AMBA Advanced high-performance bus lite

Advanced Microcontroller Bus Architecture (AMBA) overview

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.

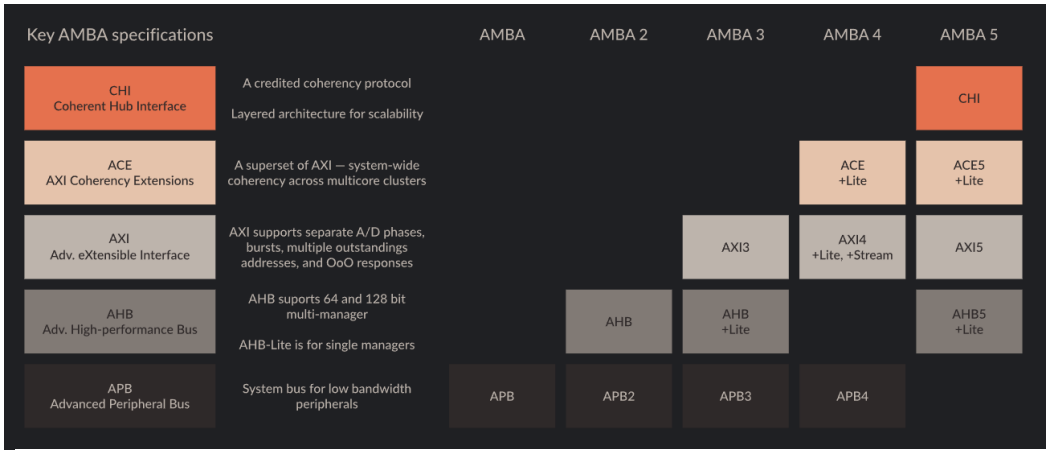


Figure 1: AMBA versions, components & specifications

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

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.

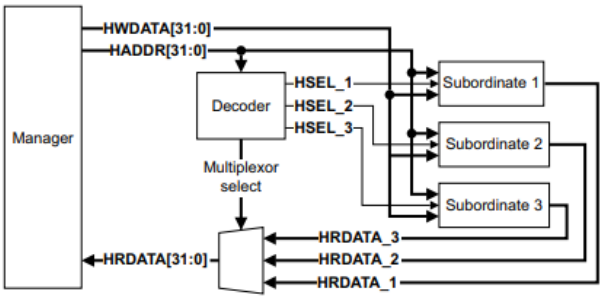


Figure 2
AMBA AHB lite

AHB lite bus constituents:

- Managers
- Subordinates
- Address decoders
- Multiplexors

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.

AMBA AHB lite Design components

1) Manager

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

- **Control Signals**
- **Data Signals**

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

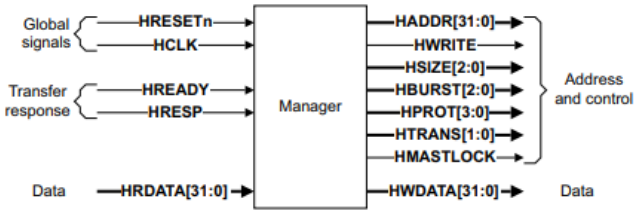


Figure 3
AMBA AHB Manager

2) Subordinate

A subordinate receives transfers from the manager and responds accordingly:

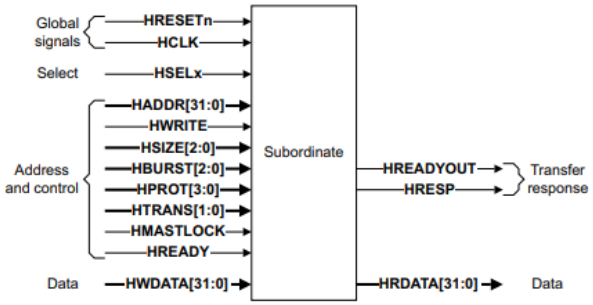


Figure 4
AMBA AHB Subordinate

The Subordinate signals back to the Manager:

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

3) Interconnect

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

In 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.

