Module: SV for Verification
Section: Final Project **Task:** AHB3 Lite Protocol Verification

AHB3 LITE Protocol Verification - Project Link

Final Project

➢ Introduction:

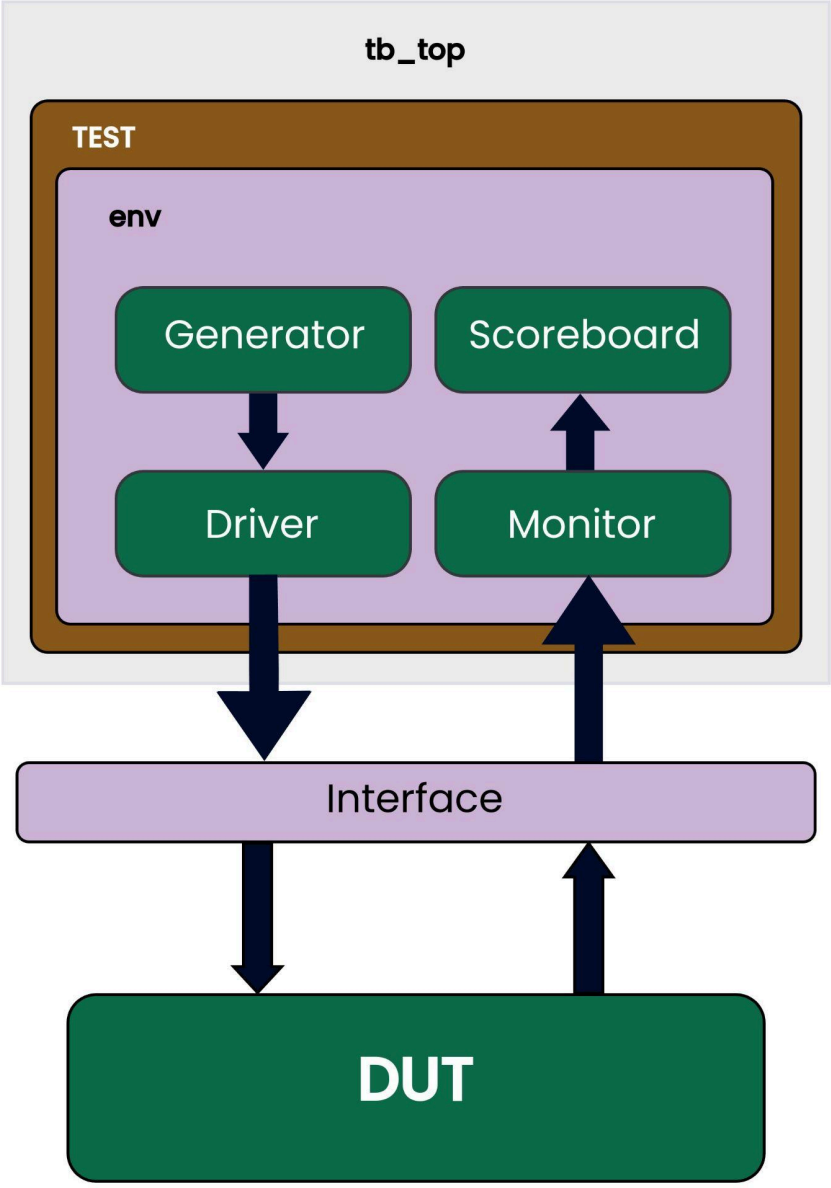
This project focuses on the verification of the AHB3 Lite Protocol, a streamlined version of the AMBA AHB protocol widely used in modern SoC designs. The document describes the development of a layered testbench to test various features of the **AHB3 Lite interface**, including burst transfers, address alignment, and different transfer sizes. Each layer of the testbench is structured to isolate and verify specific protocol features.

➢ Verification Environment:

Here's a concise description of each component of my layered testbench:

1. **Transaction**: Defines fields and signals required to generate stimuli.
2. **Interface**: Holds the design signals, allowing the testbench to drive and monitor DUT activities.
3. **Generator**: Creates and randomizes transactions, generating stimuli for the testbench, which is sent to the Driver.
4. **Driver**: Receives transactions from the Generator and drives the data into the DUT through the interface.
5. **Monitor**: Observes and captures interface activity, converting it into transaction data, which is then sent to the Scoreboard.
6. **Scoreboard**: Compares actual transaction data with expected values (from golden references or a reference model) to verify DUT behavior.
7. **Environment**: Acts as a container for grouping components like the generator, driver, monitor, and scoreboard.
8. **Test**: Creates the environment and manages the stimulus generation and driving process.
9. **Testbench Top**: The top-level module connecting the DUT and Test, with the interface linking the DUT and testbench components.

