahb_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 READY AHB UVC

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

No	Variable	API command
1	HMASTLOCK	<code><env>.cfg.ral_HMASTLOCK = <value>;</code>
2	HPROT	<code><env>.cfg.ral_HPROT = <value>;</code>
3	HNONSEC	<code><env>.cfg.ral_HNONSEC = <value>;</code>
4	HEXCL	<code><env>.cfg.ral_HEXCL = <value>;</code>
5	HMASTER	<code><env>.cfg.ral_HMASTER = <value>;</code>
6	HAUSER	<code><env>.cfg.ral_HAUSER = <value>;</code>
7	HWUSER	<code><env>.cfg.ral_HWUSER = <value>;</code>

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_ahb_env.cfg.add_RAL_ADDR_RANGE (  
    .start_addr    (<ral_min_addr>),  
    .end_addr      (<ral_max_addr>),  
    .expected_resp ({LVM_AHB_OK})  
);
```

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_ahb_env*", "has_adp", 1'b0);
```

- To turn OFF predictor:

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

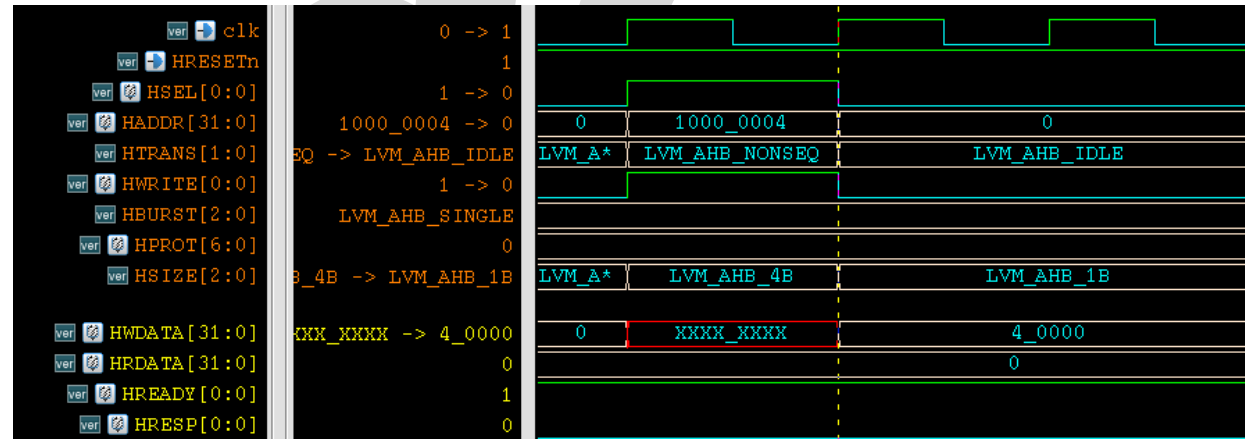
Ref: tests/lvm_ahb_30_no_component_test.sv