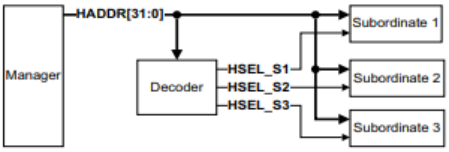


Figure 5
AMBA AHB lite decoder

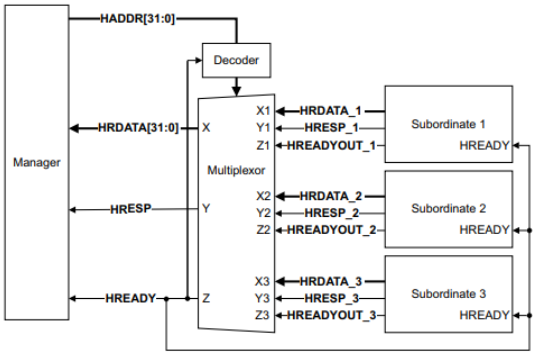


Figure 6
AMBA AHB lite Multiplexor

- **Multiplexor**

  The AHB protocol utilizes a read data multiplexor. After the Manager sends the address and control signals to all the Subordinates, with the decoder selecting the appropriate Subordinate during the data phase of the transfer. Any response data from the selected Subordinate, passes through the read data multiplexor to the Manager.

- **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.

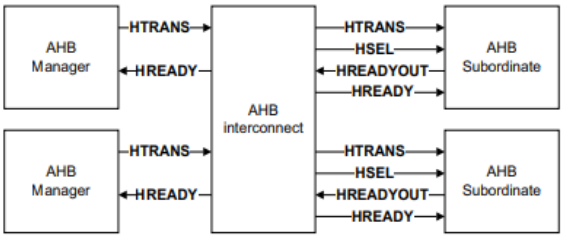


Figure 7
AMBA AHB Interconnect

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.

AMBA Advanced high-performance bus lite

Design of a re-configurable synthesizable pipelined AMBA AHB LITE

Design constituents:

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

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.

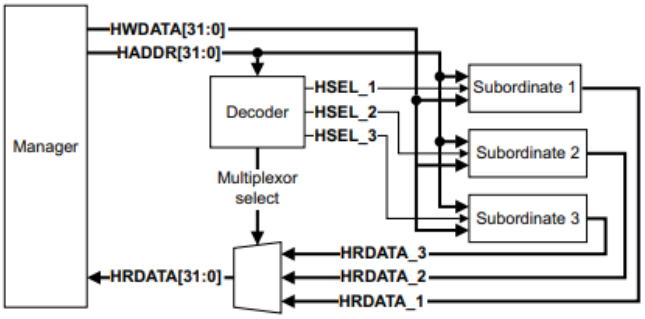


Figure 8
AMBA AHB lite depicting the connections between different subordinates & the manager

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**.

AMBA Advanced high-performance bus lite

AMBA AHB-lite test-plan

TESTS	DESCRIPTION	NO. BUGS FOUND
1. reset_test	: Asserting reset for 15 clk cycles and checking the outputs	Passed
2. IDLE_test	: Randomizing stimulus and driving **HTRANS = IDLE**	Passed
3. WRITE_SINGLE_test	: Randomizing stimulus and **READ** from the AHB subordinates **HBURST = SINGLE**	Passed
4. READ_SINGLE_test	: Randomizing stimulus and **WRITE** to the AHB subordinates **HBURST = SINGLE**	Passed
5. WRITE_INCR_test	: Randomizing stimulus and driving a **WRITE** with **HBURST = INCR (randomized INCR length)**	Passed
6. READ_INCR_test	: Randomizing stimulus and driving a **READ** with **HBURST = INCR (randomized INCR length)**	Passed
7. WRITE_READ_INCR_test	: Randomizing stimulus and driving a **WRITE then READ** with **HBURST = INCR (randomized INCR length)**	Passed
8. WRITE_WRAP4_test	: Randomizing stimulus and driving a **WRITE** with **HBURST = WRAP4**	Passed
9. READ_WRAP4_test	: Randomizing stimulus and driving a **READ** with **HBURST = WRAP4**	Passed
10. WRITE_READ_WRAP4_test	: Randomizing stimulus and driving a **WRITE then READ** with **HBURST = WRAP4**	Passed
11. WRITE_INCR4_test	: Randomizing stimulus and driving a **WRITE** with **HBURST = INCR4**	Passed
12. READ_INCR4_test	: Randomizing stimulus and driving a **READ** with **HBURST = INCR4**	Passed
13. WRITE_READ_INCR4_test	: Randomizing stimulus and driving a **WRITE then READ** with **HBURST = INCR4**	Passed
14. WRITE_WRAP8_test	: Randomizing stimulus and driving a **WRITE** with **HBURST = WRAP8**	Passed
15. READ_WRAP8_test	: Randomizing stimulus and driving a **READ** with **HBURST = WRAP8**	Passed
16. WRITE_READ_WRAP8_test	: Randomizing stimulus and driving a **WRITE then READ** with **HBURST = WRAP8**	Passed
17. WRITE_INCR8_test	: Randomizing stimulus and driving a **WRITE** with **HBURST = INCR8**	Passed
18. READ_INCR8_test	: Randomizing stimulus and driving a **READ** with **HBURST = INCR8**	Passed
19. WRITE_READ_INCR8_test	: Randomizing stimulus and driving a **WRITE then READ** with **HBURST = INCR8**	Passed
20. WRITE_WRAP16_test	: Randomizing stimulus and driving a **WRITE** with **HBURST = WRAP16**	Passed