   Here's a block diagram of my Verification Environment:



➢ **Modifying the Testbench:**

Transaction:

- **Can I run predefined sequences (e.g., reset sequence, random write sequence, random read/write bursts, directed test)?**
  - Yes, predefined sequences can be implemented by creating specific transaction scenarios (e.g., reset sequences, directed tests). Sequences

can be customized using transaction parameters or constraints to generate both random and non-random data, such as random read/write bursts or a specific sequence for a directed test.

- **Can I debug easily if my test fails?**
  - Yes, debugging is easier when each transaction is isolated, allowing you to track individual reads/writes, address/data phases, and see where the mismatch occurred. Moreover, print statements can be used to track individual transactions and examine DUT's behavior.
- **Do I need one or multiple transactions for bursts?**
  - It depends on the type of burst. For WRAP or INCR bursts, you can manage multiple transactions within a single burst using counters or a single transaction class that handles the entire burst.

Generator:

1. **How to control how many transactions get generated?**
   - The number of transactions can be controlled by adjusting the repeat count inside the generator.
2. **How do I generate non-random transactions when needed?**
   - You can disable randomization on specific fields using **`rand_mode(0)`** or directly set values inside the transaction.

Driver:

1. **Are the interface signals driven according to the spec?**
   - Yes, the DRIVER port ensures that the interface signals (e.g., HADDR, HWRITE, HWDATA etc) are driven based on the AHB3-Lite protocol specifications. The interface mapping ensures compliance with the spec.
2. **Does the transaction have proper address/data phases?**
   - Yes, the transaction class defines the correct address and data phases. This includes handling address alignment, and ensuring correct data transactions.
3. **Do I need to sample inputs to decide whether to drive outputs on the next clocking event?**
   - Yes, inputs like **`HREADY, HWRITE`** and **`HTRANS`** need to be sampled to determine whether the next operation can occur. This ensures the driver waits for valid signals before driving new data.

Monitor:

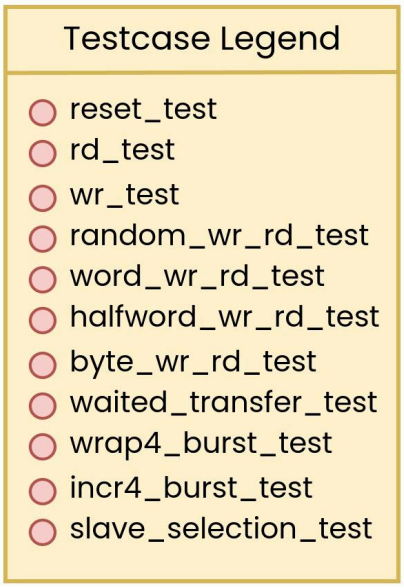
1. **Are the interface signals sampled according to the spec?**
    - Yes, the monitor observes and samples signals based on the AHB3-Lite protocol, ensuring accurate capture of transactions and protocol compliance.
2. **Does the transaction have proper address/data phases?**
    - Yes, the MONITOR port ensures the transaction follows the correct address/data phases and captures valid read/write cycles for comparison.

Scoreboard:

1. **Does the scoreboard implement proper endianness?**
    - Yes, I have properly implemented the Little Endian memory system.
2. **How to change endianness if required?**
    - In order to change the endianness, we can swap the byte ordering during data comparison.
3. **How to not compare reset values and to compare only those memory locations that have already been written?**
    - We can use additional flag to track valid memory writes before performing comparisons.
4. **Should scoreboard memory be static or dynamic?**
    - This depends on the test. If the memory size is fixed, static memory is simpler and efficient. For more flexible designs or larger memory, dynamic memory allocation allows more scalability.

➢ **Testplan Legend:**

Please refer to the testplan **Legend** below for the appropriate names for each testcase.



