

Design & Verification of A Re-Configurable Synthesizable AMBA AHB-LITE

SystemVerilog Verilog UVM SVA Bash Scripting Do files

Abdelrahman Mohamad
Yassien



1 Table of Contents

1. Advanced Microcontroller Bus Architecture (AMBA) overview	3
1.1 What is Advanced Microcontroller Bus Architecture?	3
2. AMBA Advanced High-performance bus (AHB) lite overview	4
2.1 AHB lite bus constituents.....	4
Type chapter title (level 3)	4
3. AMBA AHB-lite Design Components	5
3.1 Manager.....	5
3.2 Subordinate	5
3.1 Interconnect	5
Decoder	5
Multiplexor.....	6
Interconnect with AHB Interfaces.....	6
4. Design of a re-configurable synthesizable pipelined AMBA AHB LITE	7
4.1 Design constituents	7
5. Verification of a reconfigurable, pipelined AMBA AHB lite using UVM and SVA	8
5.1 The UVM Environment Hierarchy.....	8
5.2 AMBA AHB-LITE Test-Plan.....	9
5.3 Coverage Groups	11
5.3.1 Instance Based Coverage Groups	11
5.3.1.1 Data Frame Toggling Instance-based Cover groups	11
5.3.1.2 Data Transition Toggling Instance-based Cover groups	12
5.3.1.3 Data Transition Instance-based Cover groups	13
5.3.1.4 Data Frame Instance-based Cover groups	13
5.3.2 Static Based Coverage Groups.....	14
5.4 Issues Faced During the Verification Process	15
5.3.1 Mimicking Pipelining	15
5.3.2 uvm_tlm_analysis_fifo overwriting.....	15
5.3.3 When is the test ending? Test Termination, How and When?	17
5.3.4 Asynchronous reset in pipelined designs & verification environments.....	18
5.5 Waveforms.....	19
5.5.1 runall_waited_test waveforms	19
5.5.2 runall_test waveforms.....	21
5.6 Assertion Table	22

6. Future Work	23
6.1 Verification Wise.....	24
6.1.1 Future UVM Environment Hierarchy.....	25
6.2 Design Wise	25
6.3 Scripting Wise	25

■ List of Figures

Figure 1 AMBA versions, components & specifications	4
Figure 2 AMBA AHB lite	5
Figure 3 AMBA AHB Manager	6
Figure 4 AMBA AHB Subordinate	6
Figure 5 AMBA AHB lite decoder	6
Figure 6	7
Figure 7 UVM Hierarchy diagram of the Verification Environment	9
Figure 8 Instance-Based Cover group Body used for toggle coverage	12
Figure 9 Instance-Based Cover group Body used for data-frame toggle coverage	13
Figure 10 Instances handles	13
Figure 11 Instances Construction	13
Figure 12 Instances Sampling	13
Figure 13 Instance-Based Cover group Body used for data-transition toggle coverage	13
Figure 14 The rest of the Instance-Based Cover groups for DF & DT toggle coverage	13
Figure 15 Instance-Based Cover group body for data frames to cover all HSIZE possible value transitions (DT)	14
Figure 16 Instances handles	14
Figure 17 Instances Creation	14
Figure 18 Instance-Based Cover group body for data frames to cover all HSIZE possible values (DF)	14
Figure 19 Static Cover groups Construction	15
Figure 20 Static Cover groups Sampling	15
Figure 21 Explaining the common misconception of how UVM_TLM_FIFO or UVM_TLM_ANALYSIS_FIFO function, resulting in overwrite (1)	16
Figure 22 Explaining the common misconception of how UVM_TLM_FIFO or UVM_TLM_ANALYSIS_FIFO function, resulting in overwrite (2)	17
Figure 23 Explaining the common misconception of how UVM_TLM_FIFO or UVM_TLM_ANALYSIS_FIFO function, resulting in overwrite (3)	17
Figure 24 Explaining the common misconception of how UVM_TLM_FIFO or UVM_TLM_ANALYSIS_FIFO function, resulting in overwrite (4)	18
Figure 25 uvm phase_ready_to_end & the delay_phase task	18
Figure 26 uvm final_phase	18
Figure 27 Explaining the proposed solutions for on how to mirror the effect of the asynchronous reset behaviour on a DUT in the Verification Environment	19
Figure 28 Asynchronous reset behaviour in the dut	20
Figure 29 Values before & After RESET is asserted	20
Figure 30 Values during IDLE transfer	20
Figure 31 Values during WRAP16 Burst Transfer	21
Figure 32 Values during INCR Burst Transfer	21
Figure 33 Values during accessing default subordinate (SUB4) & the resulting error	21
Figure 34 Values during INCR4 Write Burst followed by a SINGLE Write transfer	22
Figure 35 Values during WRAP4 Write Burst followed by an INCR Write transfer to default subordinate (SUB4) resulting in ERROR	22
Figure 36 Values during WRAP8 Read Burst transfer to default subordinate (SUB4) resulting in ERROR followed by an INCR16 Write transfer	22
Figure 37 Values during INCR Read Burst transfer	22
Figure 38 Future UVM Hierarchy diagram	25

1. Advanced Microcontroller Bus Architecture (AMBA) overview

1.1 What is Advanced Microcontroller Bus Architecture?

AMBA is a standardized design MCU architecture which arm developed to simplify and standardize the design of microprocessors, microcontrollers, peripherals at very different levels of abstraction. It is made to be widely reused in different SoC parts and ASICs utilizing reusability, compatibility, flexibility and support.

Bus interfaces, like AMBA, in general are categorized according to their **Bandwidth & Latency**, therefore amba was made to try and achieve the best possible results in both fields.

AMBA has gradually developed over time, with different components/features being added to it to ensure it being a state-of-the-art architecture which something as impactful as Arm's cortex M & several other influential and widely used paradigms to rely on.

Key AMBA specifications	AMBA	AMBA 2	AMBA 3	AMBA 4	AMBA 5
CHI Coherent Hub Interface	A credited coherency protocol Layered architecture for scalability				CHI
ACE AXI Coherency Extensions	A superset of AXI — system-wide coherency across multicore clusters			ACE +Lite	ACE5 +Lite
AXI Adv. eXtensible Interface	AXI supports separate A/D phases, bursts, multiple outstandings addresses, and OoO responses		AXI3	AXI4 +Lite, +Stream	AXI5
AHB Adv. High-performance Bus	AHB supports 64 and 128 bit multi-manager AHB-Lite is for single managers		AHB	AHB +Lite	AHB5 +Lite
APB Advanced Peripheral Bus	System bus for low bandwidth peripherals	APB	APB2	APB3	APB4

Figure 1
[AMBA versions, components & specifications](#)

In this document we're going to focus on AMBA AHB5, discuss its specifications & report on the design & verification

2 AMBA Advanced High-performance bus (AHB) lite overview

AMBA AHB is a bus interface suitable for **high-performance** synthesizable designs. It defines the interface between components, such as Managers, interconnects, and Subordinates.

2.1 AHB lite bus constituents:

- Managers
- Subordinates
- Address decoders
- Multiplexors

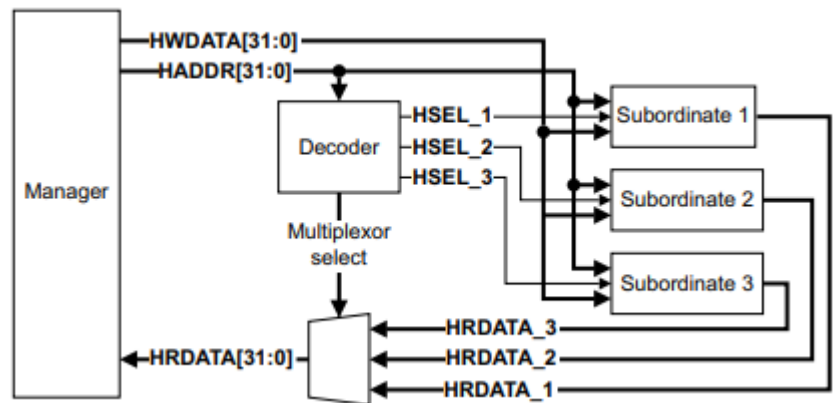


Figure 2
AMBA AHB lite

2.2 AMBA AHB implements the features required for high-performance, high clock frequency systems including:

- Burst transfers
- Single clock-edge operation
- Non-tristate implementation
- Configurable data bus widths
- Configurable address bus widths

The most common **AHB Subordinates** are internal memory devices, external memory interfaces, and high-bandwidth peripherals. Although low-bandwidth peripherals can be included as AHB Subordinates, for system performance reasons, they typically reside on the AMBA Advanced Peripheral Bus (APB). Bridging between the higher performance AHB and APB is done using an AHB Subordinate, known as an APB bridge.

3 AMBA AHB-lite Design Components

3.1 Manager

A manager communicates with the subordinate through 2 types of feed:

- Control Signals
- Data Signal

AHB also supports **pipelined communication** to increase throughput and maintain the high-performance paradigm.