TESTS	DESCRIPTION	NO. BUGS FOUND
21. READ_WRAP16_test	: Randomizing stimulus and driving a **READ** with **HBURST = WRAP16**	Passed
22. WRITE_READ_WRAP16_test	: Randomizing stimulus and driving a **WRITE then READ** with **HBURST = WRAP16**	Passed
23. WRITE_INCR16_test	: Randomizing stimulus and driving a **WRITE** with **HBURST = INCR16**	Passed
24. READ_INCR16_test	: Randomizing stimulus and driving a **READ** with **HBURST = INCR16**	Passed
25. WRITE_READ_INCR16_test	: Randomizing stimulus and driving a **WRITE then READ** with **HBURST = INCR16**	Passed
26. ADDRESS_ERROR_INJECTION_test	:Overriding constraints **to inject an invalid address during a burst transaction to check error response.**	Passed
27. PRIVILEGE_ERROR_INJECTION_r_test	**: Overriding constraints to inject subordinate_p_r with a read operation without the correct HPROT value to check privilege error response**	Passed
28. 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
29. runall_test		
30. runall_waited_test		

Coverage Groups

Instance based cover groups

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

The next cover groups are made using an array of instances and are very applicable/easy to implement and use in functional toggling functional coverage:

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
```

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

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.

Sample2 at 2nd clock edge: HWDATA/vector = 0101 for example,

I	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

Sample3 at 3rd clock edge: HWDATA/vector = 1010 for example,

I	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.

1. 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
```

2. 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];
```

3. 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);
```

4. 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();
```

The previous cover group covered the **toggling of data frame values of the bits of HWDATA bus, this one** covers the **toggling data transition values of the bits of HWDATA** bus.

- **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
```

- **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
```

Coverage Groups (cont.)

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.

1. Making the 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
```

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

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

3. 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);
```

4. 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();
```

The previous cover group covered the **data transition values of HTRANS**, **this one** covers the **data frame values of the of HTRANS** bus.