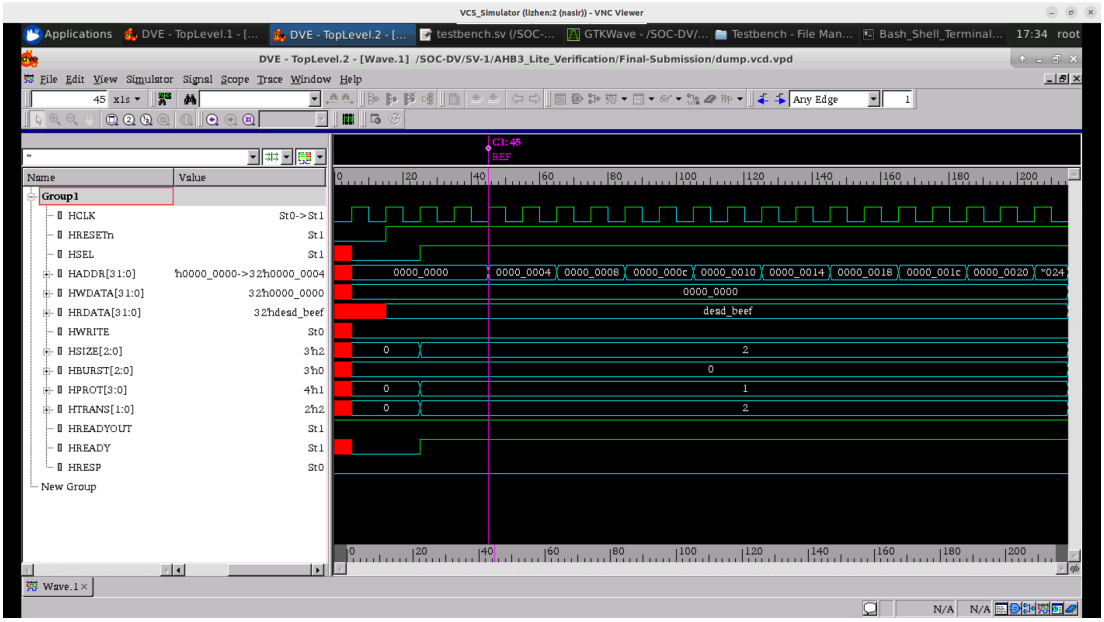
➢ **Simulations:**

You can run each testcase by uncommenting that particular testcase and running tcl script file named “run.do”, just enter the following command in the project directory for each testcase:

```
tclsh run.do
```

Here are the simulation results for each testcase.

1. **reset_test**



2. **rd_test:**

NOTE: You'd need to change the "memory" initialization file in design.sv and scoreboard.sv in order for default read to have better results.

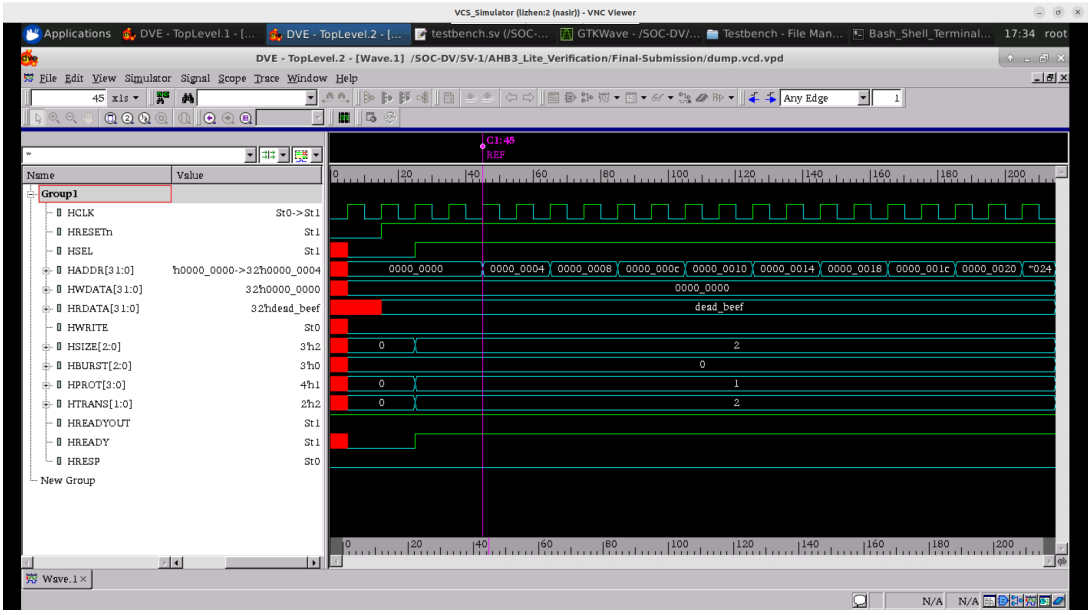
testbench.sv design.sv

```
//   Synthesizable (y/n) : Yes
//   Other               :
// -FHDR-------------------------------------------------------------

`include "ahb3lite_pkg.sv"
`include "rl_ram_1r1w.sv"
`include "rl_ram_1r1w_generic.sv"


module ahb3lite_sram1rw
import ahb3lite_pkg::*;
#(
  parameter MEM_SIZE          = 0,      //Memory in Bytes
  parameter MEM_DEPTH         = 256,    //Memory depth
  parameter HADDR_SIZE        = 32,
  parameter HDATA_SIZE        = 32,
  parameter TECHNOLOGY        = "GENERIC",
  parameter REGISTERED_OUTPUT = "NO",
//  parameter INIT_FILE         = "memory_init.mem"
  parameter INIT_FILE         = "rd_mem.mem"        // Only Uncomment for rd_test
)
(
  input                       HRESETn
```

