

Contents

▪ Basic UVM Test bench Architecture block diagram supporting Emulation.....	3
▪ The main criteria in developing UVM TB Emulation friendly	4
▪ Digital design flow	5
▪ Functional verification flow	6
▪ Computational storage project.....	8
○ Features.....	8
○ Architecture.....	8
○ Assumptions	8
○ Small computation storage module	9
○ Preloading	10
○ Testing	11
• Verification plan.....	11
• UVM test bench.....	12
• UVM TB environment in details	13
• Results	22
○ Comparing final memory contains with preloaded file	27
Figure 1 Simple UVM environment supporting Emulation.	3
Figure 2 Verifying each form of the design.....	7
Figure 3 Simple Verification process diagram	7
Figure 4 Solutions for low code & functional coverage.....	7
Figure 5 Computational storage architecture	8
Figure 6 Preloading script	10
Figure 7 UVM TB architecture	12
Figure 8 Scoreboard algorithm.....	18
Figure 9 Exporting final memory contains.....	27
Figure 10 Comparison script	27
Figure 11 Comparison result in case of large number of transactions	28
Figure 12 Comparison result in case of 200 transactions	28

■ Basic UVM Test bench Architecture block diagram supporting Emulation

To accelerate the performance of the simulation, we need to utilize some FPGA-based hardware that consumes less time than general-purpose CPUs. That's the emulation.

The communication between the hardware "emulator" and the software "simulator" is defined by SCE-MI, which is a standard clarifying how the test bench running on the simulator can communicate with the design running on the emulator.

There are three ways to perform this communication: macro-based, function-based, and pipe-based. Each with pros and cons. Function-based means that the simulator can call synthesizable functions inside the design running on the emulator, which means the pure simulation test bench needs to be modified to do this job.

UVM testing supporting emulation mainly consists of two separate domains: the test bench domain (software) and the HDL domain (hardware), which communicate with each other to do the job. The monitor and the driver are split into two parts: untimed (to be placed in the software domain and called proxy) and timed (to be placed in the hardware domain and called BFM).

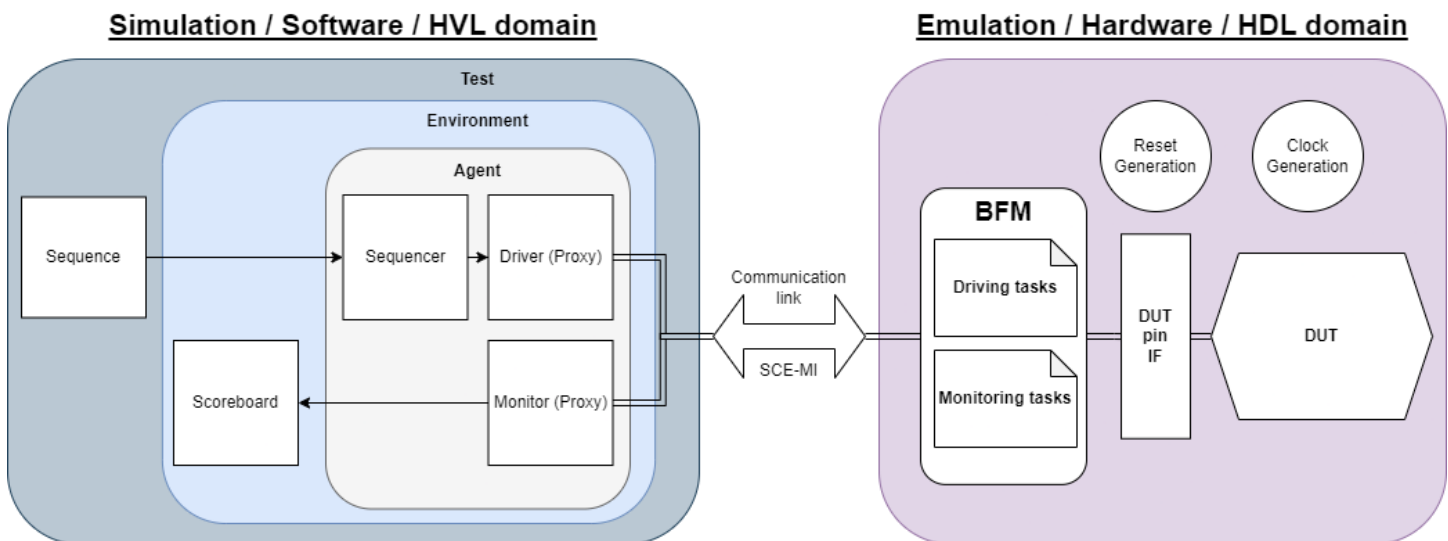


Figure 1 Simple UVM environment supporting Emulation.

- The main criteria in developing UVM TB Emulation friendly

As I mentioned in the previous section, the two main domains used in co-emulation are completely separated. The simulation, or HVL domain, consists of non-synthesizable testbench code separated from the synthesizable code running on the emulator. So, the code is analyzed, and each piece is assigned to a suitable domain.

Any timing statements must be moved from the simulation domain to the hardware domain to avoid affecting the emulation performance. The communication link between both domains transmits data in the form of packets, which means there are no cross-domain signals. The BFM in the hardware domain is used to establish indirect communication between the simulator and the DUT (this communication may affect performance, so it needs to be "infrequent and data-rich").

When the simulator wants to execute time-related behavior, it will call some tasks implemented inside the BFM to do the job (as the communication between the simulator and the DUT became indirect, as I mentioned). The clock and reset generation are also added to the hardware domain, although they are not synthesizable (some guidelines are used to handle this problem). If there are any shared parameters between both domains, they are declared as shared parameters and packed.

The UVM TB must be optimized so that the performance is maximized. As I mentioned, the communication link between both domains may affect the performance if it's busy, so it's better to reduce and remove unnecessary transactions. Also, to split the TB into two domains, the written code needs to be written in standard form to facilitate the process.