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

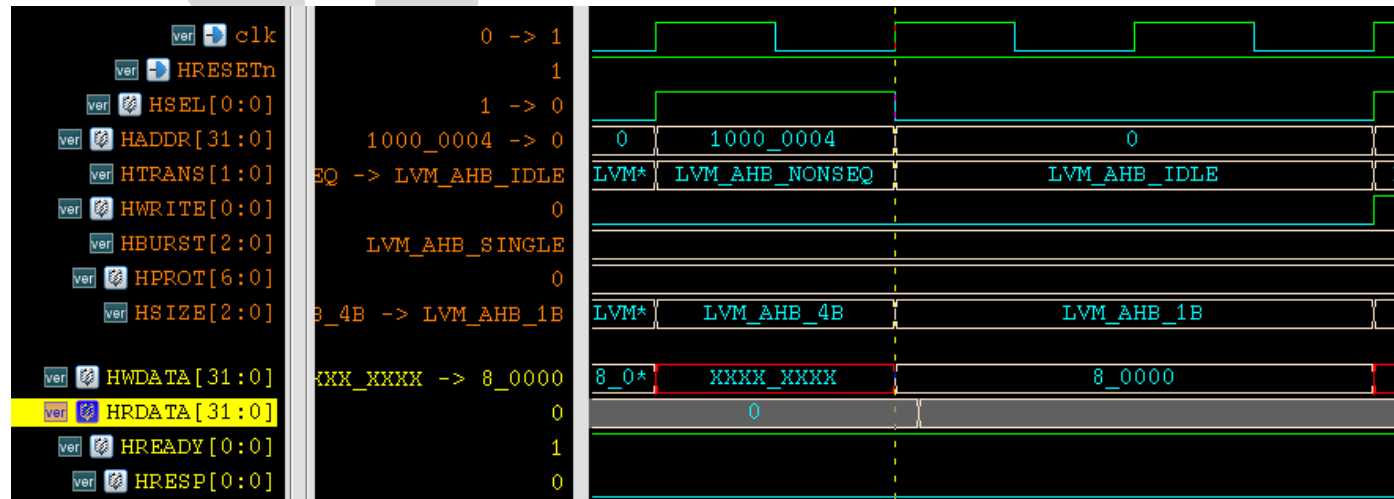
FULL REG WRITE ACCESS

- This UVC supports full register write access as shown previously:
 - `urm.<regB>.write(status, mydata);`
- It will do the hwrite.
- During write, HSIZE=4B 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 AHB 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 AHB packet with corresponding HSIZE.
- 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));
```

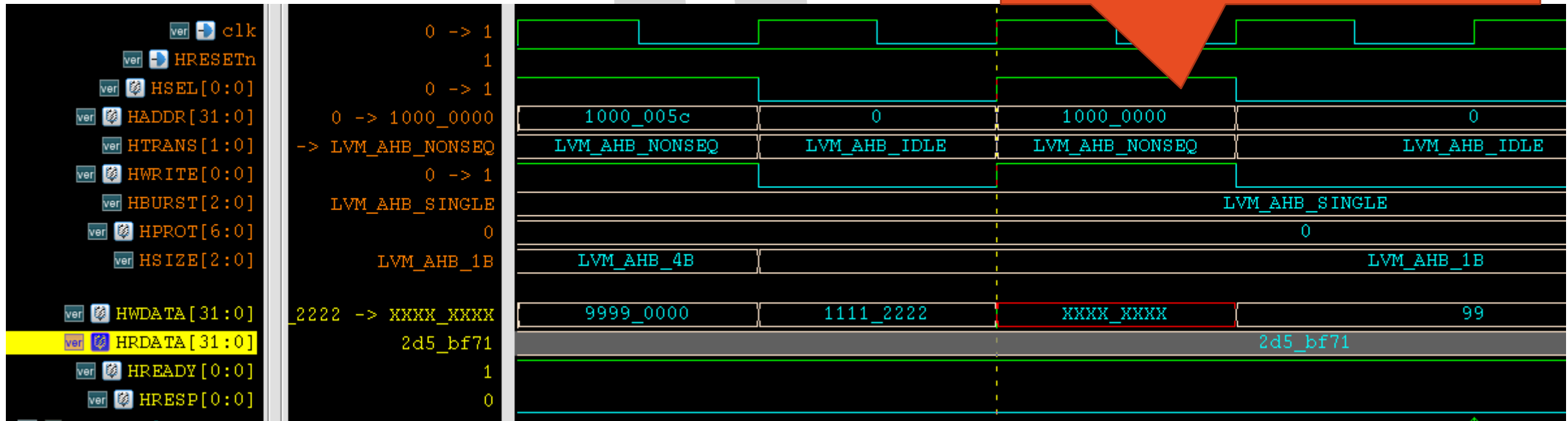
The size must be 8 bits

Must be individually accessible

eg.field.write (normally is r
1B as below:

- eg.field.write (normally is r
1B as below:

```
urm.control_0.B0 .write
(status,32'h99);
```



PARTIAL REG WRITE ACCESS

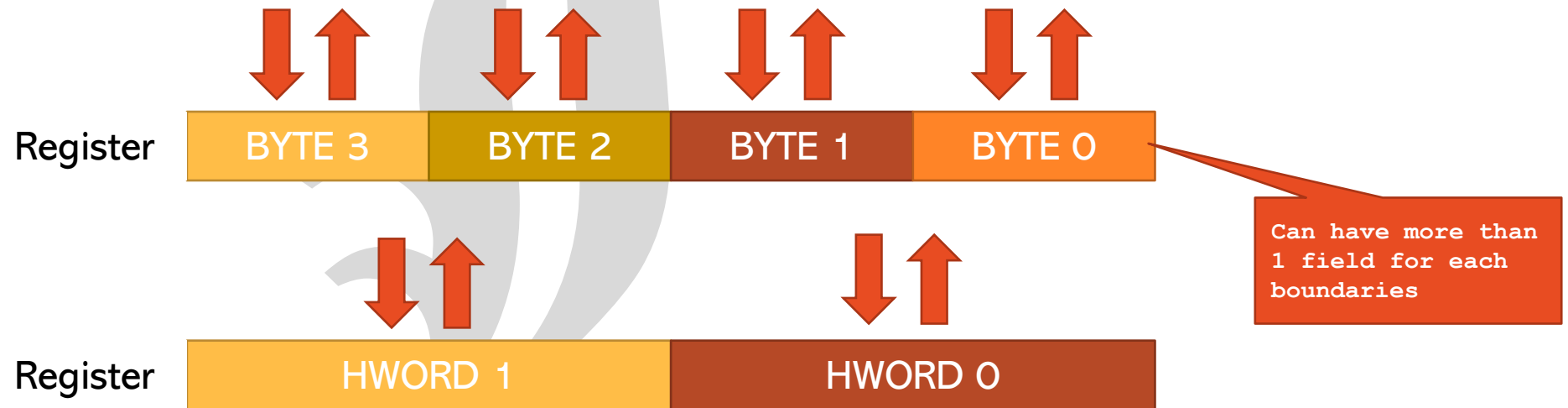
- This means, user can easily access various byte boundary fields, for example:
 - 8 bits, can be at byte 0, 1, 2 or 3 (provided fields are 8 bits wide and fill up whole byte)
 - 16 bits, can be byte 1_0, byte 2_1, or byte 3_2 (provided fields are 16 bits wide and fill up whole hword)
- However, if the 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_OC.B0.set_compare(UVM_NO_CHECK);

m_ahb_env.ral_partial_access.map = urm.default_map; // connect to your desired map to verify
m_ahb_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
[lvm_ral_partial]      LVM_HWORD 0  32'h10000080 [ 0: 0] urm.control_80.b0
[lvm_ral_partial]      LVM_HWORD 0  32'h10000080 [ 1: 1] urm.control_80.b1
[lvm_ral_partial]      LVM_HWORD 0  32'h10000080 [ 2: 2] urm.control_80.b2
[lvm_ral_partial]      LVM_HWORD 0  32'h10000080 [ 3: 3] urm.control_80.b3
[lvm_ral_partial]      LVM_HWORD 0  32'h10000080 [15: 4] urm.control_80.b15_4
[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
```