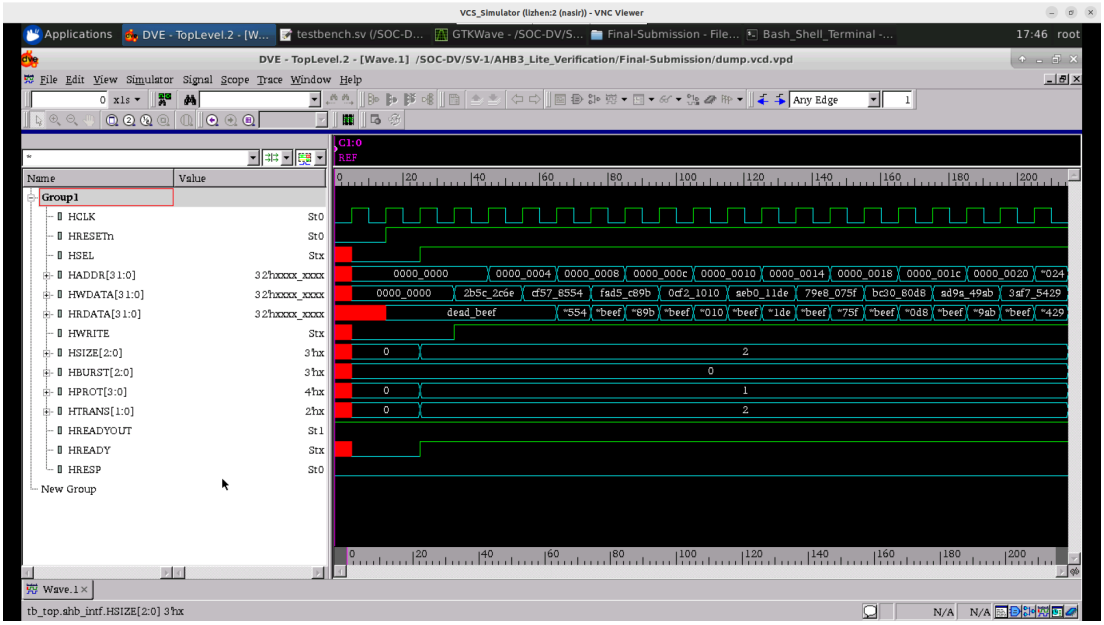
And for Scoreboard:

```
////////////////////////////////////////////////////////////////
//
// Constructor
//
function new(mailbox mon2sco);
  this.mon2sco = mon2sco;
  //$readmemh("memory_init.mem", mem);                    // Initialize Memory to a Known State

  /*
   *  Only Enable rd_mem.mem if you are trying to run rd_test.
   *
   *  "rd_mem.mem" is a memory initialized to randomly generated
   *  numbers so we can verify read operations.
   *
   *  [NOTE] :: Make sure to provide the DUT with the same
   *  Memory as Scoreboard for correct configuration. (INIT_FILE)
   */

  $readmemh("rd_mem.mem", mem);                         // For rd_test Only

endfunction
```