■ Digital design flow

Starting with the design specifications, the digital design engineer starts understanding the required design and creating the initial structure. This structure is then mapped to the corresponding HDL, using HDLs like Verilog or System Verilog. The HDL is then synthesized into some technology-based cells (gates, FFs, and other logic components), creating a gate-level netlist. At each stage, it's a must to verify the current form of the design, and if there are any problems or bugs, the design engineer must modify the design.

To verify the timing, a static timing analysis is performed on the design to check the timing requirements and make sure that there are no violations.

Adding some extra hardware called DFT aims to facilitate the testing of the chips after the fabrication process. Fabricating millions of transistors may result in defective chips; many structural problems may occur, like "stuck at" when a certain node is stuck at 1 or 0. DFT facilitates the process of structural testing. Since extra HW is added, we need to verify that the new form of the design is equivalent to the previous one; formal verification will check this equivalence.

The next step is floor planning, where the location of the cells is determined. There are some "DRC" checks that need to be met for a successful manufacturing process; these checks are specified by the fabrication foundry. Power and area are also considered in this step.

After floor planning, we place the cells in their locations and connect them (route) in a proper way to meet the timing requirements and achieve a suitable area. At this point, we include wiring delays in our calculations, and it's critical to make sure that the given form is functionally correct and meets the timing requirements. Another formal verification is applied to check the equivalence.

After all these steps, if everything is okay, we can send the design to be fabricated, but that's not everything. As I mentioned, fabricating millions of transistors is not that easy, so we need to put a test plan to verify the functionality of the manufactured chips and make sure there are no problems with "stuck at" or wrong gate behavior or short circuits, or any other problems. We define a test model focusing on a certain problem and the corresponding patterns to be driven to the chip to detect any defects.

■ Functional verification flow

The goal of functional verification is to make sure that the behavior of the current form of the design is as expected and matches the design specifications. When the specs are mapped to the corresponding RTL, the verification engineer must make sure that the RTL matches and works well. Generally, when moving from one form to another, we need to verify that the new form behaves the same way as the previous one. For example, after synthesizing the RTL, we need to verify that the provided GLN matches the RTL. Detecting bugs at early stages is very important and critical.

Starting from the specification document, the verification engineer starts understanding the design and the functionality and extracts the features representing the design and detects any conflicts in the specs and reports them. To verify the extracted features, we start writing a verification plan including the required tests and other details, to be followed by the verification engineer.

In the verification plan we include the verification technique, we have some options; simulation based, formal based and semi-formal. The simulation based consumes much time as we stimulate the design and monitor the response, we can accelerate the process utilizing emulation as mentioned in the first section. Another option is to use formal verification, mainly based on mathematical models to check the equivalence between the given design and a well designed and known golden model. When the formal proves, it does not mean that the design is correct, it means that both forms are only equivalent, and when the formal disproves it gives a counter example to guide the verification engineer to the problem.

As I mentioned, the tests are included in the verification plan, for each feature we need to know what stimulus to be driven to the DUT to hit this feature and under what conditions and what is the expected output.

The verification plan includes the coverage target. After extracting the features, we need to set priorities for them as the TTM is very critical and the verification process takes too much time, the goal is to cover the most important features. To cover these features, we start generating random tests and feed them to the DUT and check the coverage percentage. If the target is not achieved yet, we may feed some directed tests to hit certain features to improve the coverage. Also changing the random seed may result in better coverage percentages. The coverage also includes code coverage to cover some statements, branches, conditions, FSM states and others.

The assertions are also important in the verification process, they ensure some temporal features in the design. System Verilog assertions are mainly divided into two types, the immediate (executed once) and concurrent (executed at each triggering event).

After writing the verification plan, we start developing the test bench and begin the process. We may reuse general legacy environments if they are available and suitable for the given design.

The following diagrams summarize the previous section.

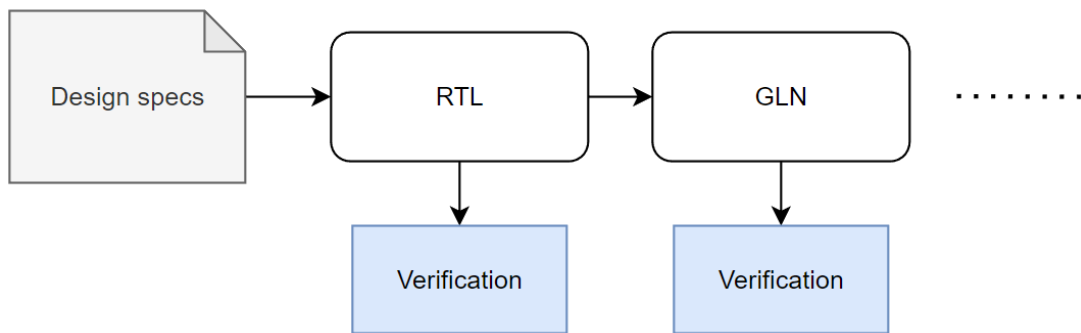


Figure 2 Verifying each form of the design.