ACCESS	RESET	DESIRED	MIRRORED	VOLATILE
RW	1'h0	1'h1	1'h1	0
RW	1'h0	1'h1	1'h1	0
RW	1'h0	1'h0	1'h0	0
RW	1'h0	1'h0	1'h0	0
RW	12'h0	12'hc1a	12'hc1a	0

PARTIAL REGISTER ACCESS VERIFICATION

```
[lvm_ral_partial] ----< Verifying BYTE 3 access >----
```

						ACCESS	RESET	DESIRED	MIRRORED	VOLATILE
[lvm_ral_partial]	LVM_BYTE 3	ADDR	VECTOR	FIELD_PATH						
[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	WORD 1	WORD 0
Register 1	WORD 1	WORD 0
Register 2	WORD 1	WORD 0
Register 3	WORD 1	WORD 0
Register 4	WORD 1	WORD 0
Register 5	WORD 1	WORD 0

Register 0	WORD 1	WORD 0
Register 1	WORD 1	WORD 0
Register 2	WORD 1	WORD 0
Register 3	WORD 1	WORD 0
Register 4	WORD 1	WORD 0
Register 5	WORD 1	WORD 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.

Register 0	WORD
Register 1	WORD
Register 2	WORD
Register 3	WORD
Register 4	WORD
Register 5	WORD

Register 0	WORD
Register 1	WORD
Register 2	WORD
Register 3	WORD
Register 4	WORD
Register 5	WORD

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_ahb_env.ral_burst_access.support_partial = 1;           // 1 means support 1B, 2B accesses (smaller than bus size access)
m_ahb_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_ahb_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_ahb_env.ral_burst_access.bytes_per_beat   = 'z';          // Randomize bytes per beat, user can put 1/2/4/8
m_ahb_env.ral_burst_access.start(null);
```

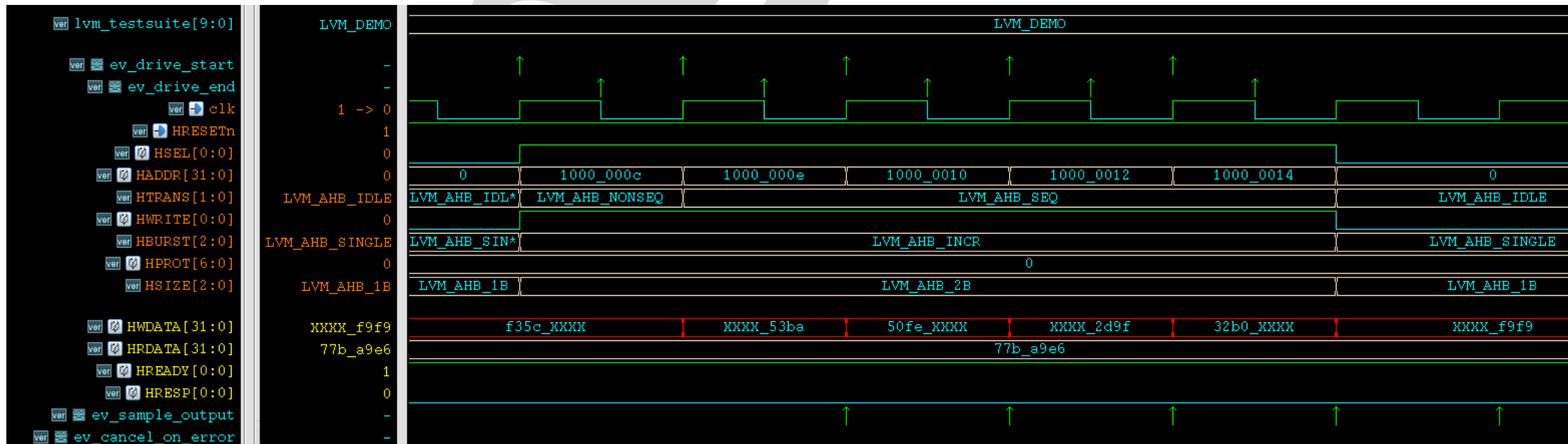
BURST REGISTER ACCESS VERIFICATION

```
[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] 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] 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] WRITE Beat=0 addr='h10000080 size=LVM_HWORD data='hac55
[lvm_ral_burst] WRITE Beat=0 addr='h10000082 size=LVM_HWORD data='hea04
[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 AHB PACKET TO RAL TARGET