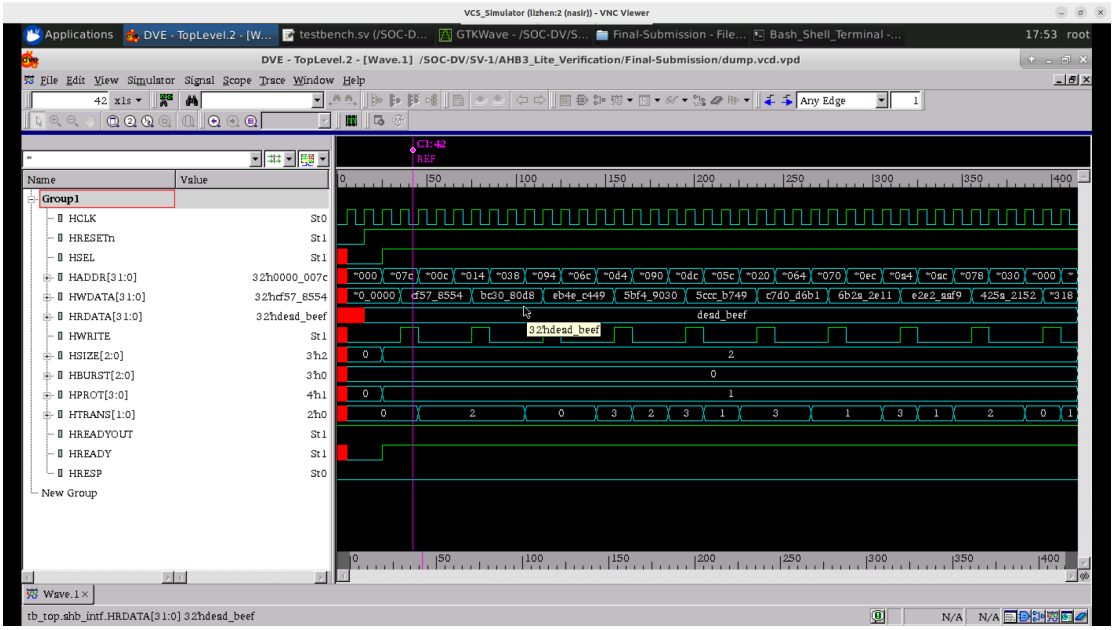
Here's the simulation:



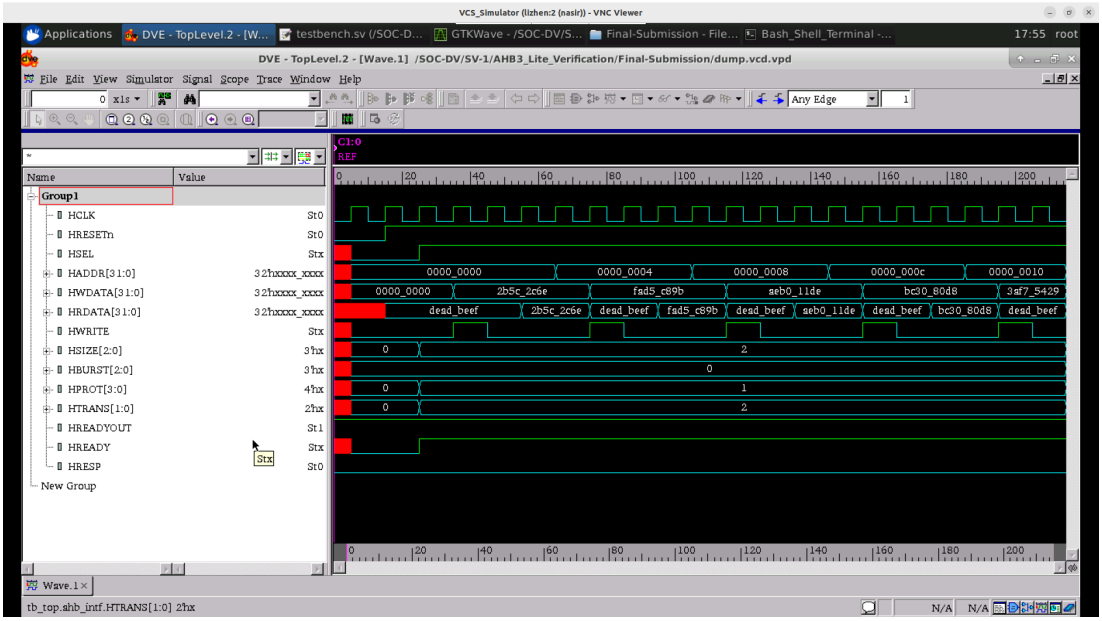
3. **wr_test:**
   Make sure to revert changes you made previously for **rd_test**.