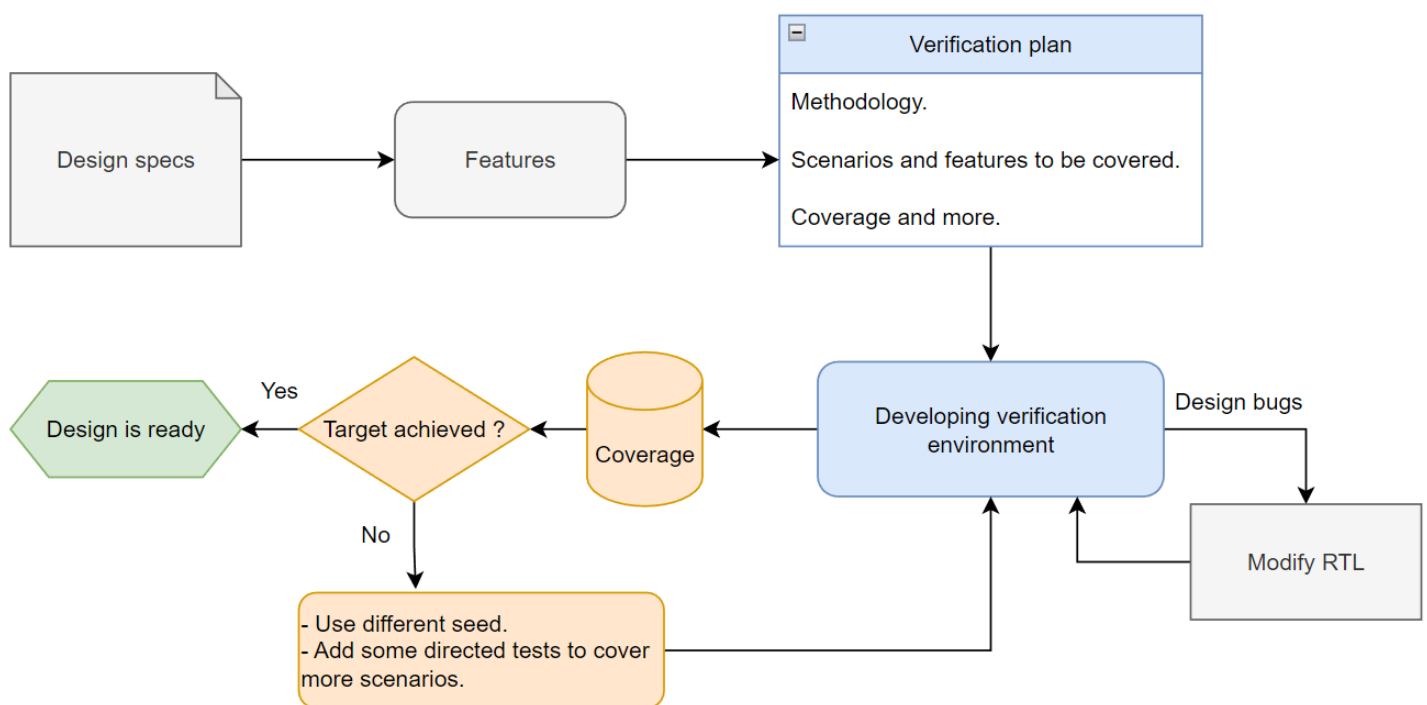


Figure 3 Simple Verification process diagram

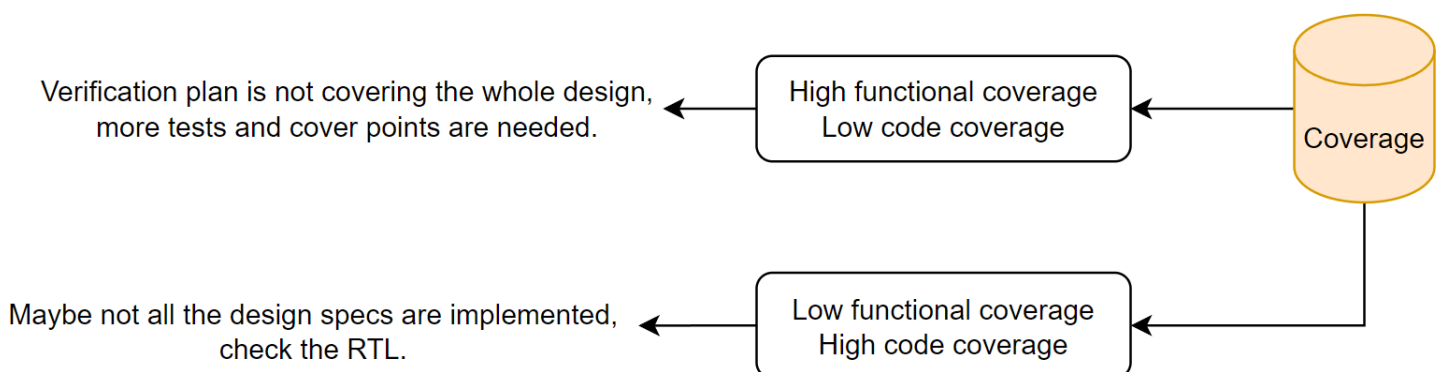


Figure 4 Solutions for low code & functional coverage

■ Computational storage project

○ Features

Reading stored values, writing in a certain location, performing addition and subtraction operations on the stored values, and saving the result in a given location.

○ Architecture

Starting with high level design, to implement such module, it's obvious that an adder and subtractor are needed to add and subtract the stored values in the given locations. The module needs to decode the given commands to behave accordingly, so the four different operations (read/write/add/subtract) will be encoded into (00/01/11/10) respectively. A two-dimensional array with parameterized width and depth will represent the required memory. The following diagram clarifies the architecture:

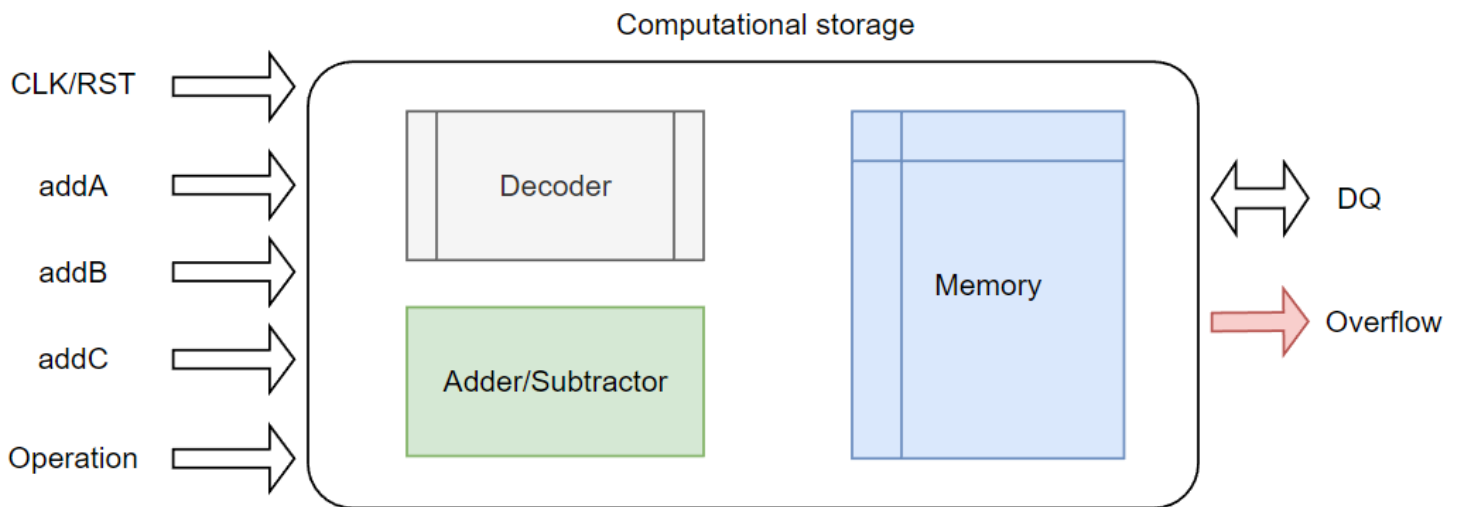


Figure 5 Computational storage architecture

○ Assumptions

- When resetting the design, I assumed that the memory will be loaded with the initial values again since this condition is not specified in the specs.
- I assumed that the memory width is 32bits and the depth is 1024.
- I added extra signal to detect the overflow, I assumed that it's needed by the CPU as such problem may cause many failures.

○ Small computation storage module

```
module CompStorage #(parameter WIDTH = 32, parameter DEPTH = 1024)
(
  inout [WIDTH-1:0] DQ,
  output overflow,

  input CLK,
  input RESET,
  input [$clog2(DEPTH)-1:0] addA,
  input [$clog2(DEPTH)-1:0] addB,
  input [$clog2(DEPTH)-1:0] addC,
  input [1:0] OPERATION
);

localparam RD_MEM_CMD = 2'b00,
           WR_MEM_CMD = 2'b01,
           ADD_CMD     = 2'b11,
           SUB_CMD     = 2'b10;

reg [WIDTH-1:0] DQ_temp;
/*
Defining temp variable to be used to detect the overflow.
*/
reg [WIDTH:0] tempResult;

//Memory
reg [WIDTH-1:0] MEM [0:DEPTH-1];
initial
begin
  $readmemh("MEM_PRE_HEX.txt", MEM);
end
```

```
//Procedurals
always @(posedge CLK or negedge RESET)
begin
  if(!RESET)
  begin
    $readmemh("MEM_PRE_HEX.txt", MEM);
  end
  else
  begin
    case (OPERATION)
      RD_MEM_CMD:
      begin
        DQ_temp <= MEM[addA];
      end

      WR_MEM_CMD:
      begin
        MEM[addC] <= DQ;
      end

      ADD_CMD:
      begin
        MEM[addC] <= MEM[addA] + MEM[addB];
        tempResult <= MEM[addA] + MEM[addB];
      end

      SUB_CMD:
      begin
        MEM[addC] <= MEM[addA] - MEM[addB];
      end
      default:
      begin
        DQ_temp <= MEM[addA];
      end
    endcase
  end
end
```



```
//Continuous assignments
/*
Drive "DQ" only during read operation.
*/
assign DQ = (OPERATION == 2'b00)? DQ_temp : 'bz ;
/*
Assert overflow flag when addition result overflows.
*/
assign overflow = (OPERATION == 2'b11 && tempResult[WIDTH])? 1'b1 : 1'b0;
endmodule
```

At every positive edge clock, the operation is decoded, and according to the operation, the DUT handles the process (writing in the memory in the case of a write operation, reading from the memory in the case of a read operation, adding or subtracting stored values, and saving the result inside the memory).

I developed the whole design in one module because it's simple

○ Preloading

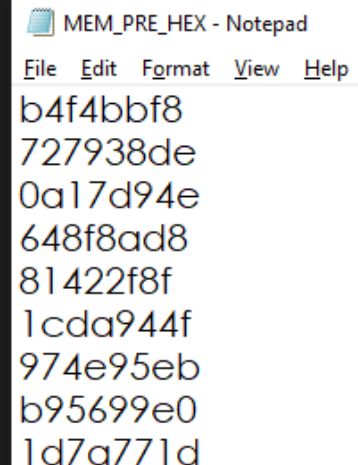
Using Python, I created a script to generate the memory preloading file. The script is simple, it takes the required width and depth and randomizes hex values to be saved inside a text file.

```
import secrets

Depth = 1024
Width = 32

#open text file
with open("MEM_PRE_HEX.txt", "w") as file:

    #write random hex values
    for index in range(Depth):
        file.write(secrets.token_hex(int(Width/8))+'\n')
```



```
MEM_PRE_HEX - Notepad
File Edit Format View Help
b4f4bbf8
727938de
0a17d94e
648f8ad8
81422f8f
1cda944f
974e95eb
b95699e0
1d7a771d
```

Figure 6 Preloading script

○ Testing

● Verification plan

- The extracted features are reading, writing, adding, and subtracting. The goal is to ensure that each of them works properly. I will construct a UVM TB to verify the functionality.
- For the verification metric “Coverage”, I will try to cover all possible scenarios (“four operations, different addresses, and others”), starting with pure random stimuli and then observing the results. If some cases are not covered, I will utilize directed testing to cover them.
- Sequences and corner cases:

Test	Required sequence	Expected behavior
Before any operations, check that the initial values stored inside the memory match the preloading file.	Drive some read operations at the beginning of the simulation.	The provided values match the preloading file values.
Resetting the DUT after some random operations to make sure it will return to its initial values.	Drive random operations then assert reset signal. Then drive read operations.	The DUT memory will return to the same initial values loaded at the beginning of the simulation.
Write a random value into a certain memory location and read that value in the following cycle.	Drive a certain address C, random value, and the “write” operation. Then drive the same address C and “Read” operation.	The DUT will store the given value in the given location and in the following cycle, the stored value will be provided.
Writing two values into certain locations then perform adding operation on them, then retrieving the stored result.	Drive two values into different locations. Then drive “Add” operation. Then read the stored result.	The result must be the addition of the two values.
Writing two values into certain locations then perform subtracting operation on them, then retrieving the stored result.	Drive two values into different locations. Then drive “Sub” operation. Then read the stored result.	The result must be the subtraction of the two values.
Adding two large values so that the addition operation will assert the overflow flag.	Add two large values and observe the overflow flag.	The overflow flag is asserted.
Adding two values and storing the result in the same location as one of the operands.	Drive the two locations of certain operands, and the third location is the same as one of them.	The stored result will be as expected.