BACKGROUND AHB READ DATA CHECK

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

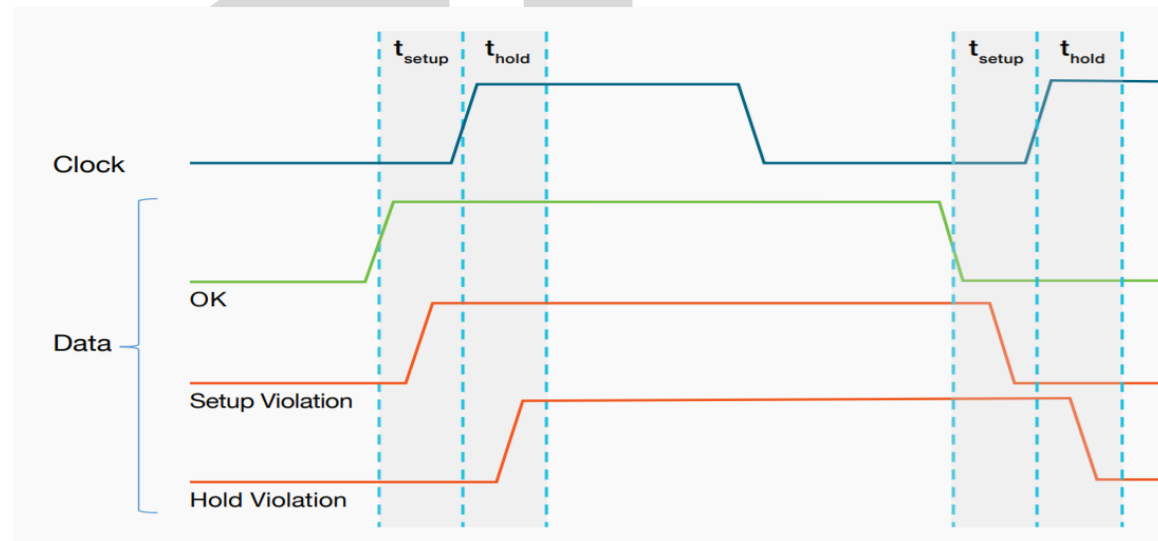
- User can retrieve the current stored data from memory database using the following API as well.

```
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.

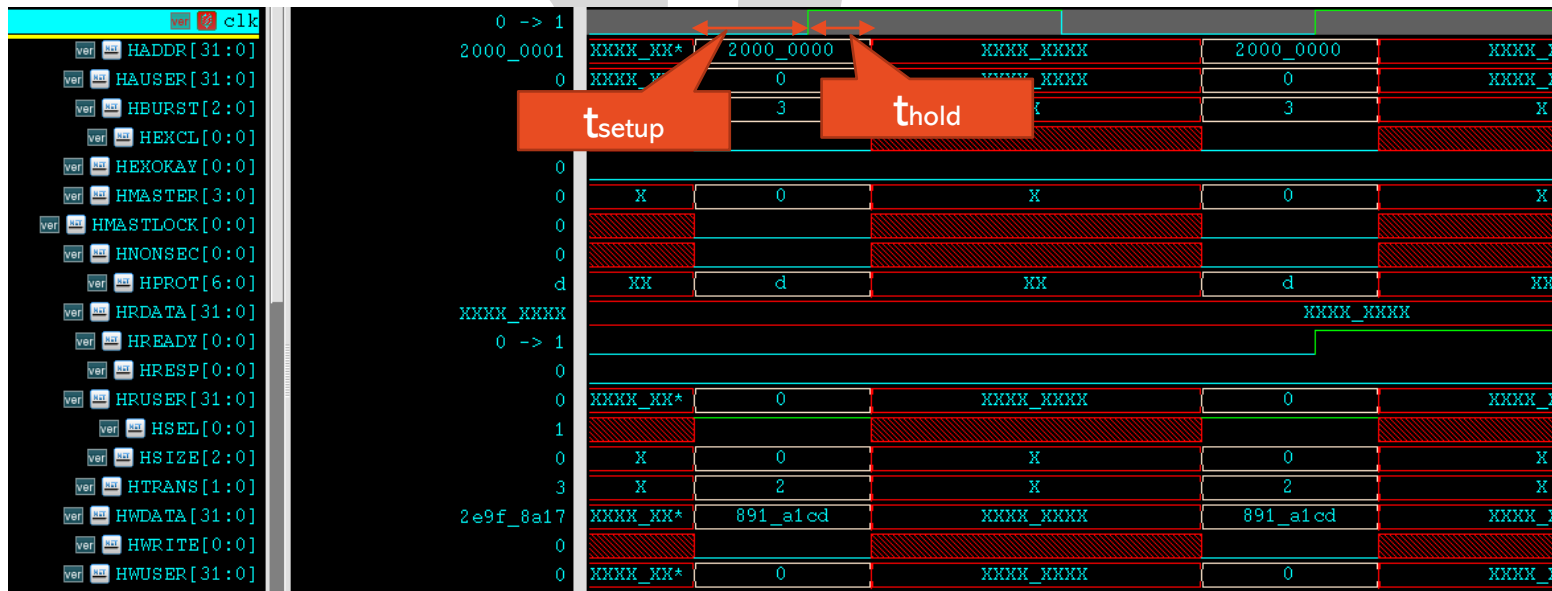


Ref:

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

X INJECTION

- LVM AHB 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_AHB_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
10. write_32, read_32, write_64, read_64,
write_128, read_128, write_256,
read_256, write_512, read_512,
write_1024, read_1024

1. READ

- UVC sends an AHB 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_AHB_WRAP4),
    .HSIZE      (LVM_AHB_2B),
    .HMASTLOCK  (LVM_AHB_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_AHB_2B),
    .HMASTLOCK(LVM_AHB_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, *_32 means word, and also *_64, *_128, *_256, *_512, *_1024
 - simpler hwdatas entry (no need manual alignment, api will do the job for you)
 - simpler API output

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

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

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

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


SUMMARY: IMPACT OF DRV_WAIT_OUTPUT

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

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) ,  
    .hwddata  (32'h11111111) ,  
    .hresp    (hresp) ,  
    .hexokay  (hexokay)  
);
```