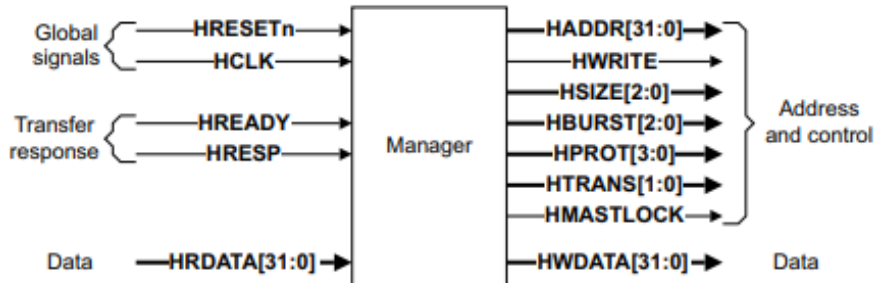


Figure 3
AMBA AHB Manager

3.2 Subordinate

A subordinate receives transfers from the manager and responds accordingly:

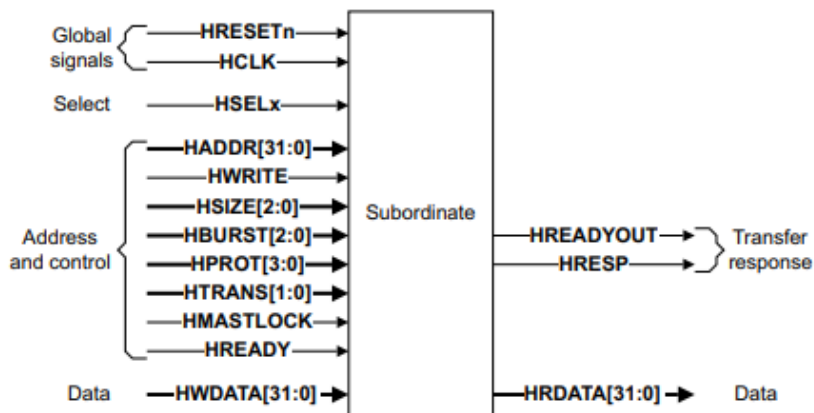


Figure 4
AMBA AHB Subordinate

The Subordinate signals back to the Manager:

- ✚ The **completion or extension** of the bus transfer.
- ✚ The **success or failure** of the bus transfer.

3.3 Interconnect

An interconnect component provides the connection between Managers and Subordinates in a system.

In the case of Single Master-Multiple Subordinates (AHB lite), a decoder & a multiplexor are used in the interconnect's stead.

▪ Decoder

The decoder receives the **address provided by the Manager** and in turn provides asserts the select line **HSELx** for the **subordinate** which the transaction is meant for.

The decoder also sends the **same signal to the multiplexor** **delayed by the number of stages of the pipelining**, or the **number of cycles needed for the subordinate to provide a response**.

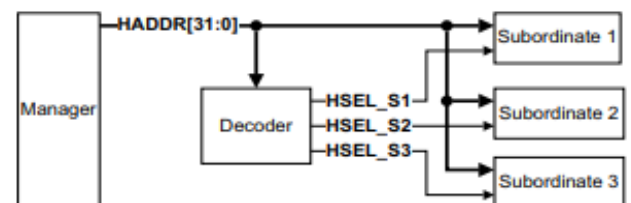


Figure 5
AMBA AHB lite decoder

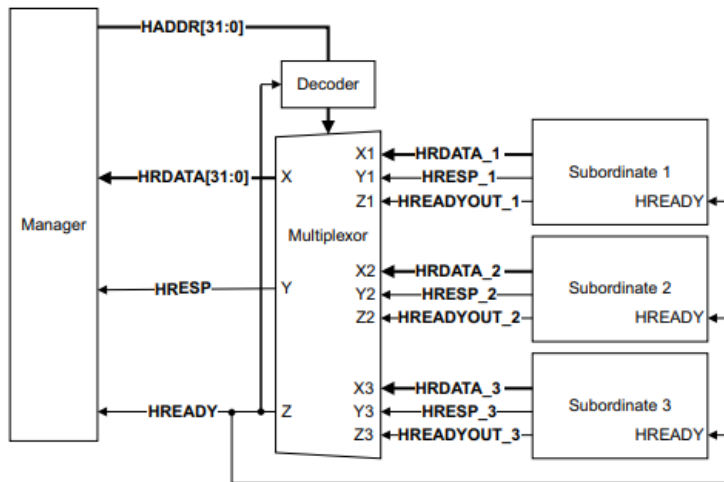


Figure 6
AMBA AHB lite Multiplexor

▪ Interconnect with AHB interfaces

Generic interconnect products can offer AHB as an interface option, among others such as AMBA AXI or AMBA APB. Figure 4-3 shows how a generic interconnect might implement HTRANS, HREADY, and HSEL.

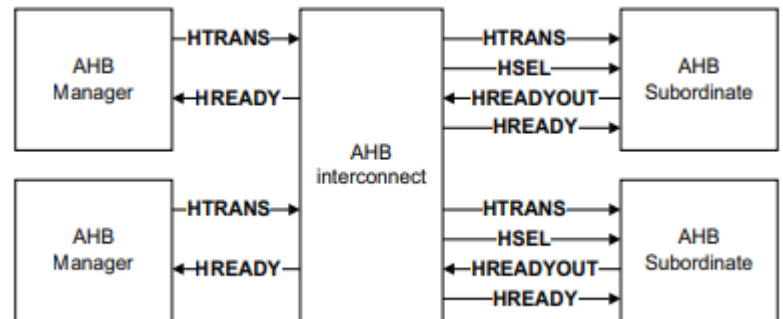


Figure 7
AMBA AHB Interconnect

The Manager side of the interconnect uses HTRANS to indicate valid transfers and has a single HREADY signal. HREADY is used to stall a transfer when the Subordinate has inserted wait states or when the Manager is waiting for arbitration from within the interconnect. The Subordinate side of the interconnect also includes an HSEL output and two HREADY signals. HREADYOUT from the Subordinate is passed to the Managers to insert wait states. The HREADY output from the interconnect can be used to stall a Subordinate if the data phase of the previous transfer is stalled.

An alternative implementation would be for HSEL to be tied HIGH on the Subordinates and the interconnect to override HTRANS to IDLE for unselected Subordinates.

4 Design of a re-configurable synthesizable pipelined AMBA AHB LITE

4.1 Design constituents:

- Verification Environment acting as the Manager.
- Six Subordinates
- Address Decoder
- Multiplexor

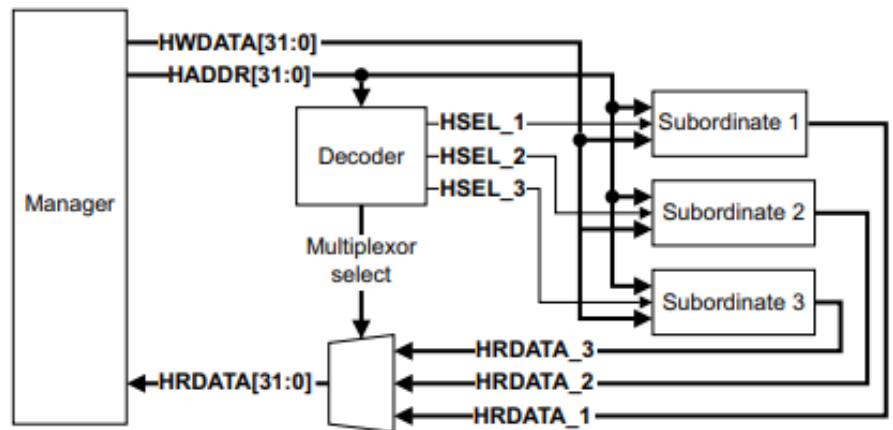


Figure 8

AMBA AHB lite depicting the connections between different subordinates & the manager

Subordinate 1, 2 & 3:

- **Acting as normal subordinates** integrated with memory blocks (ROMs).
- Each requiring **no privilege** level from HPROT

Subordinate 4:

- **Acting as the default subordinate** which is prescribed in the specifications document.
- **Answering each select with an ERROR as well as a READY response.**

Subordinate 5:

- **Acting as normal subordinates** integrated with memory blocks (ROMs).
- **Requiring a specified privilege level from HPROT for WRITE operations.**

Subordinate 6:

- **Acting as normal subordinates** integrated with memory blocks (ROMs).
- **Requiring a specified privilege level from HPROT for WRITE & READ operations.**

Re-Configurable Design:

- HWDATA & HRDATA** data busses are reconfigurable to different widths (**32, 64, 128, 256, 512, 1024**) as per the specification document.
- HADDR** address bus is reconfigurable to widths of **10 & 16 & 32** bits.