- UVM test bench

After developing the verification plan, it's UVM time. I built a UVM test bench with the following structure:

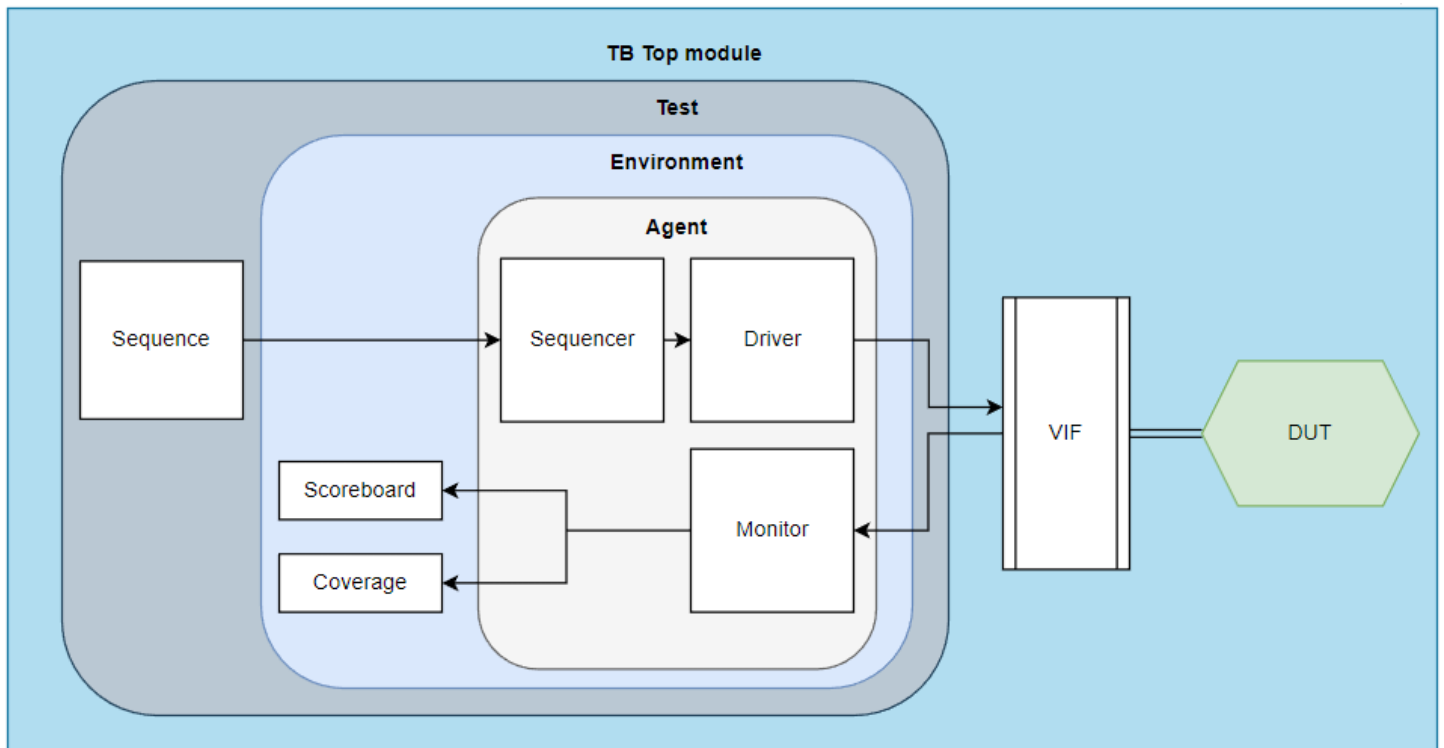


Figure 7 UVM TB architecture

Environment details in the following pages

- UVM TB environment in details

Interface and Sequence item:

```
interface CS_if (input logic CLK);

logic                                RESET;
logic  [$clog2(`DEPTH)-1:0]         addA;
logic  [$clog2(`DEPTH)-1:0]         addB;
logic  [$clog2(`DEPTH)-1:0]         addC;
logic  [1:0]                         OPERATION;

logic                                overflow;

/*
For the inout signal "DQ", i defined a temporary variable to facilitate
the driving mechanism, driving "DQ" @write operation only.
*/
wire  [`WIDTH-1:0]                  DQ;
logic  [`WIDTH-1:0]                  DQ_temp;

assign DQ = (OPERATION == 2'b01)? DQ_temp : 'bz ;

//Clocking block
clocking cb @ (posedge CLK);
    default input #1step output #0 ;

    output addA, addB, addC, RESET, OPERATION;

    input overflow;
    inout DQ;
endclocking : cb1

endinterface: CS_if
```

```
class CS_transaction extends uvm_sequence_item;

//Constructor
function new(input string inst = "transaction");
    super.new(inst);
endfunction : new

//Data members
bit                                RESET;

rand bit  [$clog2(`DEPTH)-1:0]     addA;
rand bit  [$clog2(`DEPTH)-1:0]     addB;
rand bit  [$clog2(`DEPTH)-1:0]     addC;

rand bit  [1:0]                     OPERATION;
rand bit  [`WIDTH-1:0]              DQ_temp;

bit      [`WIDTH-1:0]               DQ;
bit      overflow;

//Reg to Factory
`uvm_object_utils_begin(CS_transaction)
`uvm_field_int(RESET,UVM_DEFAULT)
`uvm_field_int(addA,UVM_DEFAULT)
`uvm_field_int(addB,UVM_DEFAULT)
`uvm_field_int(addC,UVM_DEFAULT)
`uvm_field_int(OPERATION,UVM_DEFAULT)
`uvm_field_int(DQ,UVM_DEFAULT)
`uvm_field_int(DQ_temp,UVM_DEFAULT)
`uvm_field_int(overflow,UVM_DEFAULT)
`uvm_object_utils_end

endclass : CS_transaction
```

For the interface, I defined a temporary variable “DQ_temp” to facilitate the driving mechanism by driving “DQ” during write operations only. I was trying to reverse the DUT behavior (Driving at read operations only). The clocking block usage will be explained later.