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

API PART 2: PACKET WAITING

115

1. `wait_all_done`

1. WAIT_ALL_DONE

- This API waits for all the AHB 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 AHB 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

- AHB 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 AHB 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

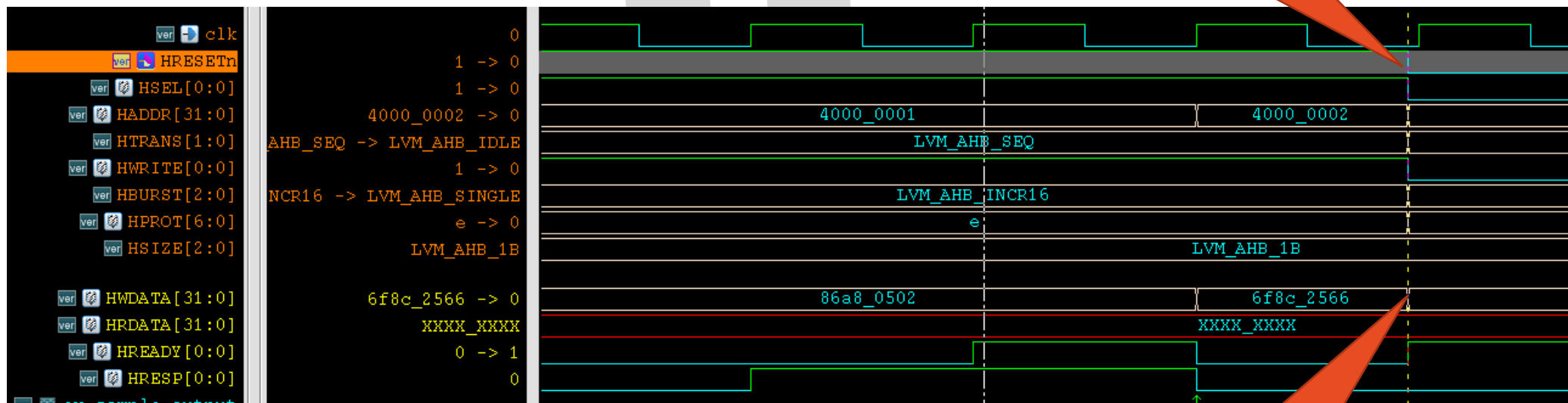
```
-----
+-----+
| HPROT[3]=1      HPROT[2]=1      HPROT[1]=0      HPROT[0]=1      |
| LVM_AHB_MODIFIABLE  LVM_AHB_BUFFERABLE  LVM_AHB_UNPRIVILEGED  LVM_AHB_DATA_ACCESS  |
+-----+
```

- 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.



Reset happens

Signals go idle

SEQUENCE ITEM & SEQUENCE WRITING

SEQ ITEM VARIABLE NAMES

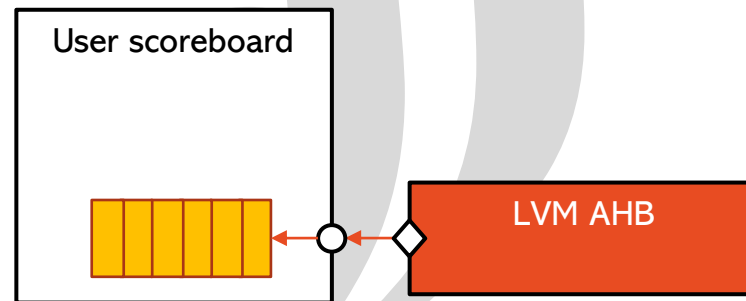
- LVM AHB sequence item consists of standard AHB bus signals.
- Red colour font means they are array.

HADDR
HBURST
HMASTLOCK
HPROT
HSIZE
HNONSEC
HEXCL
HMASTER
HTRANS
HWRITE

HSEL
HAUSER
HWUSER
HRUSER
HWDATA []
HRDATA []
HREADY
HRESP []
HEXOKAY[]
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 AHB 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

AHB TESTSUITES

AHB 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 AHB 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_01_ral_access_test	RAL access examples with drv_wait_output = LVM_ON and LVM_OFF.
lvm_ahb_02_single_accesses_test	s_write, s_read, write_16, read_16, write_8, read_8 usage examples.
lvm_ahb_02a_ral_partial_access_test	How to start the build in API to verify ral for partial access
lvm_ahb_02b_ral_access_cfg_test.sv	RAL access examples with runtime configurable signal values like: HPROT, HMASTER etc. Also demonstrate predictor working for various packets.
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	AHB 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	AHB 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:

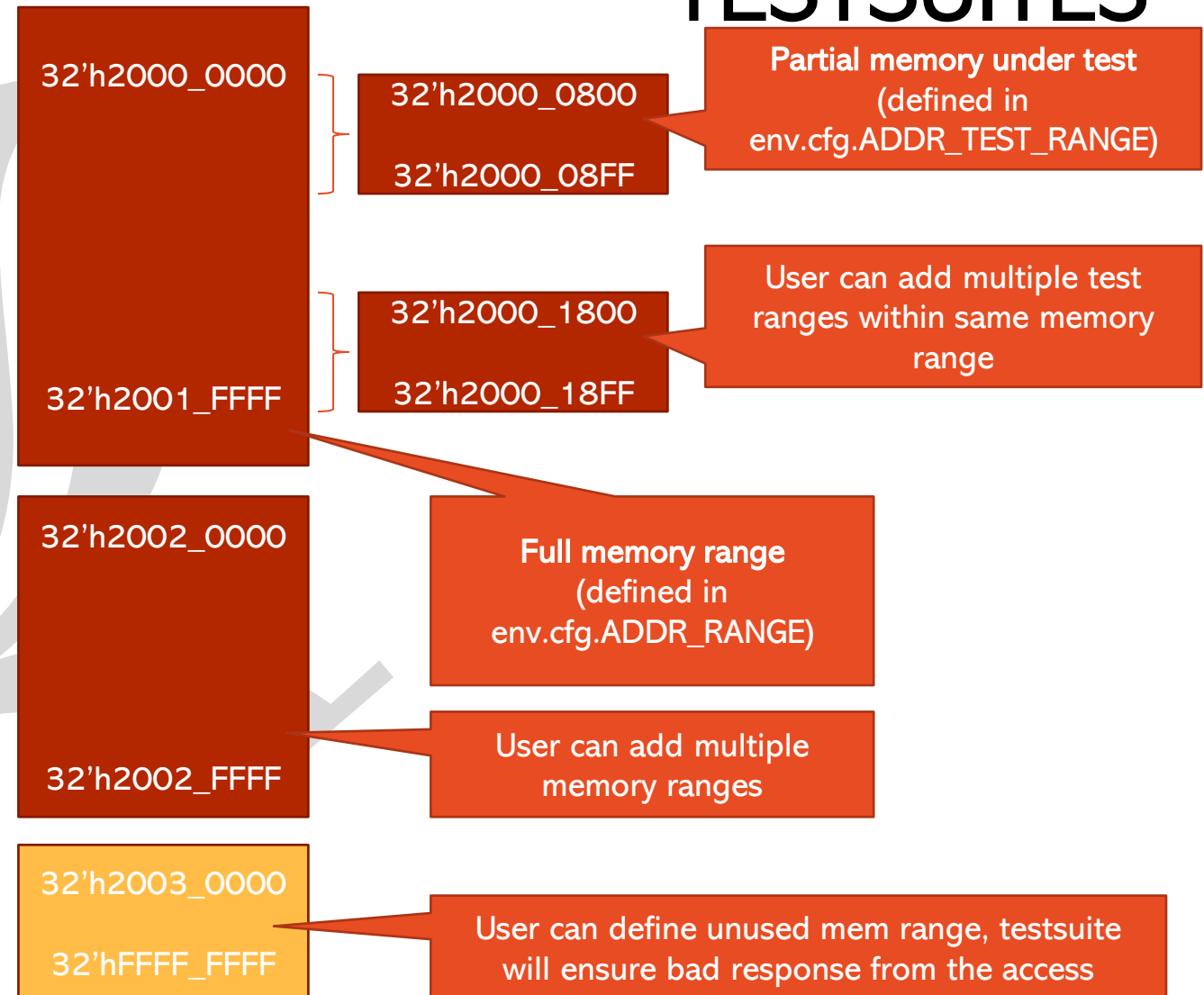
```
make run t=lvm_ahb_master_testsuites_test plusarg="+lvm_ahb_total_pkt=150 +lvm_ahb_scenario=1"
```


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)  
);
```

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

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 AHB VIP

1. Request a copy of the LVM AHB 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_AHB_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 AHB copy is correct.

```
cd $LVM_AHB_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

```
// lvm class
$LVM_AHB_PATH/../../lvm_class/lvm_class.f

// LVM_AHB UVC filelist
-f $LVM_AHB_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 AHB 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_AHB_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_WIDTH(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_AHB_VALUES      m_ahb_env;
uvm_status_e      status;        // UVM status
uvm_reg_data_t    myrdata;       // Read Data
// API outputs
`LVM_AHB_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_AHB_VALUES::type_id::create("m_ahb_env", this);
    ...
endfunction
```

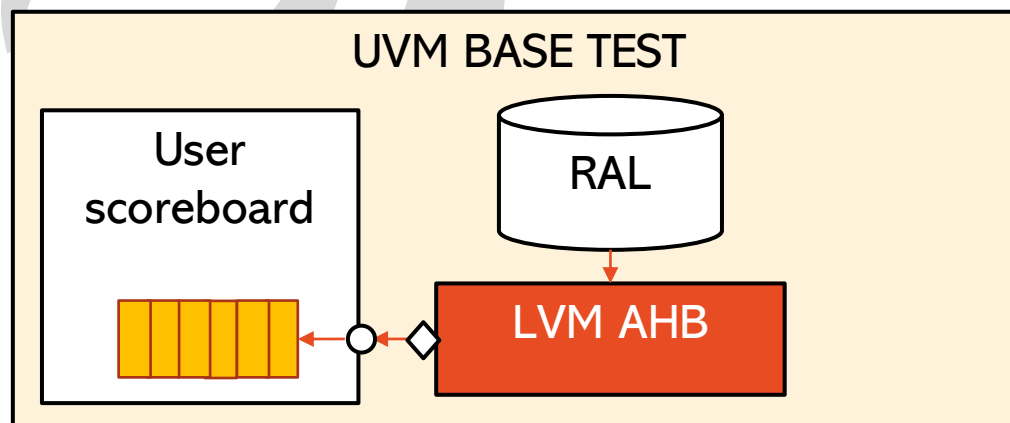
CONNECTING & CONFIGURING THE UVC

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);  
    // Connect to your RAL's default map  
    if(m_ahb_env.cfg.has_prd)  
        m_ahb_env.prd.map      = urm.default_map;  
    if(m_ahb_env.cfg.has_adp)  
        urm.default_map.set_sequencer(m_ahb_env.agt.sqr,  
        m_ahb_env.adp);  
    // Optional step to improve RAL predictor performance  
    m_ahb_env.cfg.add_RAL_ADDR_RANGE    (  
        .start_addr(<your ral min addr>),  
        .end_addr(<your ral max addr>),  
        .expected_resp({LVM_AHB_OK}));
```