- **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
```

Coverage Groups (cont.)

Instance based cover groups

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.**

Traditional 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 cover groups

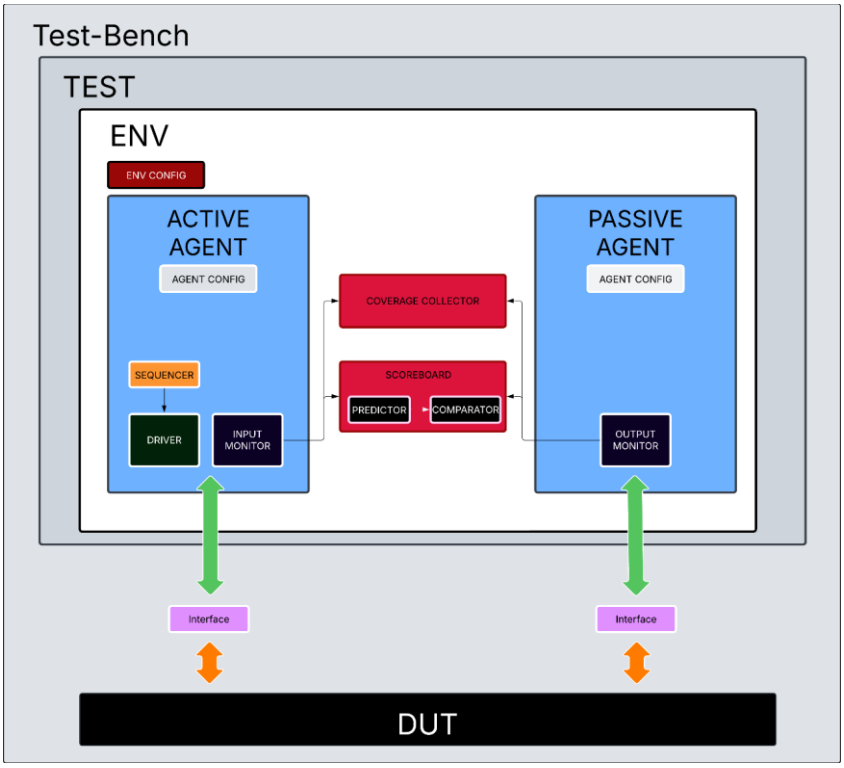
```
input_cov_copied = new();
RESET_covgrp        = new;
WRITE_covgrp        = new;
TRANS_covgrp        = new;
BURST_covgrp        = new;
SIZE_covgrp         = new;
SUBORDINATE_SELECT_covgrp = new;
ADDR_covgrp         = new;
HWDATA_covgrp       = new;
```

Sampling of the 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();
```

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

Starting with the uvm hierarchy:

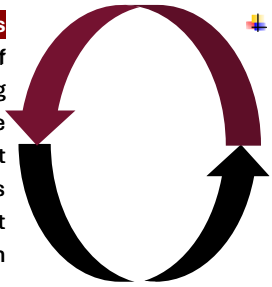


Issues faced during the verification process:

- **Mimicking pipelining.**

Reentrant tasks, events & fork-joins

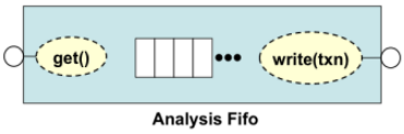
At first, I thought about making a mix of reentrant functions, tasks and utilising fork - joins to mimic the pipelining of the design. Initially, it seemed to work, but it requires a lot of resources and is less reliable to debug & use. Also, different simulators behave differently with some of the elements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



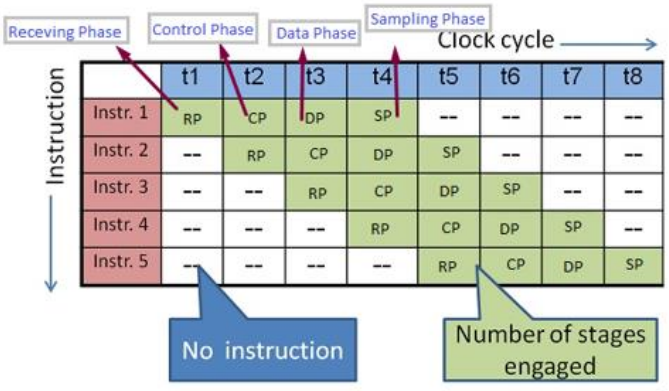
Always blocks, counters & flag control

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.

- **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.