For the sequence item or transaction class, I’m randomizing the inputs and “DQ_Temp” as it will be assigned to “DQ” during write operations. I’m also utilizing UVM field macros for all the data members.

Sequences:

```
class CS_random_sequence extends uvm_sequence#(CS_transaction);  
  
`uvm_object_utils(CS_random_sequence)  
  
function new(input string inst = "CS_random_sequence");  
    super.new(inst);  
endfunction : new  
  
CS_transaction transaction;  
  
task body();  
  
    transaction = CS_transaction::type_id::create("transaction");  
  
    //Reset  
    start_item(transaction);  
    transaction.RESET = 1'b0;  
    finish_item(transaction);  
  
    //Random sequence  
    repeat(10000) begin  
        start_item(transaction);  
  
        transaction.RESET = 1'b1;  
        assert(transaction.randomize())  
        else `uvm_error("Sequence","Randomization failed")  
  
        finish_item(transaction);  
    end  
endtask : body  
  
endclass : CS_random_sequence
```

```
/*  
Write into a certain location, then read from the same location.  
*/  
//Write  
start_item(transaction);  
transaction.RESET = 1'b1;  
transaction.OPERATION = 2'b01;  
transaction.addC = 'b1111;  
transaction.DQ_temp = 'b1;  
finish_item(transaction);  
  
//Read  
start_item(transaction);  
transaction.RESET = 1'b1;  
transaction.OPERATION = 2'b00;  
transaction.addA = 'b1111;  
finish_item(transaction);  
  
/*  
Add two values , then read the result.  
*/  
start_item(transaction);  
transaction.RESET = 1'b1;  
transaction.OPERATION = 2'b11;  
transaction.addA = 'b0110;  
transaction.addB = 'b0111;  
transaction.addC = 'b1111;  
finish_item(transaction);  
  
//Read  
start_item(transaction);  
transaction.RESET = 1'b1;  
transaction.OPERATION = 2'b00;  
transaction.addA = 'b1111;  
finish_item(transaction);  
  
/*  
Sub two values , then read the result.  
*/
```

For the random sequence, I randomized 10,000 transactions to cover the four operations and test different random scenarios. I also utilized some directed sequences to test specific scenarios.

Driver:

```
class CS_driver extends uvm_driver#(CS_transaction);
`uvm_component_utils(CS_driver)

//Constructor
function new(string name = "CS_driver", uvm_component parent = null);
    super.new(name,parent);
endfunction: new

//Build phase
virtual CS_if vif;
function void build_phase(uvm_phase phase);
    if(!uvm_config_db #(virtual CS_if)::get(this, "", "vif", vif))
        `uvm_fatal("NO VIF", "Interface Not Found");
endfunction : build_phase

//Run phase
virtual task run_phase(uvm_phase phase);
forever begin
    CS_transaction transaction;
    @(vif.cb);

    seq_item_port.get_next_item(transaction);
    drive(transaction);
    reportInfo(transaction);
    seq_item_port.item_done();

end
endtask : run_phase
```

```
task drive(CS_transaction transaction);

vif.RESET <= transaction.RESET;
vif.addA <= transaction.addA;
vif.addB <= transaction.addB;
vif.addC <= transaction.addC;
vif.OPERATION <= transaction.OPERATION;

/*
Driving "DQ_temp" which is assigned to "DQ" port.
*/
if (transaction.OPERATION == 2'b01) begin
    vif.DQ_temp <= transaction.DQ_temp;
end
endtask

task reportInfo(CS_transaction transaction);
$display("##### Driving the DUV #####");

if (transaction.RESET == 1'b0) begin
    $display("- Current simulation time:%0t\nXX Resetting XX");
end
else if (transaction.OPERATION == 2'b00) //Read
    $display("- Current simulation time:%0t\n- Operation: Read");
else if (transaction.OPERATION == 2'b01) //Write
    $display("- Current simulation time:%0t\n- Operation: Write");
else if (transaction.OPERATION == 2'b11) //Add
    $display("- Current simulation time:%0t\n- Operation: Add");
else //Subtraction
    $display("- Current simulation time:%0t\n- Operation: Subtraction");

$display("#####\n");
endtask
```

I'm using a clock period of 20 ns. For driving and monitoring, I have many options. One of them is to drive at the positive edge and capture the DUT output at the following edge. This could be achieved by utilizing the clocking blocks with zero input and output skews (example: drive at 0 ns, capture output at 20 ns).

```
//Clocking block
clocking cb @ (posedge CLK);
    default input #0 output #0 ;
```

Another option is to drive the DUT at a skew of #1 ns after the positive edge for example and monitoring the DUT outputs at #1step input skew (also utilizing the clocking block). This option consumes more time. However, it's recommended.

```
//Clocking block
clocking cb @ (posedge CLK);
    default input #1step output #1 ;
```

Here, it won't be a problem using any option as I'm verifying the functionality, so I will go with the first option for this design (easier to debug).

Monitor:

```
class CS_monitor extends uvm_monitor;
`uvm_component_utils(CS_monitor)

//Interface Handler
virtual CS_if vif;
//Analysis Port Handler
uvm_analysis_port #(CS_transaction) analysis_port;

//Constructor
function new(input string inst = "MON", uvm_component parent);
super.new(inst,parent);
analysis_port = new("analysis_port", this);
endfunction : new

//Build phase
virtual function void build_phase (uvm_phase phase);
super.build_phase(phase);
if(!uvm_config_db#(virtual CS_if)::get(this, "", "vif", vif))
`uvm_fatal("NOVIF","Interface Not Found");
endfunction : build_phase

//Run phase
virtual task run_phase (uvm_phase phase);
forever begin
CS_transaction transaction;

@vif.cb;
transaction = CS_transaction::type_id::create("transaction");
capture(transaction);
analysis_port.write(transaction);
end
endtask : run_phase

task capture(output CS_transaction transaction);
CS_transaction temp = CS_transaction::type_id::create("transaction");
//monitoring
temp.RESET = vif.RESET;
temp.addA = vif.addA;
temp.addB = vif.addB;
temp.addC = vif.addC;

temp.OPERATION = vif.OPERATION;

temp.DQ = vif.cb.DQ;
temp.overflow = vif.cb.overflow;

transaction = temp;
endtask
endclass: CS_monitor
```

Not too much here, capturing the DUT outputs at the clocking event and transmitting the captured values through the analysis port.