```
        if(m_ahb_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_ahb_env.port.connect(<your sbd export>);  
        m_ahb_env.port.connect(<your cov export>);  
    endfunction
```

WHAT YOU HAVE NOW

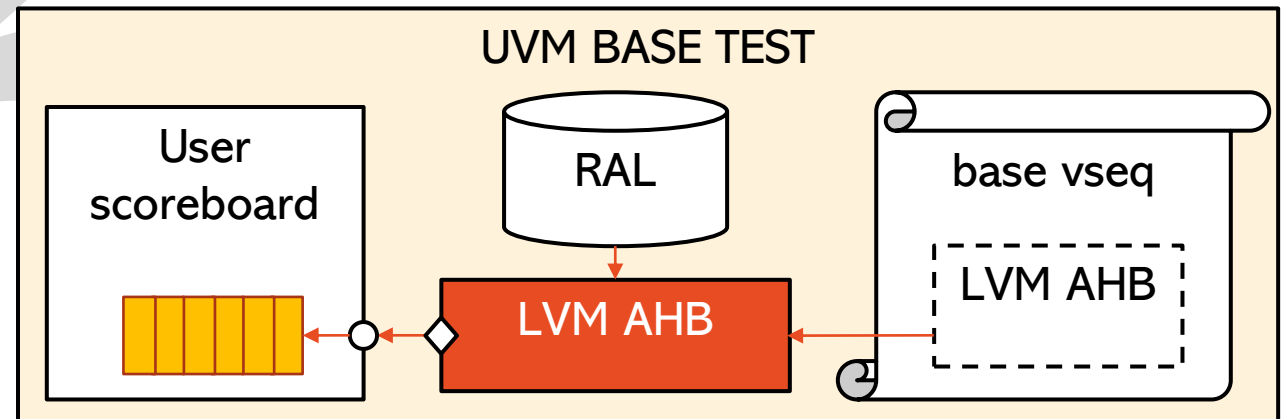


OPTIONAL STEP

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>.AHB_env = m_ahb_env;  
endfunction
```



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

8. Declare all AHB 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 AHB reset signal>;
```

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

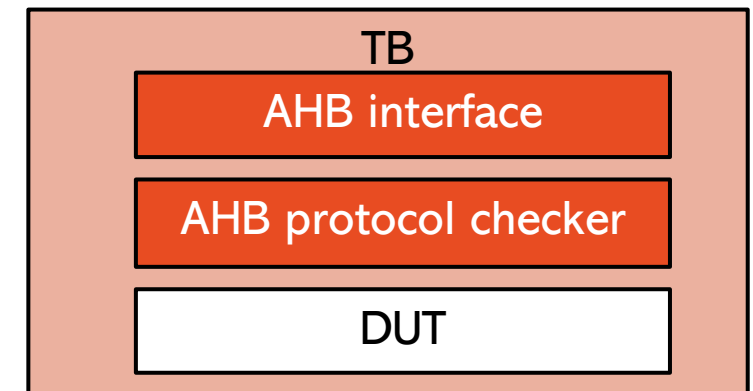
9. Include the lvm_ahb_connection.sv (mostly need modifications)

```
// 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_AHB_WRAP8),
    .HSIZE      (LVM_AHB_2B),
    .HMASTLOCK  (LVM_AHB_M_UNLOCK),