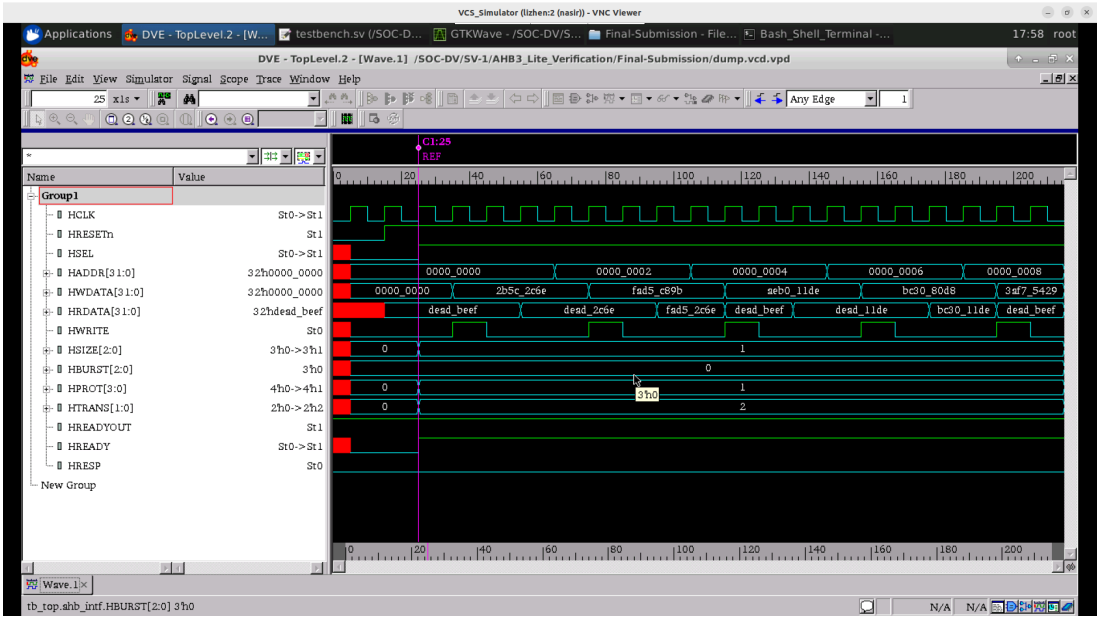
4. **random_wr_rd_test:**



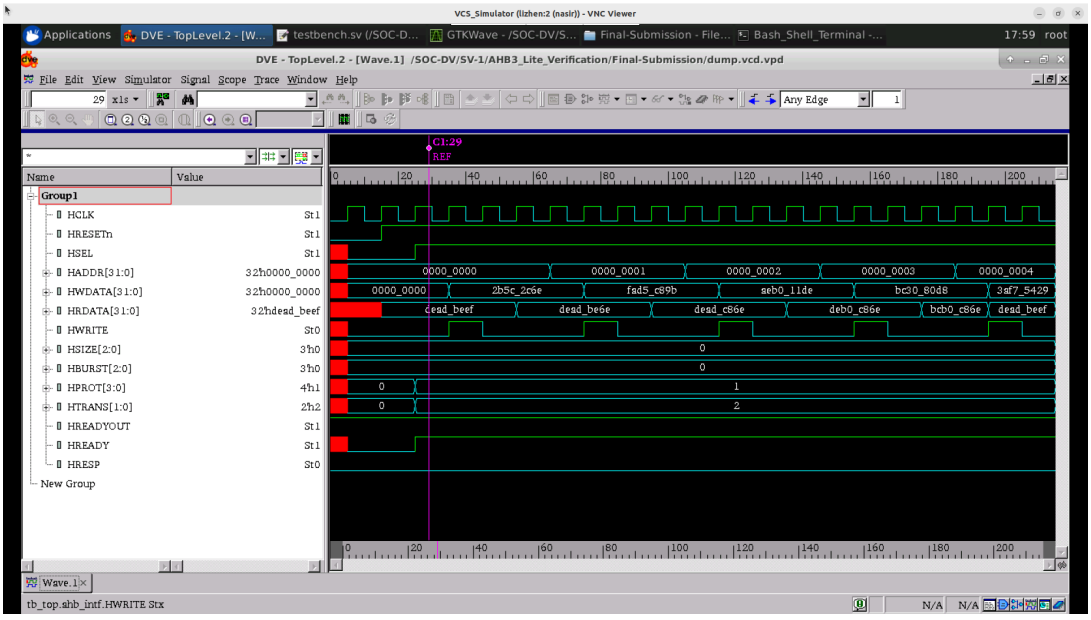
5. word_wr_rd_test:



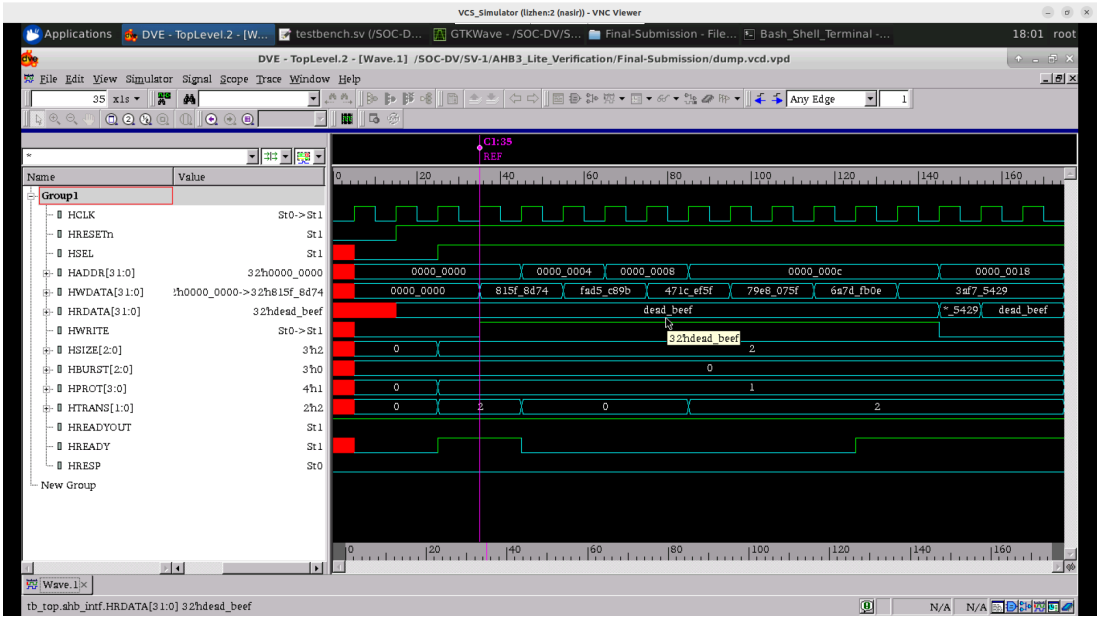
6. halfword_wr_rd_test:



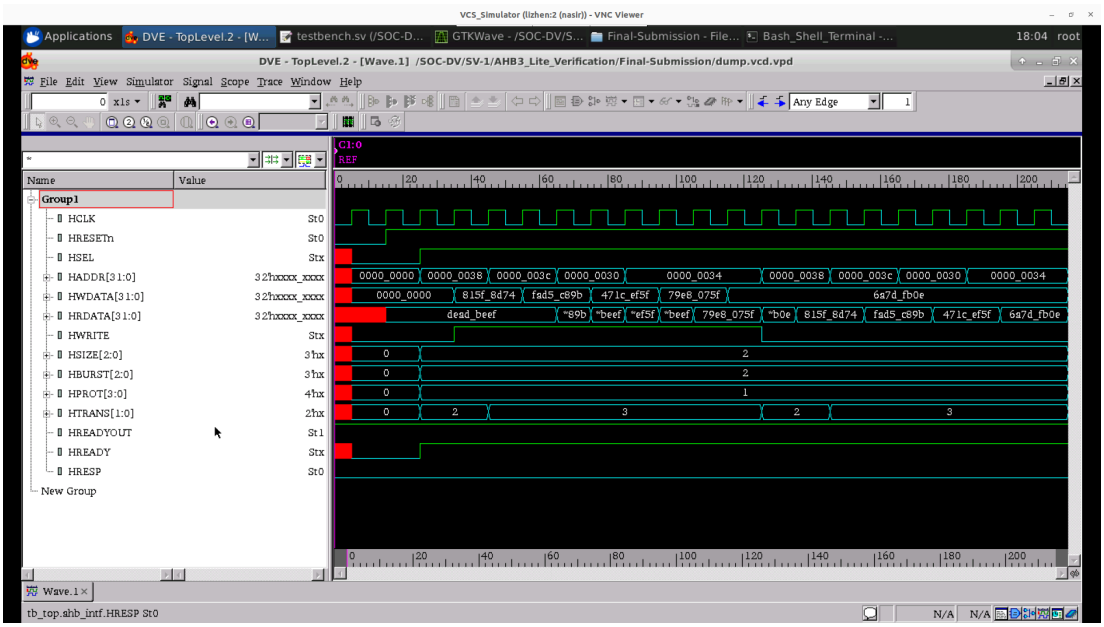
7. byte_wr_rd_test:



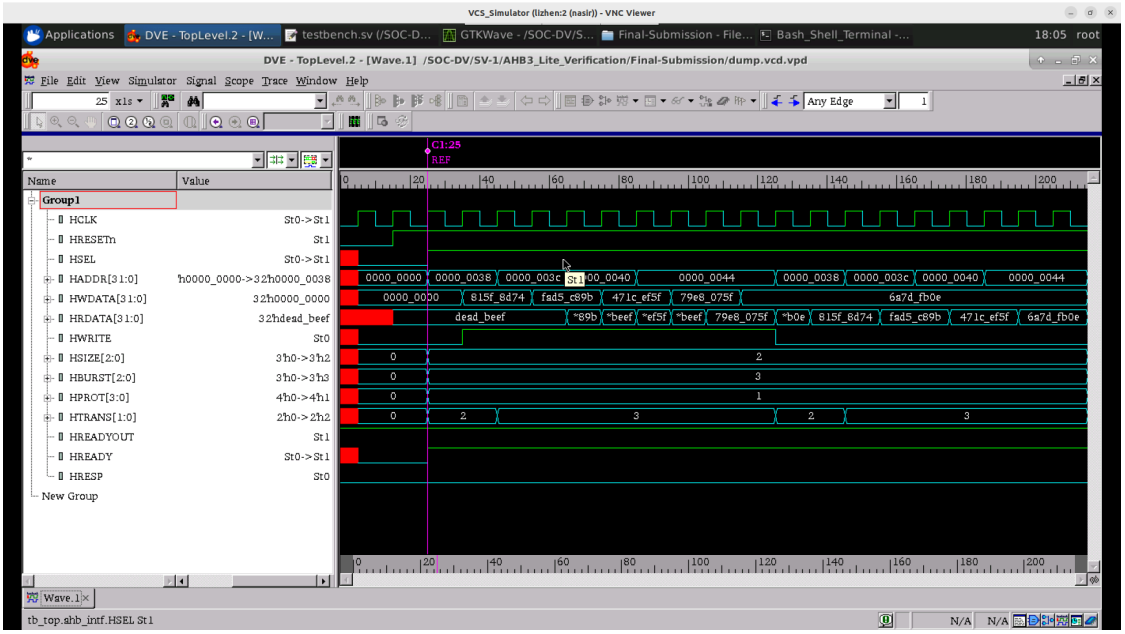
8. waited_transfer_test:



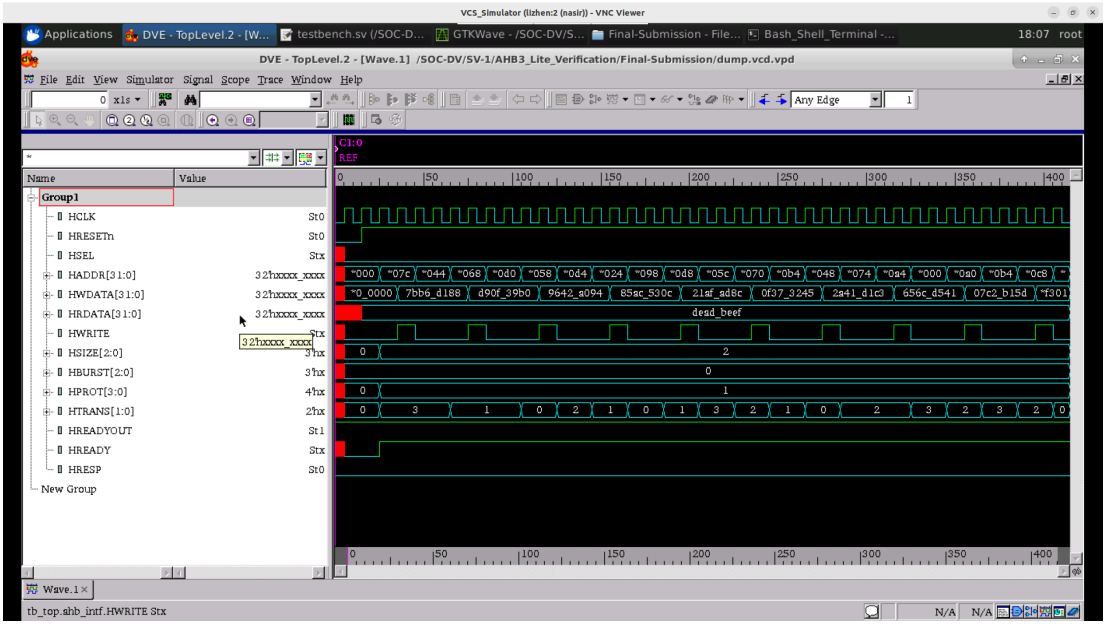
9. wrap4_burst_test:



10. incr4_burst_test:



11. slave_selection_test:



➢ **Reference Links:**

- [Tesptlan]
- [Code]