Coverage:

Here I'm covering the different operations, the overflow flag values, and the different addresses. I used automatic bins to cover the different addresses. I also defined cross coverage to cover the case when two large numbers are added (to check the overflow).

```
class CS_coverage extends uvm_subscriber #(CS_transaction);
`uvm_component_utils(CS_coverage)

bit          RESET;
bit [1:0]    OPERATION;
bit          overflow;
bit [$clog2(`DEPTH)-1:0] addA;
bit [$clog2(`DEPTH)-1:0] addB;
bit [$clog2(`DEPTH)-1:0] addC;

//Cover group
covergroup CS_cover_group;
//Covering different operations
Operations: coverpoint OPERATION iff(RESET) {
    bins Read = {0};
    bins Write = {1};
    bins Sub = {2};
    bins Add = {3};
}

//Covering different addresses
addA: coverpoint addA iff(RESET){
    bins Diff_values[] = {[0:$]};
}
addB: coverpoint addB iff(RESET){
    bins Diff_values[] = {[0:$]};
}
addC: coverpoint addC iff(RESET){
    bins Diff_values[] = {[0:$]};
}

//Covering overflow values (0&1)
Overflow: coverpoint overflow iff(RESET){
    bins high = {1};
    bins low = {0};
}
```

```
//Covering addition && overflow (0&1)
Add_OF: cross Operations,Overflow iff(RESET) {
    bins of1 = Add_OF with(Operations == 3 && Overflow == 1);
    bins of0 = Add_OF with(Operations == 3 && Overflow == 0);

    ignore_bins ig1 = Add_OF with (Operations !=3 && Overflow);
    ignore_bins ig2 = Add_OF with (Operations !=3 && ~Overflow);
}
endgroup

//Constructor
function new (input string name, uvm_component parent);
    super.new(name, parent);
    CS_cover_group = new();
endfunction : new

//Write method
function void write(CS_transaction t);
    RESET      = t.RESET;
    OPERATION   = t.OPERATION;
    overflow    = t.overflow;
    addA        = t.addA;
    addB        = t.addB;
    addC        = t.addC;
    CS_cover_group.sample();
endfunction : write
endclass
```


Scoreboard:

In the scoreboard, my idea is to mimic the DUT behavior. First, I will define a 2D array matching the DUT memory dimensions and initiate it at the beginning of the simulation with the same values loaded to the DUT. The captured transaction will be passed to the scoreboard via the analysis port, these captured transactions will be applied to the scoreboard memory.

Reset operation: scoreboard memory will return to the initial state.

Read operation: compare the stored value in the scoreboard memory with the captured value from the DUT and if they match, the test passes.

Write operation: write the given value in the given location (in the scoreboard memory).

Addition operation: add the two stored values and store them, and check if the overflow flag matches the DUT output.

Subtraction operation: subtract the two stored values and store them in the given location.