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

```
m_ahb_env.write(
    .HADDR      (32'h1000_00F8),
    .HBURST     (LVM_AHB_INCR),
    .HSIZE      (LVM_AHB_4B),
    .HMASTLOCK  (LVM_AHB_M_UNLOCK),

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

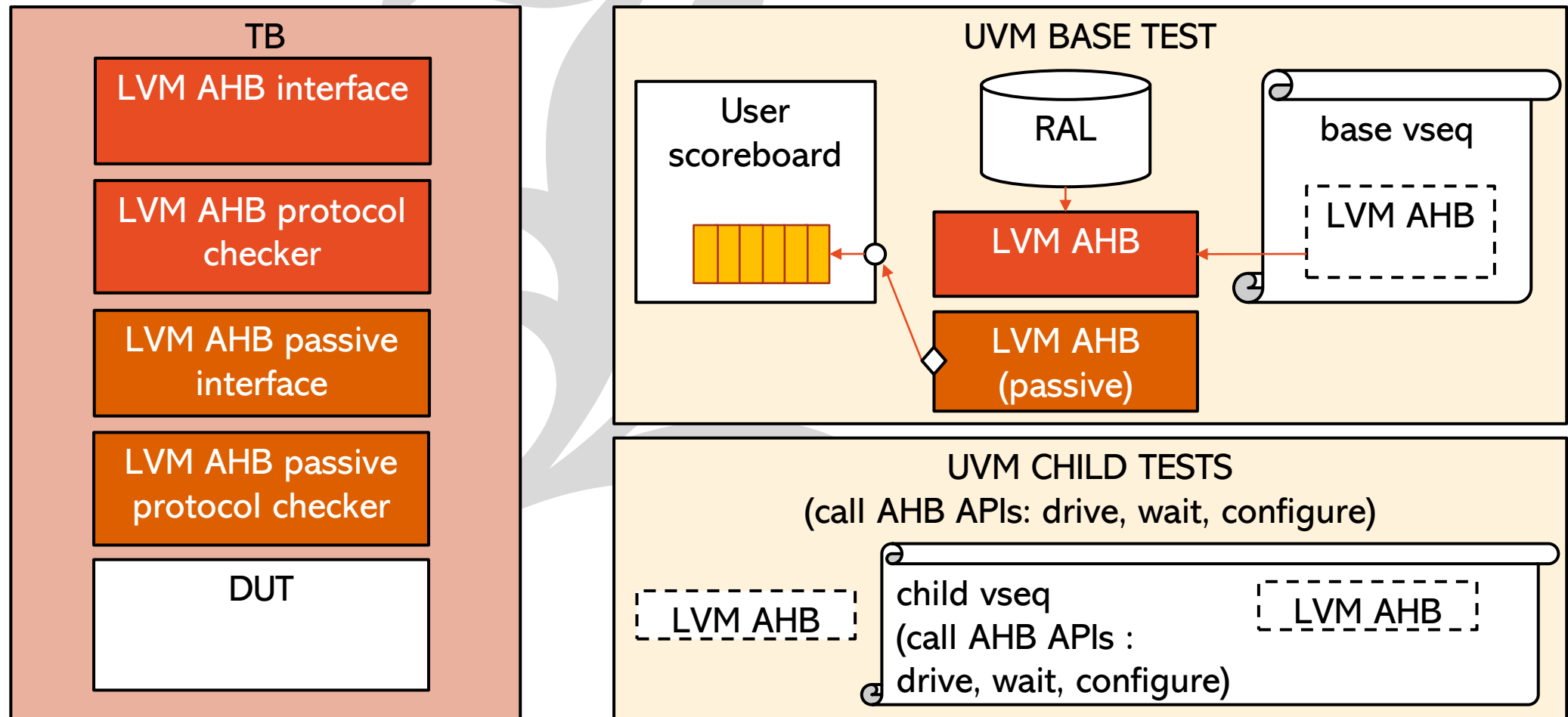
INSTALLING LVM AHB AS PASSIVE UVC

PASSIVE AHB UVC

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

```
make sim passive_ahb=on t=lvm_ahb_01_ral_access_test
```


EXAMPLE OF TWO LVM AHB 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_AHB_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_AHB_VALUES::type_id::create("m_lvm_ahb_passive", this);
```

```
// Turning off master env's monitor & predictor (if you are using LVM's AHB UVC)
```

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

```
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);  
  
    // Connect to your RAL's default map  
    if(m_lvm_ahb_passive.cfg.has_prd)  
        m_lvm_ahb_passive.prd.map      = urm.default_map;  
  
    // Connection to your SBD / Coverage  
    m_lvm_ahb_passive.port.connect(my_sbd.user_export);  
  
    // Optional step to improve RAL predictor performance  
    m_lvm_ahb_passive.cfg.add_RAL_ADDR_RANGE    (  
        .start_addr(<your ral min addr>),  
        .end_addr(<your ral max addr>),  
        .expected_resp({LVM_AHB_OK}));
```

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

4. By reusing all AHB wires in module top, including the clock supply and reset, connect them to AHB 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

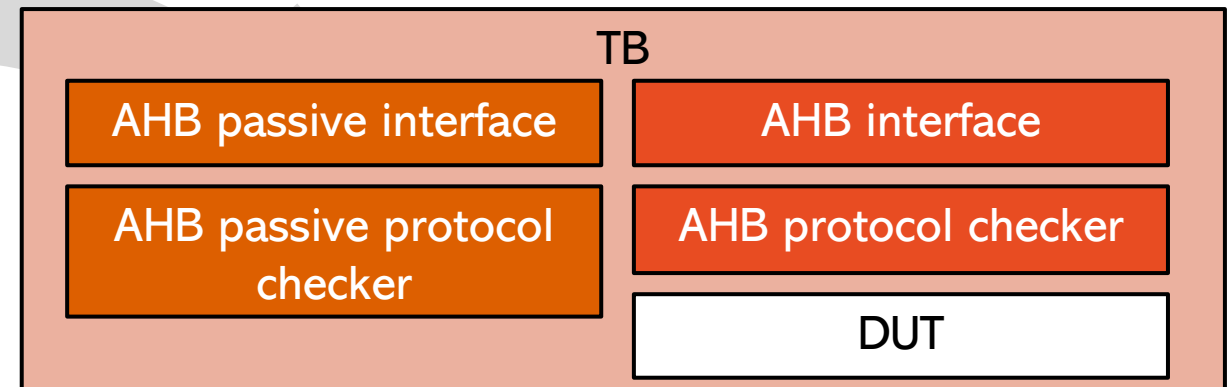
assign HCLK      = <your clock supply>;

assign HRESETn   = <your AHB 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 AHB traffic into a file.
 - Start checking the AHB protocol on the signals.
 - Start writing the UVC's analysis TLM port.

TERMS AND CONDITIONS

155

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.