Further Additions:

- For challenging verification dilemmas, I elected to **use ROMs instead of RAMs.**
- The data being transmitted is either opcode/data (not accounted for) while also being **non-cacheable and non-bufferable.**

5 Verification of a reconfigurable, pipelined AMBA AHB lite using UVM and SVA

5.1 The UVM Environment Hierarchy

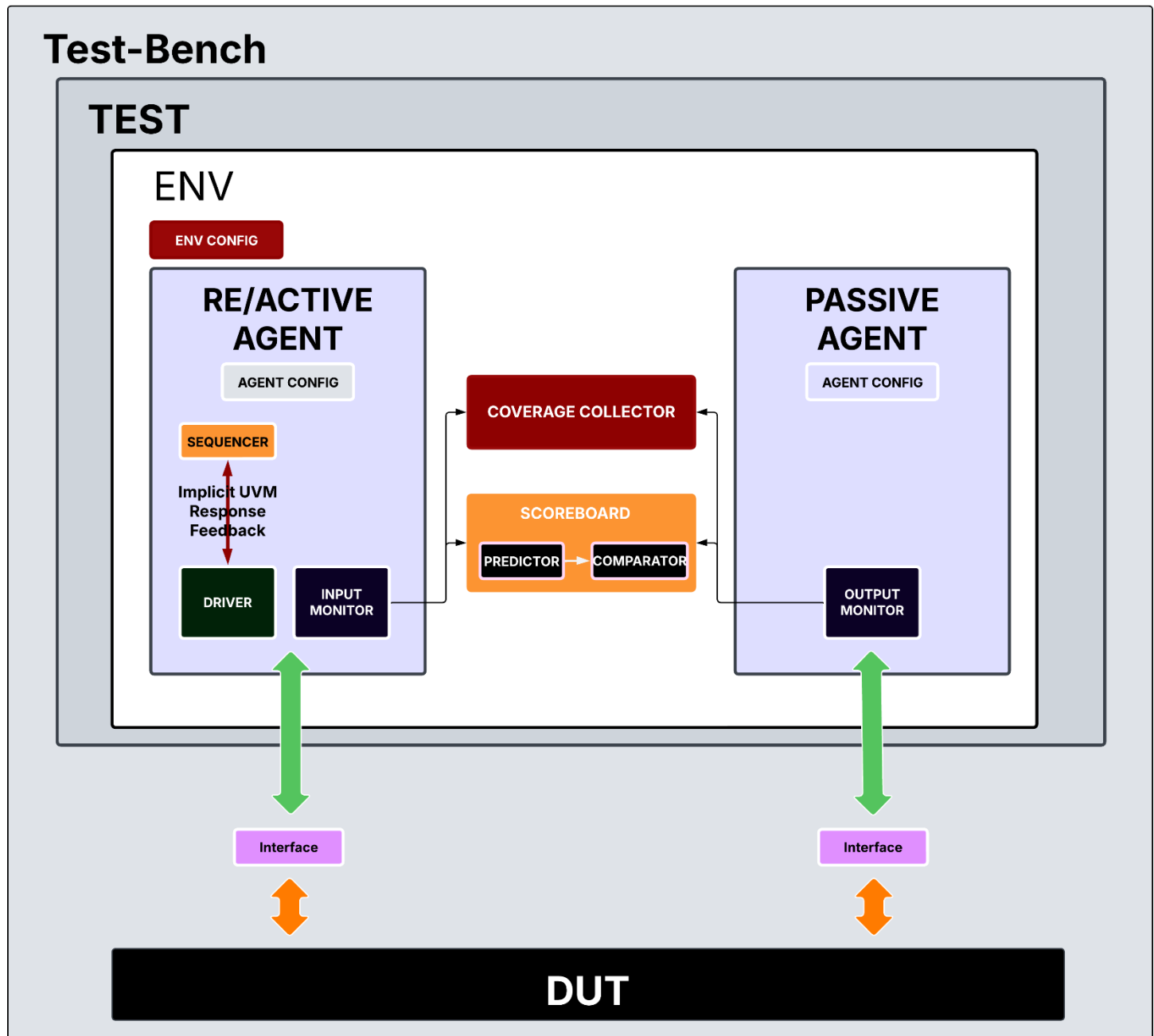


Figure 7 UVM Hierarchy diagram of the Verification Environment

5.2 AMBA AHB-lite test-plan

TESTS	DESCRIPTION	NO. BUGS FOUND
reset_test	: Asserting reset for 15 clk cycles and checking the outputs	Passed
IDLE_test	: Randomizing stimulus and driving HTRANS = IDLE	Passed
WRITE_SINGLE_test	: Randomizing stimulus and READ from the AHB subordinates HBURST = SINGLE	Passed
READ_SINGLE_test	: Randomizing stimulus and WRITE to the AHB subordinates HBURST = SINGLE	Passed
WRITE_INCR_test	: Randomizing stimulus and driving a WRITE with HBURST = INCR (randomized INCR length)	Passed
READ_INCR_test	: Randomizing stimulus and driving a READ with HBURST = INCR (randomized INCR length)	Passed
WRITE_READ_INCR_test	: Randomizing stimulus and driving a WRITE then READ with HBURST = INCR (randomized INCR length)	Passed
WRITE_WRAP4_test	: Randomizing stimulus and driving a WRITE with HBURST = WRAP4	Passed
READ_WRAP4_test	: Randomizing stimulus and driving a READ with HBURST = WRAP4	Passed
WRITE_READ_WRAP4_test	: Randomizing stimulus and driving a WRITE then READ with HBURST = WRAP4	Passed
WRITE_INCR4_test	: Randomizing stimulus and driving a WRITE with HBURST = INCR4	Passed
READ_INCR4_test	: Randomizing stimulus and driving a READ with HBURST = INCR4	Passed
WRITE_READ_INCR4_test	: Randomizing stimulus and driving a WRITE then READ with HBURST = INCR4	Passed
WRITE_WRAP8_test	: Randomizing stimulus and driving a WRITE with HBURST = WRAP8	Passed
READ_WRAP8_test	: Randomizing stimulus and driving a READ with HBURST = WRAP8	Passed
WRITE_READ_WRAP8_test	: Randomizing stimulus and driving a WRITE then READ with HBURST = WRAP8	Passed
WRITE_INCR8_test	: Randomizing stimulus and driving a WRITE with HBURST = INCR8	Passed

TESTS	DESCRIPTION	NO. BUGS FOUND
READ_INCR8_test	: Randomizing stimulus and driving a READ with HBURST = INCR8	Passed
WRITE_READ_INCR8_test	: Randomizing stimulus and driving a WRITE then READ with HBURST = INCR8	Passed
WRITE_WRAP16_test	: Randomizing stimulus and driving a WRITE with HBURST = WRAP16	Passed
READ_WRAP16_test	: Randomizing stimulus and driving a READ with HBURST = WRAP16	Passed
WRITE_READ_WRAP16_test	: Randomizing stimulus and driving a WRITE then READ with HBURST = WRAP16	Passed
WRITE_INCR16_test	: Randomizing stimulus and driving a WRITE with HBURST = INCR16	Passed
READ_INCR16_test	: Randomizing stimulus and driving a READ with HBURST = INCR16	Passed
WRITE_READ_INCR16_test	: Randomizing stimulus and driving a WRITE then READ with HBURST = INCR16	Passed
runall_test	Runs all the above tests' sequences consecutively with no delay whatsoever (Stress Testing)	Passed
runall_waited_test	Runs all of the above tests' sequences consecutively and waits for the response before driving another item/starting another sequence	Passed
ADDRESS_ERROR_INJECTION_test	:Overriding constraints to inject an invalid address during a burst transaction to check error response.	Passed
PRIVILEGE_ERROR_INJECTION_wr_test	: Overriding constraints to inject subordinate_p_wr with a read & write operation without the correct HPROT value to check privilege error response	Passed

Table 1
Includes all tests done in the verification process & their explanation

5.3 Coverage Groups

5.3.1 Instance based cover groups

- **HWDATA_df_cg**: Covering the toggling of each bit of HWDATA.

5.3.1.1 Data Frame Toggling Instance-based Cover groups

HWDATA is a **re-configurable bus that can be configured as low as 32 bits of width and up to 1024 bits of width**. How can we ensure that all its driving pins/wires are working correctly without writing extensive & exhausting code? Through this.

```
covergroup HWDATA_df_cg(input bit [DATA_WIDTH-1:0] position, ref bit [DATA_WIDTH-1:0] vector);
  df: coverpoint (vector & position) != 0;
  option.per_instance = 1;
endgroup : HWDATA_df_cg
```

Figure 8 Instance-Based Cover group Body used for toggle coverage

HOW IT WORKS: Assume HWDATA is a 4-bit width bus

Table2 (SAMPLE1) at 1st clock edge: HWDATA/vector = 0000 for example,

i	position	vector	result
i=0	0001	0000	0000
i=1	0010	0000	0000
i=2	0100	0000	0000
i=4	1000	0000	0000

Now the bin of each instance has collected the coverage of ZERO for its respective bit of HWDATA.

Table3 (SAMPLE2) at 2nd clock edge: HWDATA/vector = 0101 for example,

l	position	vector	Initial result	result
i=0	0001	0101	0000	0001
i=1	0010	0101	0000	0000
i=2	0100	0101	0000	0100
i=4	1000	0101	0000	0000

Now the bin of each instance collects the coverage again, but this time, only 2 bits of HWDATA got toggled

Table4 (SAMPLE3) at 3rd clock edge: HWDATA/vector = 1010 for example,

l	position	vector	Initial result	old result	result
i=0	0001	1010	0000	0001	0000
i=1	0010	1010	0000	0000	0010
i=2	0100	1010	0000	0100	0000
i=4	1000	1010	0000	0000	1000

Now the bin of each instance collects the coverage again, but this time, the remaining bits of HWDATA got toggled.
therefore achieving 100% coverage for toggling for wide buses.