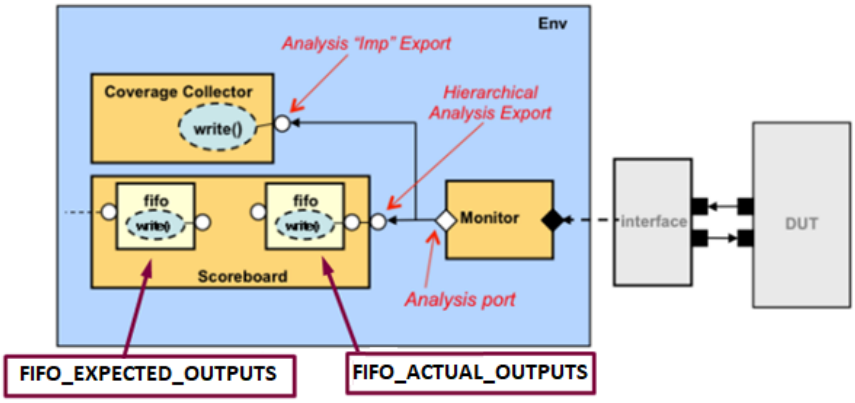


Analysis Fifo

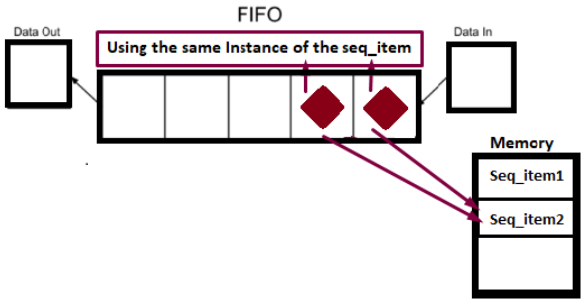


Issues faced during the verification process (cont.):

- **uvm_tlm_analysis_fifo overwriting (cont.)**



  - 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).

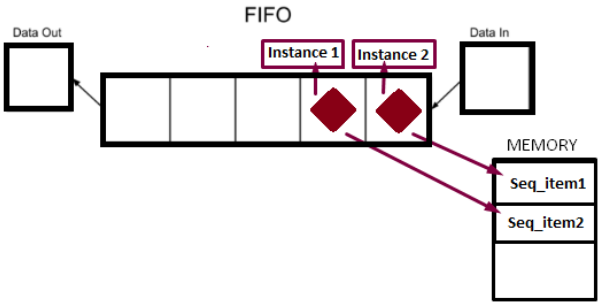


Issues faced during the verification process (cont.):

- **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.



- **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 turning 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
```

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));
  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
```

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.

P.S. The predictor_tr_counter is sometimes decremented during clearing the expected_seq_item_fifo when an asynchronous reset is asserted

Issues faced during the verification process (cont.):

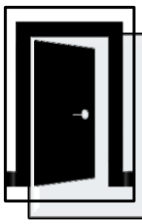
- **Asynchronous reset in pipelined designs & verification environments**

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?**

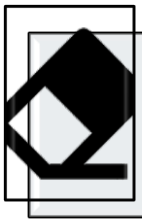
Delaying the Inputs Monitor & The Predictor

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

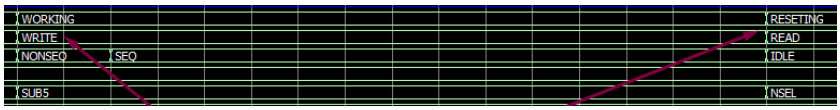
In this solution, No delay is required, but if a reset transaction is found, then the fifo of the expected sequence_item in the comparator is flused, also if one of the flushed items already **WRITTEN** a location in the subordinate memory, it reverts back to its original state through a function defined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

After discussing with someone 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 working engineers in the field.

Issues faced during the verification process (cont.):

Clearing the expected Sequence Item FIFO (Cont.)

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



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.**

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

Assertions REPORT

Feature	Assertion
The **reset assertion duration** should at least be **15 clock cycles**.	***reset_duration_assert***
When reset is asserted, no subordinates should be selected, i.e. **(HADDR = 0)**	**reset_addr_assert**
When HTRANS == IDLE, **HREADY** response should always be **HIGH**	**Idle_ready_assert**
When HTRANS == IDLE, inputs (HSIZE, HBURST, HWRITE) **should all be 0**	Idle_inputs_assert
When a burst transfer is issued, it must be followed by an **IDLE transfer**	**Incr4_idle_assert** **Incr8_idle_assert** **Incr16_idle_assert** **wrap4_idle_assert** **wrap8_idle_assert** **wrap16_idle_assert**
When a burst transfer is issued, **HTRANS** must be == **NONSEQ**	**burst_trans_nonseq_assert**
When a burst transfer is issued, **HTRANS** must be **== SEQ AFTER 1 CYCLE** (except for INCR burst)	**burst_trans_seq_assert**

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);