Making the cover group body:

```
covergroup HWDATA_df_tog_cg(input bit [DATA_WIDTH-1:0] position, input sequence_item cov);
    option.per_instance = 1;
    df: coverpoint (cov.HWDATA & position) != 0 iff(cov.HRESETn && cov.HTRANS != IDLE && cov.HTRANS != BUSY);
endgroup : HWDATA_df_tog_cg
```

Figure 9 Instance-Based Cover group Body used for data-frame toggle coverage

Making the array of instances, an instance for each bit of the bus:

```
HWDATA_df_tog_cg HWDATA_df_tog_cg_bits [DATA_WIDTH-1:0];
```

Figure 10 Instances handles

Construct each instance with the variable *i* being shifted to the left at each loop iteration (0001, 0010, 0100...etc.):

```
foreach(HWDATA_df_tog_cg_bits[i]) HWDATA_df_tog_cg_bits[i] = new(1'b1<<i, input_cov_copied);
```

Figure 11 Instances Construction

Sample each time the write function is called in the coverage collector:

```
foreach(HWDATA_df_tog_cg_bits[i]) HWDATA_df_tog_cg_bits[i].sample();
```

Figure 12 Instances Sampling

5.3.1.2 Data Transition Toggling Instance-based Cover groups

- **HWDATA_dt_tog_cg:** Covering the toggling of each bit of HWDATA.

```
covergroup HWDATA_dt_tog_cg(input bit [DATA_WIDTH-1:0] position, input sequence_item cov);
    option.per_instance = 1;
    dt: coverpoint (cov.HWDATA & position) != 0 iff(cov.HRESETn && cov.HTRANS != IDLE && cov.HTRANS != BUSY){
        bins tr[] = (0 => 1, 1 => 0);
    }
endgroup : HWDATA_dt_tog_cg
```

Figure 13 Instance-Based Cover group Body used for data-transition toggle coverage

The next cover groups do the same for different buses:

- **HADDR_df_tog_cg:** Covering the toggling of each bit of HWDATA.
- **HADDR_dt_tog_cg:** Covering the toggling of each bit of HWDATA.
- **HSEL_df_tog_cg:** Covering the toggling of each bit of HWDATA.
- **HSEL_dt_tog_cg:** Covering the toggling of each bit of HWDATA.

```
covergroup HADDR_df_tog_cg(input bit [ADDR_WIDTH-BITS_FOR_SUBORDINATES-1:0] position, input sequence_item cov);
    option.per_instance = 1;
    df: coverpoint (cov.HADDR[ADDR_WIDTH-BITS_FOR_SUBORDINATES-1:0] & position) != 0 iff(cov.HRESETn && cov.HTRANS != IDLE && cov.HTRANS != BUSY);
endgroup : HADDR_df_tog_cg

covergroup HADDR_dt_tog_cg(input bit [ADDR_WIDTH-BITS_FOR_SUBORDINATES-1:0] position, input sequence_item cov);
    option.per_instance = 1;
    dt: coverpoint (cov.HADDR[ADDR_WIDTH-BITS_FOR_SUBORDINATES-1:0] & position) != 0 iff(cov.HRESETn && cov.HTRANS != IDLE && cov.HTRANS != BUSY){
        bins tr[] = (0 => 1, 1 => 0);
    }
endgroup : HADDR_dt_tog_cg

covergroup HSEL_df_tog_cg(input bit [BITS_FOR_SUBORDINATES-1:0] position, input sequence_item cov);
    option.per_instance = 1;
    df: coverpoint (cov.HADDR[ADDR_WIDTH-1:ADDR_WIDTH-BITS_FOR_SUBORDINATES] & position) != 0 iff(cov.HRESETn && cov.HTRANS != IDLE && cov.HTRANS != BUSY);
endgroup : HSEL_df_tog_cg

covergroup HSEL_dt_tog_cg(input bit [BITS_FOR_SUBORDINATES-1:0] position, input sequence_item cov);
    option.per_instance = 1;
    dt: coverpoint (cov.HADDR[ADDR_WIDTH-1:ADDR_WIDTH-BITS_FOR_SUBORDINATES] & position) != 0 iff(cov.HRESETn && cov.HTRANS != IDLE && cov.HTRANS != BUSY){
        bins tr[] = (0 => 1, 1 => 0);
    }
endgroup : HSEL_dt_tog_cg
```

Figure 14 The rest of the Instance-Based Cover groups for DF & DT toggle coverage

5.3.1.3 Data Transition Instance based Cover Groups

- **HSIZE_dt_cg:** Covering the transitions of the HSIZE input values.

This one is a bit tricky, since **HSIZE** can take values up to 7, which in turn forces **HWDATA** & **HRDATA** to **WRITE** & **READ** up to 1024 bits of data (depending on the configuration), so for example, **HSIZE** being 7 while the bus is configured to bit width of 512 would ruin the functional coverage, therefore, **HSIZE**'s array of instances width of the cover group is also controlled by the **HWDATA_WIDTH** configuration.

- Making Cover group body:

```
covergroup HSIZE_dt_cg(input int i, input int j, sequence_item c);
    option.per_instance = 1;
    option.name = $sformatf(" dt: %0d => %0d", i, j);
    dt:coverpoint c.HSIZE iff (c.HRESETn) {
        bins tr[] = (i => j);
    }
endgroup : HSIZE_dt_cg
```

Figure 15 Instance-Based Cover group body for data frames to cover all HSIZE possible value transitions (DT)

1. Making the array of instances for each possible value of HSIZE:

```
HSIZE_dt_cg HSIZE_dt_cg_vals [AVAILABLE_SIZES][AVAILABLE_SIZES];
```

Figure 16 Instances handles

- Construct each instance with the value of i being the predeterminant to all the possible values of HSIZE:

```
foreach(HSIZE_dt_cg_vals[i,j]) HSIZE_dt_cg_vals[i][j] = new(i, j, input_cov_copied);
```

Figure 17 Instances Creation

- Sample each time the write function is called in the coverage collector:

```
foreach(HSIZE_dt_cg_vals[i,j]) HSIZE_dt_cg_vals[i][j].sample();
```

5.3.1.4 Data Frame Instance based Cover Groups

- **HTRANS_df_cg:** Covering the toggling of each bit of HWDATA.

```
covergroup HTRANS_df_cg(input int i, input sequence_item c);
    option.per_instance = 1;
    option.name = $sformatf(" df = %0d", i);
    option.weight = ((i == 1)?0:1);
    df:coverpoint c.HTRANS iff (c.HRESETn) {
        bins tr[] = {i};
        ignore_bins unreachable = {1};
    }
endgroup : HTRANS_df_cg
```

Figure 18 Instance-Based Cover group body for data frames to cover all HSIZE possible values (DF)

The next cover groups do the same for each of the following buses, each having their own df & dt value cover points

- **HRESET_df_cg**: Covering the data frame of each value for HRESETn.
- **HTRANS_df_cg**: Covering the data frame of each value for HTRANS.
- **HBURST_df_cg**: Covering the data frame of each value for HBURST.
- **HSIZE_df_cg**: Covering the data frame of each value for HSIZE
- **HPROT_df_cg**: Covering the data frame of each value for HPROT.

5.3.2 Static Based cover groups

- **RESET_covgrp**: Covering the changes & transition of HRESETn.
- **WRITE_covgrp**: Covering the changes & transition of HWRITE.
- **TRANS_covgrp**: Covering the changes & transition of HTRANS.
- **BURST_covgrp**: Covering the changes & transition of HBURST.
- **SIZE_covgrp**: Covering the changes & transitions of HSIZE.
- **SUBORDINATE_SELECT_covgrp**: Covering the changes & transitions of HSEL.
- **ADDR_covgrp**: Covering the changes & transitions of HADDR.
- **HWDATA_covgrp**: Covering the changes & transitions of the HWDATA with all its different SIZES.

Creation of the static cover groups

```
input_cov_copied = new();
RESET_covgrp      = new;
WRITE_covgrp      = new;
TRANS_covgrp      = new;
BURST_covgrp      = new;
SIZE_covgrp       = new;
SUBORDINATE_SELECT_covgrp = new;
ADDR_covgrp       = new;
HWDATA_covgrp     = new;
```

Figure 19 Static Cover groups Construction

Sampling of the static cover groups

```
RESET_covgrp.sample();
WRITE_covgrp.sample();
TRANS_covgrp.sample();
BURST_covgrp.sample();
SIZE_covgrp.sample();
SUBORDINATE_SELECT_covgrp.sample();
ADDR_covgrp.sample();
HWDATA_covgrp.sample();
```

Figure 20 Static Cover groups Sampling

5.4 Issues Faced During the Verification Process

5.4.1 Mimicking pipelining

Reentrant tasks, events & fork-joins

At first, I thought about making a mix of reentrant functions, tasks and utilising fork - joins to mimic the pipelining of the design. Initially, it seemed to work, but it requires a lot of resources and is less reliable to debug & use. Also, different simulators behave differently with some of the elements used.



Always blocks, counters & flag control

But then, using always blocks & control flags to mimic the behavior of pipelined stages seemed far maintainable, easier to use & debug. Also, much less of a hassle & different simulators have no hand in affecting the functionality due to different behaviors.



5.4.2 uvm_tlm_analysis_fifo overwriting

- A common misconception about how the class `uvm_tlm_analysis_fifo` deals with **SCALAR variables vs. REFERENCE types**:
 - In Systemverilog **class objects & arrays** are almost always **passed by reference**, while **scalar variables** are always **passed by value**.
 - Take **for example a pipelined design, a 3 stage pipelined design**, it takes **3 clock cycles for an input stimulus to be sent and for its output to progress through the DUT the output to be sampled**, in this scenario, one would have to write at least 3 times to the `uvm_tlm_analysis_fifo` (send 3 cycles of input stimulus to the inputs monitor then to the predictor and then the expected seq_item is sent to the comparator P.S. both the predictor and the comparator are classes built in the scoreboard), and that is where the issue of overwriting occurs.

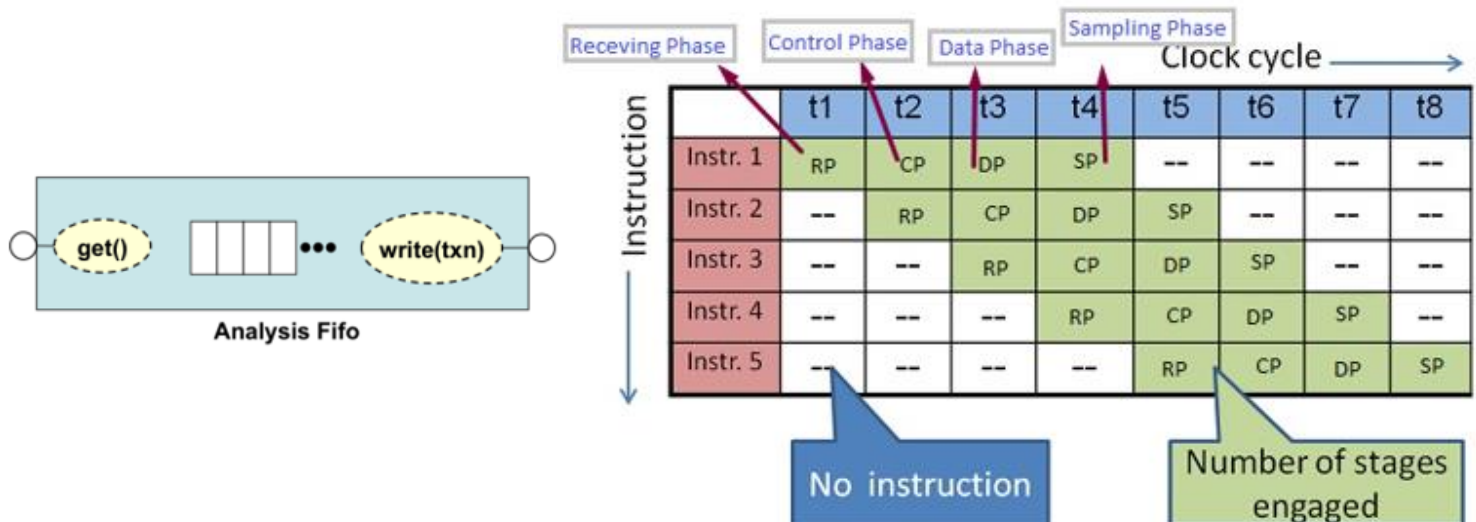


Figure 21 Explaining the common misconception of how `UVM_TLM_FIFO` or `UVM_TLM_ANALYSIS_FIFO` function, resulting in overwriting (1)

5.4.2 uvm_tlm_analysis_fifo overwriting (cont.)

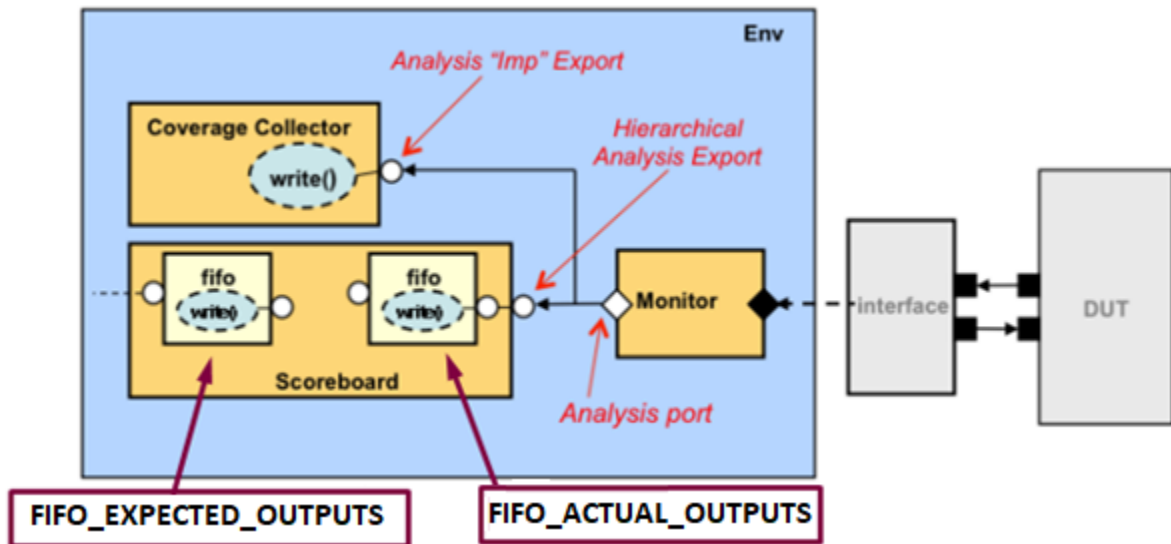


Figure 22 Explaining the common misconception of how UVM_TLM_FIFO or UVM_TLM_ANALYSIS_FIFO function, resulting in overwrite (2)

- In case of multiple writes using `uvm_tlm_analysis_fifo_obj.write(seq_item)` to the `uvm_tlm_analysis_fifo` without getting using the `uvm_tlm_analysis_fifo_obj.get(seq_item)` after each write respectively. The `uvm_tlm_analysis_fifo` does write indeed, and the functionality works correctly but the issue arises due to the fact that the `.write` function writes the handle to the memory location of the `seq_item`, not the `seq_item` itself, therefore if the handle of the `seq_item` is not different on each write, the data that you're trying to send over to the scoreboard will all be the same (same handle of the same `seq_item` pointing to the same memory location).

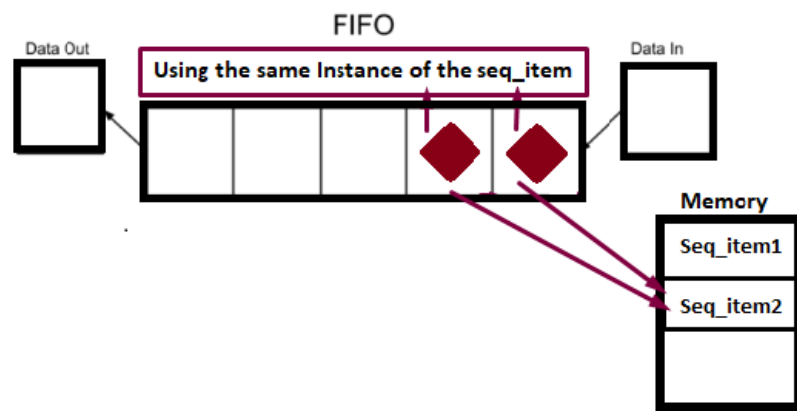


Figure 23 Explaining the common misconception of how UVM_TLM_FIFO or UVM_TLM_ANALYSIS_FIFO function, resulting in overwrite (3)

5.4.2 uvm_tlm_analysis_fifo overwriting (cont.)

Solution?

Create the seq_item instance each time you write to the uvm_tlm_analysis_fifo, that way, every instance handle is different inside the uvm_tlm_analysis_fifo.

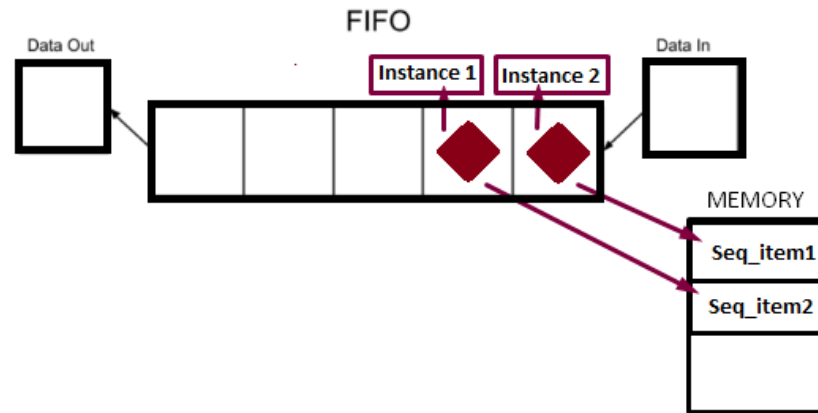


Figure 24 Explaining the common misconception of how UVM_TLM_FIFO or UVM_TLM_ANALYSIS_FIFO function, resulting in overwrite (4)

5.4.3 When is the test ending? Test Termination, How and When?

At the start of the Verification process, the test was ending and therefore the scoreboard turned off before all the sequence items were processed and compared, which made me use the final_phase to debug this behaviour.

```
function void final_phase(uvm_phase phase);
    super.final_phase(phase);
    `uvm_info("SCOREBOARD", "Scoreboard is stopping.", UVM_MEDIUM)
endfunction
```

Figure 26 uvm final_phase

To terminate the test only when all the items have been processed without complex logic in the interface & sequences, as well as remaining faithful to the uvm recommended practices & guidelines: I used 2 static unsigned integers (ints) comparator_tr_counter which increments after every comparison between an expected_seq_item & an actual_seq_item, as well as a predictor_tr_counter which increments after receiving an expected_seq_item in the comparator.

```
function void phase_ready_to_end(uvm_phase phase);
    if (phase.get_name() != "run") return;
    if (sequence_item::COMPARATOR_transaction_counter != sequence_item::PREDICTOR_transaction_counter) begin
        phase.raise_objection(.obj(this));
    end
    fork
        begin
            delay_phase(phase);
        end
    join_none
    end
endfunction

task delay_phase(uvm_phase phase);
    wait(sequence_item::COMPARATOR_transaction_counter == sequence_item::PREDICTOR_transaction_counter);
    phase.drop_objection(.obj(this));
endtask
```

Figure 25 uvm phase_ready_to_end & the delay_phase task

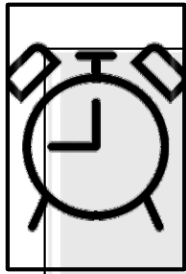
By using the phase_ready_to_end phase and a task called delay_phase, the scoreboard keeps a raised objection until both counters are equal to each other.

NOTE: The predictor_tr_counter is sometimes decremented during clearing the expected_seq_item_fifo when an asynchronous reset is asserted

5.4.4 Asynchronous reset in pipelined designs & verification environments

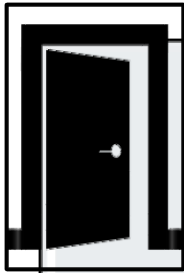
When an Asynchronous reset is implemented, it means that basically when the reset is asserted, the entire design is reset at that same moment in time, which opens a very large number of possibilities of malfunctions for a verification engineer to take into consideration & cover. In a pipelined design, one of those malfunctions would be that if an Input is sent on X clk edge, it would take 3 more clock cycles for the output to progress through the DUT and to be sampled on X+3. Alternatively, in the verification environment, the stimulus that was sent on X clk edge, would be sent to the predictor class inside the scoreboard at time X as well, and therefore assessed accordingly, and the expected_sequence_item would be sent to the comparator and cannot be modified thereafter.

Proposed Solutions?



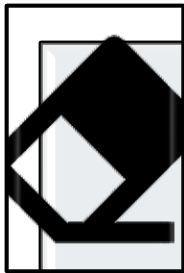
Delaying the Inputs Monitor & The Predictor

In this solution, The timing element would have to have been introduced to the verification environment, which is something I've been actively avoiding as much as I can to speed up the environment and use less resources. Speeding up the environment is something that is currently and has been for a couple of years the spotlight in the DV field.



Making the Predictor fully re-entrant

In this solution, I'd have to make the predictor class fully re-entrant, where the expected_seq_item is not predicted until the number of set transactions inputs is sent to the predictor, and upon checking that there is no reset in them, start predicting the output.



Clearing the Expected Sequence Item FIFO

In this solution, No delay is required, but if a reset transaction is found, then the fifo of the expected sequence_item in the comparator is flushed, also if one of the flushed items already **WRITTEN** a location in the subordinate memory, it reverts back to its original state through a function defined in the predictor.



Figure 27 Explaining the proposed solutions for on how to mirror the effect of the asynchronous reset behaviour on a DUT in the Verification Environment

After discussing with an [experienced individual](#) in the field the 3 solutions I came up with, I discovered that the clearing of the expected sequence fifo is the endorsed solution by most of the engineers working in the field.

5.4.4 Asynchronous reset in pipelined designs & verification environments (Cont.)

▪ Clearing the expected Sequence Item FIFO (Cont.)

But there is an issue that can occur with this solution, it is as demonstrated below:

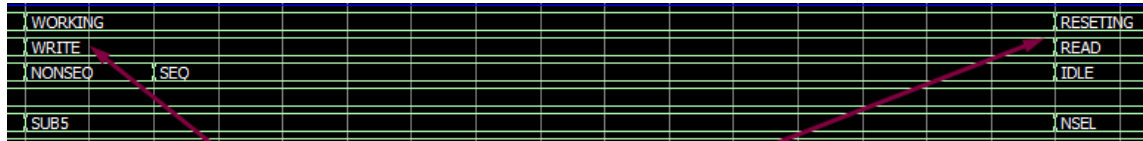


Figure 28 Asynchronous reset behaviour in the dut

As demonstrated above, **the reset is asserted when the past cycle was a write cycle**. The write transaction is sent to the predictor at the cycle before the reset, and **the predictor acts accordingly (writes in the subordinate memory in the local subordinate ROM if it's a write control/data command)**. On the other hand, at the DUT, the address phase is sent, but the data phase is replaced by the reset, therefore preventing the write into the actual ROM of the DUT. **That is when my predictor runs a function to re-write the respective subordinate's ROM location with the past value it had to avoid mismatch between the DUT's ROM and the Predictor's.**

NOTE: This is why I implemented the design/verification environment with ROMs and not RAMs; to challenge myself to find a solution.

5.5 Waveforms

5.5.1 runall_waited_test waveforms

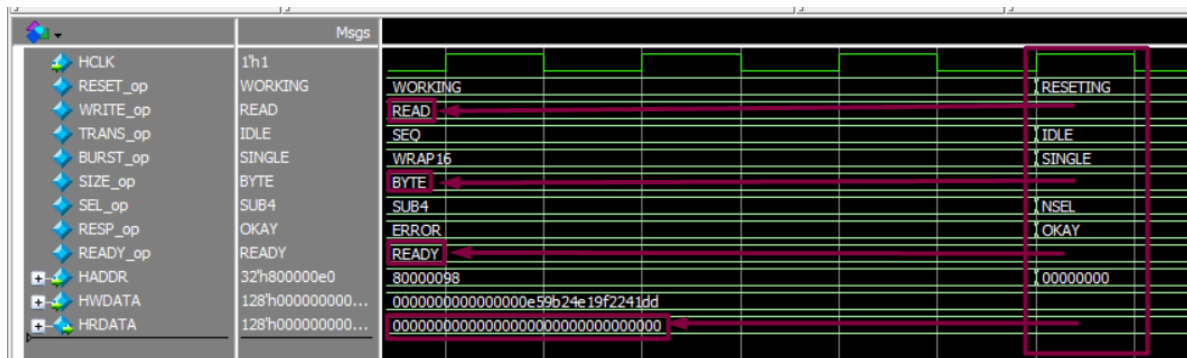


Figure 29

Values before & After RESET is asserted

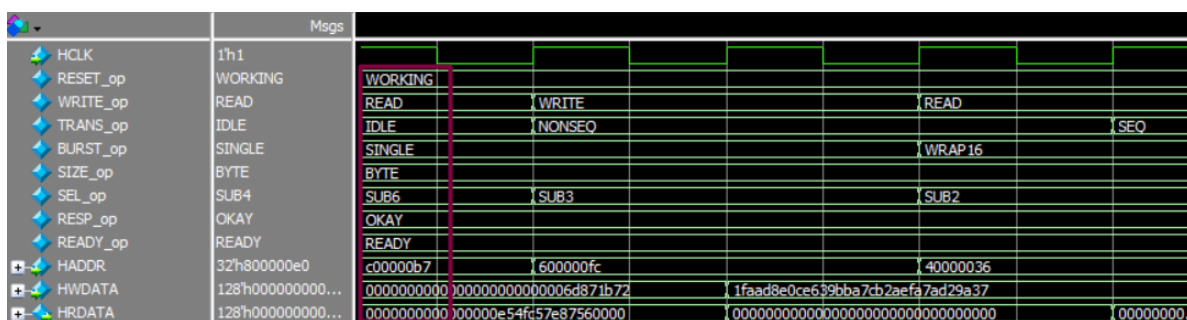


Figure 30

Values during IDLE transfer

5.5.2 runall_waited_test waveforms (cont.)

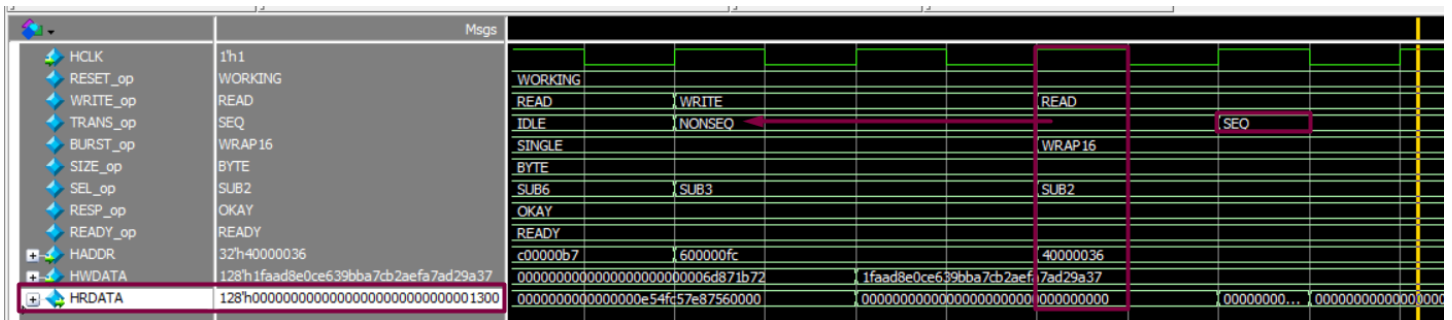


Figure 31
Values during WRAP16 Burst Transfer

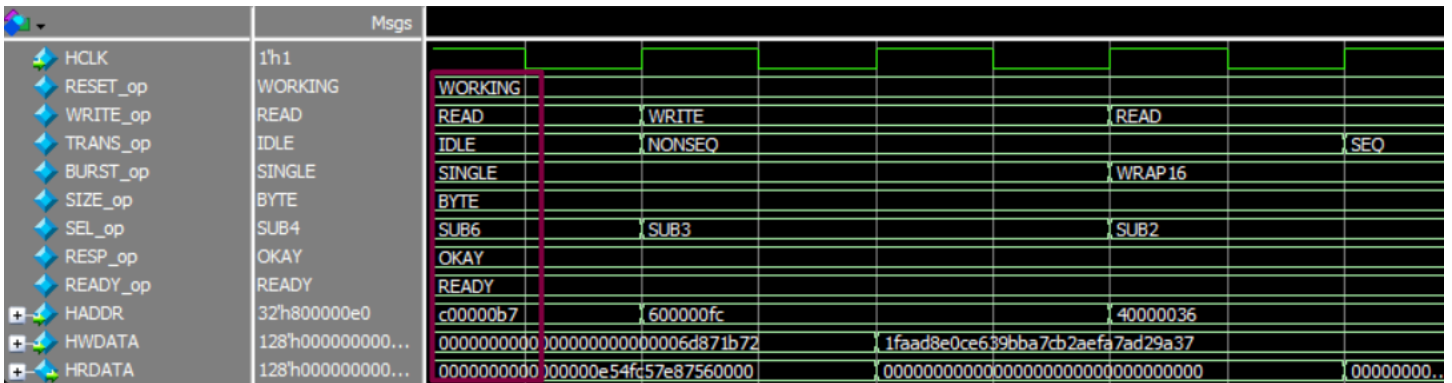


Figure 32
Values during INCR Burst Transfer

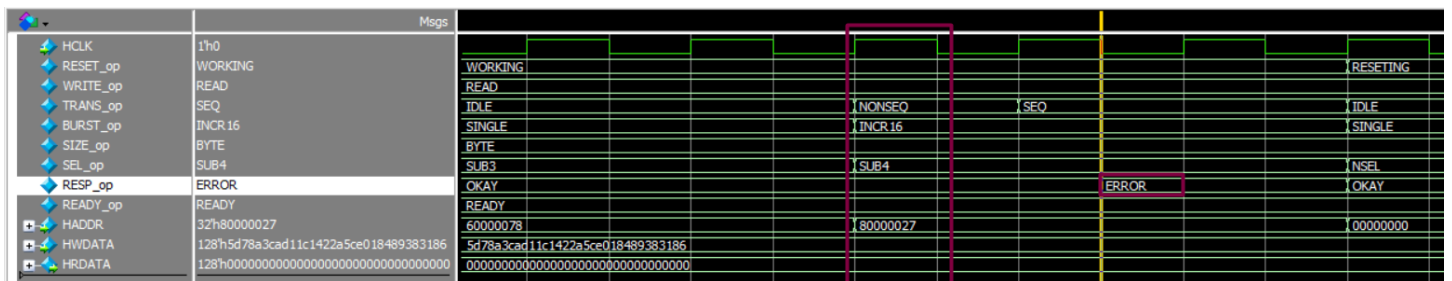


Figure 33
Values during accessing default subordinate (SUB4) & the resulting error

5.5.2 runall_test waveforms

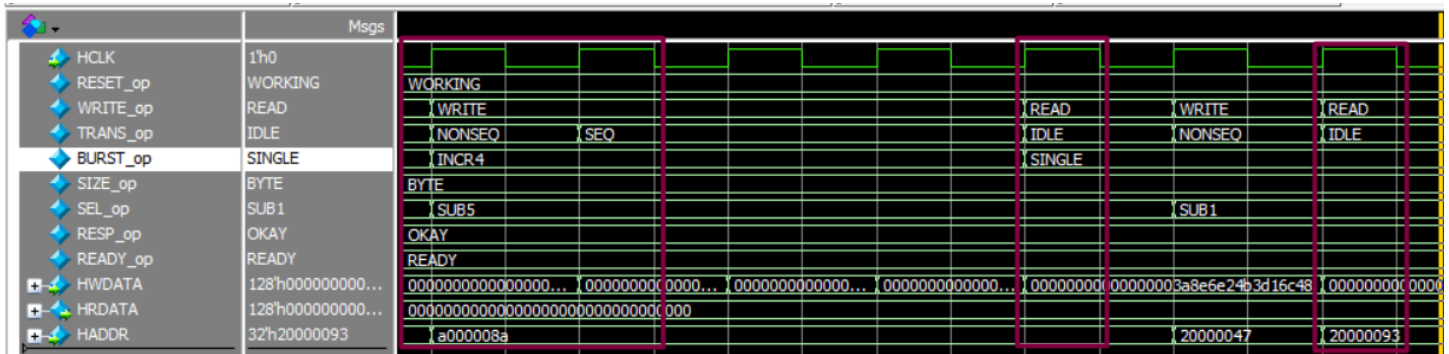


Figure 34
Values during INC4 Write Burst followed by a SINGLE Write transfer

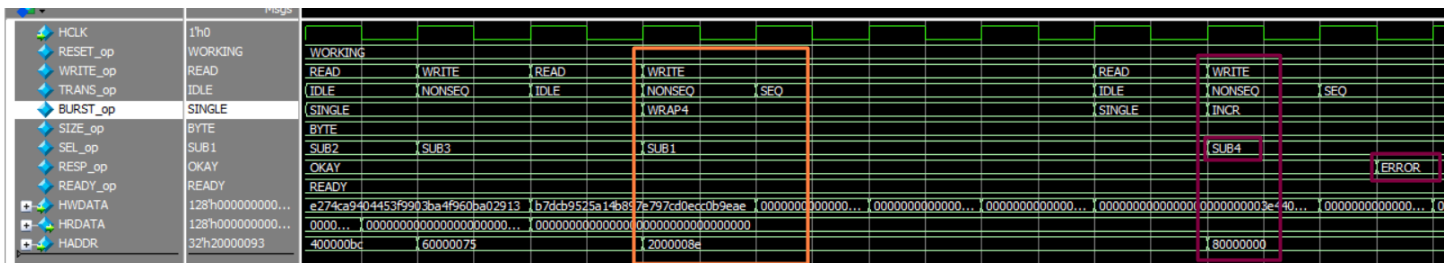


Figure 35
Values during WRAP4 Write Burst followed by an INCR Write transfer to default subordinate (SUB4) resulting in ERROR

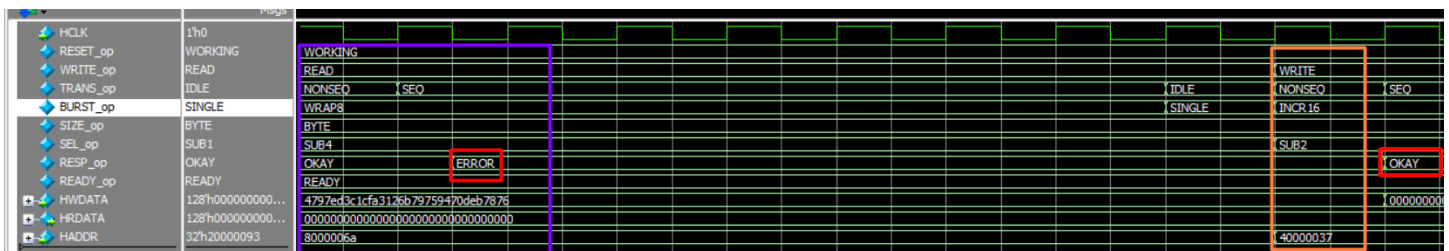


Figure 36
Values during WRAP8 Read Burst transfer to default subordinate (SUB4) resulting in ERROR followed by an INCR16 Write transfer

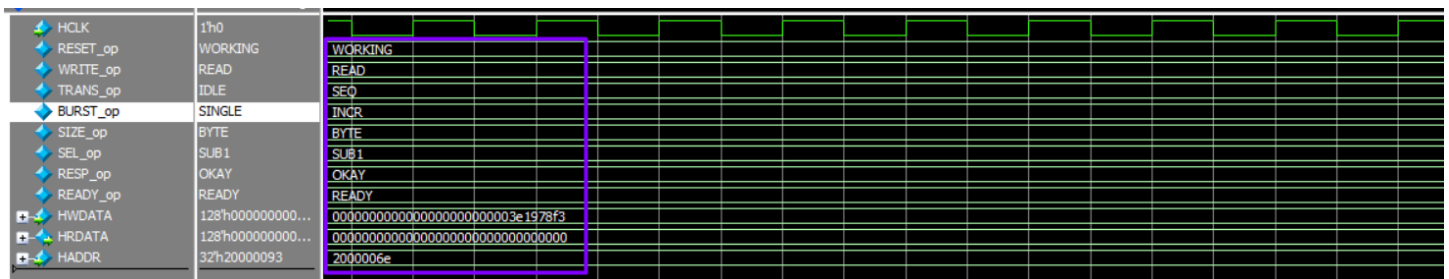


Figure 37
Values during INC4 Read Burst transfer

5.6 Assertion Report

Feature	Assertion
The reset assertion duration should at least be 15 clock cycles .	<code>reset_duration_assert</code>
When reset is asserted , no subordinates should be selected, i.e. (HADDR = 0)	<code>reset_addr_assert</code>
When HTRANS == IDLE , HREADY response should always be HIGH	<code>idle_ready_assert</code>
When HTRANS == IDLE , inputs (HSIZE , HBURST , HWRITE) should all be 0	<code>idle_inputs_assert</code>
When a burst transfer is issued, it must be followed by an IDLE transfer	<code>incr4_idle_assert</code> <code>incr8_idle_assert</code> <code>incr16_idle_assert</code> <code>wrap4_idle_assert</code> <code>wrap8_idle_assert</code> <code>wrap16_idle_assert</code>
When a burst transfer is issued, HTRANS must be == NONSEQ	<code>burst_trans_nonseq_assert</code>
When a burst transfer is issued, HTRANS must be == SEQ AFTER 1 CYCLE (except for INCR burst)	<code>burst_trans_seq_assert</code>

Table 4
Assertion Report

P.S. Check the coverage reports.

Commented [AY1]: @(posedge clk) \$fell(HRESETn) | => ##15 \$rose(HRESETn);

Commented [AY2]: @(posedge clk) ~HRESETn |-> HADDR == 0;

Commented [AY3]: @(posedge clk) HTRANS == 0 | => ##3 (HREADY);

Commented [AY4]: @(posedge clk) HTRANS == 0 |-> (HSIZE == 0) && (HBURST == 0) && (HWRITE == 0);

Commented [AY5]: @(posedge clk) ((\$rose(HBURST[0]) || \$rose(HBURST[1]) || \$rose(HBURST[2]) || \$fell(HBURST[0]) || \$fell(HBURST[1]) || \$fell(HBURST[2])) && (HBURST == 3)) |-> ##4 (HTRANS==0);

Commented [AY6]: @(posedge clk) ((\$rose(HBURST[0]) || \$rose(HBURST[1]) || \$rose(HBURST[2]) || \$fell(HBURST[0]) || \$fell(HBURST[1]) || \$fell(HBURST[2])) && (HBURST == 5)) |-> ##8 (HTRANS==0);

Commented [AY7]: @(posedge clk) ((\$rose(HBURST[0]) || \$rose(HBURST[1]) || \$rose(HBURST[2]) || \$fell(HBURST[0]) || \$fell(HBURST[1]) || \$fell(HBURST[2])) && (HBURST == 7)) |-> ##16 (HTRANS==0);

Commented [AY8]: @(posedge clk) ((\$rose(HBURST[0]) || \$rose(HBURST[1]) || \$rose(HBURST[2]) || \$fell(HBURST[0]) || \$fell(HBURST[1]) || \$fell(HBURST[2])) && (HBURST == 2)) |-> ##4 (HTRANS==0);

Commented [AY9]: @(posedge clk) ((\$rose(HBURST[0]) || \$rose(HBURST[1]) || \$rose(HBURST[2]) || \$fell(HBURST[0]) || \$fell(HBURST[1]) || \$fell(HBURST[2])) && (HBURST == 4)) |-> ##8 (HTRANS==0);

Commented [AY10]: @(posedge clk) ((\$rose(HBURST[0]) || \$rose(HBURST[1]) || \$rose(HBURST[2]) || \$fell(HBURST[0]) || \$fell(HBURST[1]) || \$fell(HBURST[2])) && (HBURST == 6)) |-> ##16 (HTRANS==0);

Commented [AY11]: @(posedge clk) ((\$rose(HBURST[0]) || \$rose(HBURST[1]) || \$rose(HBURST[2]) || \$fell(HBURST[0]) || \$fell(HBURST[1]) || \$fell(HBURST[2])) && (HBURST != 0)) |-> ##[0:1] (HTRANS == 2'b10);

Commented [AY12]: @(posedge clk) (HBURST != 0) && (HBURST != 1) && (HTRANS == 2'b10) | => (HTRANS == 2'b11);

6 Future Work

6.1 Verification Wise

- Adding a Virtual Sequencer & Virtual Sequence
 - Using the virtual sequence to run different sequences instead of the nested sequences (which are currently in use).
- Removing the usage of Implicit uvm feedback between Driver & Sequencer/Sequence for reactivity and using the output monitor & virtual sequencer to implement reactivity within the virtual sequence (using grab method) and the reactive agent. ([Example](#): I edited this example is into to a project which [Youssef Nasser](#) and I worked on during our Si-Vision Academy Training because it already used virtual sequence/p_sequencer.etc, Check the alu_monitor_out class & the random_v_seq class, Error_detected/Error_detect TLM Ports/Exports to better understand the example).
- Adding a reset agent to truly test asynchronous reset behaviour & adding a separate reset interface
- Including the Error Injection within the runall_sequence & runall_waited sequence or whatever replaces them and adjusting the predictor/comparator accordingly if needed.
- Improve Assertions (SVA).

6.1.1 Future UVM hierarchy Diagram

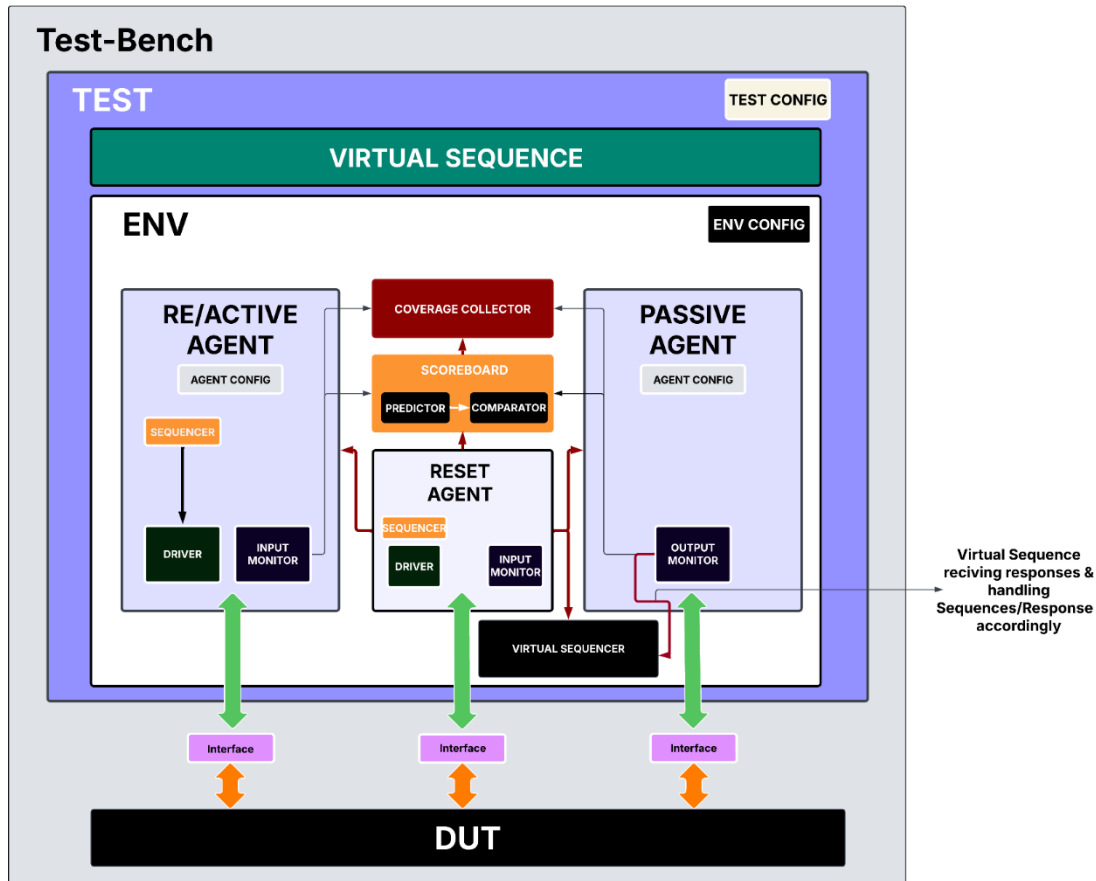


Figure 38 Future UVM Hierarchy diagram

6.2 Design Wise

- Fix the design so to fit the AHB standard of it being a single edge protocol
- Add more AHB5 features (RETRY RESPONSE & more of the HPROT functionality aside from the already implemented Privilege) and adjust the verification environment accordingly as well

6.3 Scripting Wise

- Make a separate bash script that asks for the **test name & HWDATA WIDTH & HADDR WIDTH** to be input by the user then compile and run the test accordingly.
- Make a Makefile