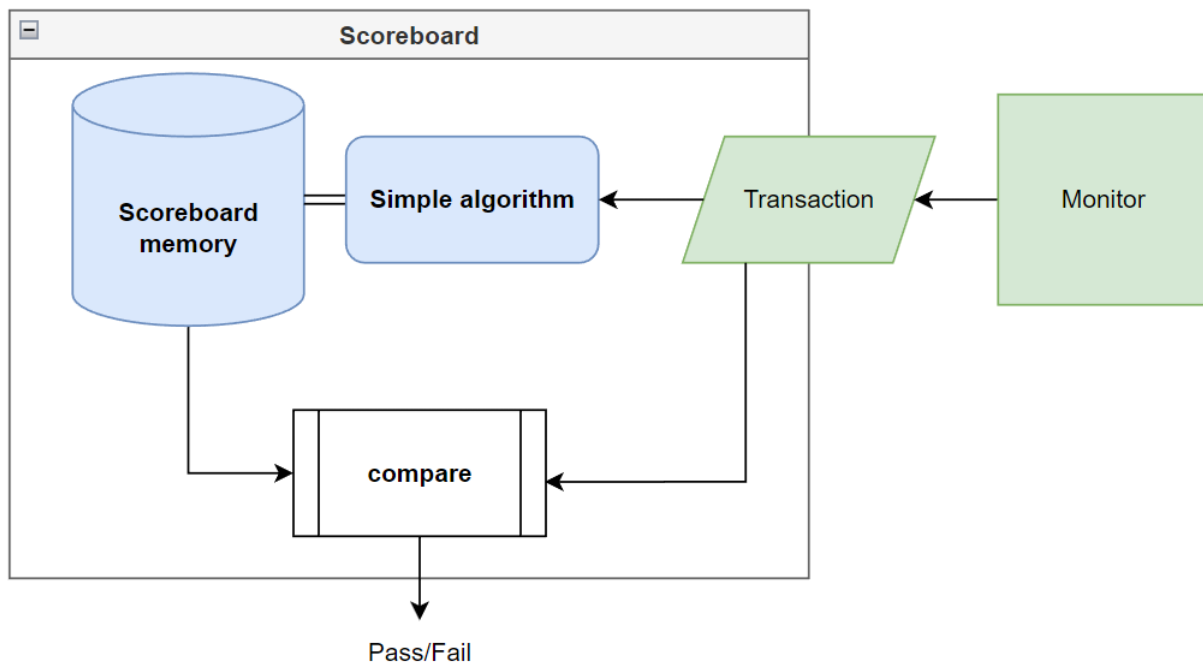


Figure 8 Scoreboard algorithm

Executing many transactions will ensure that the updated values in both the DUT and the scoreboard memory will be compared. After exercising many random transactions, a read sequence may be executed to do the comparison.

```

class CS_scoreboard extends uvm_scoreboard;
`uvm_component_utils(CS_scoreboard)
uvm_analysis_imp #(CS_transaction,CS_scoreboard) analysis_export;

function new (string name = "CS_scoreboard" , uvm_component parent = null);
super.new(name,parent);
analysis_export = new("READ",this);

$readmemh("MEM_PRE_HEX.txt", scbdMem); //Initiating the scoreboard memory.
endfunction

/*
Defining a 2D array to mimic the DUT memory.
*/
reg [`WIDTH-1:0] scbdMem [0:`DEPTH-1];
reg [`WIDTH:0] temp;

//Write function
function void write(CS_transaction transaction);

$display("##### Scoreboard report #####");
$display("- Current simulation time:%0t", $time);

/*
Reset operation: re initiate the scoreboard memory.
*/
if (transaction.RESET == 1'b0) begin
$display("XX Resetting XX");
$readmemh("MEM_PRE_HEX.txt", scbdMem);
end

/*
Write operation: write the given value at the same location given to the DUT.
*/
else if (transaction.OPERATION == 2'b01) begin
scbdMem[transaction.addC] = transaction.DQ;
end

```

```

Read operation: compare the captured value from the DUT with the stored value
in the scoreboard memory; if they match, then the test passes.
*/
else if (transaction.OPERATION == 2'b00) begin
$display("DUT output: DQ = %0h", transaction.DQ);
$display("Expected output: DQ = %0h", scbdMem[transaction.addA]);

if (scbdMem[transaction.addA] == transaction.DQ) begin
$display("Test result: Passed");
end
else begin
`uvm_error("SCOREBOARD","Test result: Failed (The captured value does not match the expected value)")
end
end

/*
Addition operation: mimic the DUT behavior and check whether the overflow flag matches.
*/
else if (transaction.OPERATION == 2'b11) begin
$display("DUT output: Overflow = %0h", transaction.overflow);
temp = scbdMem[transaction.addA] + scbdMem[transaction.addB];
scbdMem[transaction.addC] = scbdMem[transaction.addA] + scbdMem[transaction.addB];

if (temp[`WIDTH] == transaction.overflow) begin
$display("Test result: Passed");
end
else begin
`uvm_error("SCOREBOARD","Test result: Failed (The captured value does not match the expected value)")
end
end

/*
Subtraction operation, mimic the DUT behavior.
*/
else begin
scbdMem[transaction.addC] = scbdMem[transaction.addA] - scbdMem[transaction.addB];
end
$display("#####\n");
endfunction : write
endclass : CS_scoreboard

```

Agent & Environment: (not too much, containers)

```
class CS_agent extends uvm_agent;
`uvm_component_utils(CS_agent)

//Constructor
function new(input string inst = "AGT", uvm_component parent);
super.new(inst,parent);
endfunction : new

//Agent components
CS_driver    drv;
CS_sequencer seqr;
CS_monitor   mon;

//Build phase
virtual function void build_phase (uvm_phase phase);
super.build_phase(phase);

//check if Agent is ACTIVE
if(get_is_active() == UVM_ACTIVE) begin
    drv = CS_driver::type_id::create("drv", this);
    seqr = CS_sequencer::type_id::create("seqr", this);
end
mon = CS_monitor::type_id::create("mon", this);
endfunction : build_phase

//Connect phase
function void connect_phase(uvm_phase phase);
//check if Agent is ACTIVE
if(get_is_active() == UVM_ACTIVE) begin
    drv.seq_item_port.connect(seqr.seq_item_export);
end
endfunction : connect_phase

endclass: CS_agent
```

```
class CS_environment extends uvm_env;
`uvm_component_utils(CS_environment)

//Constructor
function new(string name = "CS_environment", uvm_component parent = null);
    super.new(name,parent);
endfunction: new

CS_agent    ag;
CS_scoreboard scbd;
CS_coverage cov;

//Build phase
function void build_phase(uvm_phase phase);
    scbd = CS_scoreboard::type_id::create("scbd",this);
    ag   = CS_agent::type_id::create("ag",this);
    cov  = CS_coverage::type_id::create("cov",this);
endfunction : build_phase

//Connect phase
function void connect_phase(uvm_phase phase);
    ag.mon.analysis_port.connect(scbd.analysis_export);
    ag.mon.analysis_port.connect(cov.analysis_export);
endfunction : connect_phase

endclass : CS_environment
```

Test & TB Top module:

```
class CS_base_test extends uvm_test;
`uvm_component_utils(CS_base_test)

//Constructor
function new(string inst = "TEST", uvm_component c);
    super.new(inst, c);
endfunction : new

CS_environment env;
CS_random_sequence seq;

//Build phase
virtual function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    env = CS_environment::type_id::create("ENV", this);
endfunction : build_phase

//Run phase
virtual task run_phase (uvm_phase phase);
    random_seq = CS_random_sequence::type_id::create("GEN", this);
    /* And other sequences*/

    phase.raise_objection(this);
    random_seq.start(env.ag.seqr);
    /* And other sequences*/
    phase.drop_objection(this);
endtask : run_phase
endclass : CS_base_test
```

```
module Top;
//Clock generation
logic CLK;

initial begin
    CLK = 1'b0;
    #10;
    forever begin
        CLK = ~ CLK;
        #10;
    end
end

//Interface instance
CS_if vif (.CLK(CLK));

//DUT instantiation
CompStorage DUT(
    .CLK(CLK),
    .RESET(vif.RESET),
    .addA(vif.addA),
    .addB(vif.addB),
    .addC(vif.addC),
    .OPERATION(vif.OPERATION),
    .DQ(vif.DQ),
    .overflow(vif.overflow)
);

initial begin
    $dumpfile("CS.vcd") ;
    $dumpvars;
    uvm_config_db#(virtual CS_if)::set(null, "*", "vif", vif);
    run_test("CS_base_test");
end
endmodule
```

- Results

- “Before any operations, check that the initial values stored inside the memory matches the preloading file.”

```
//Reset
start_item(transaction);
transaction.RESET = 1'b0;
finish_item(transaction);

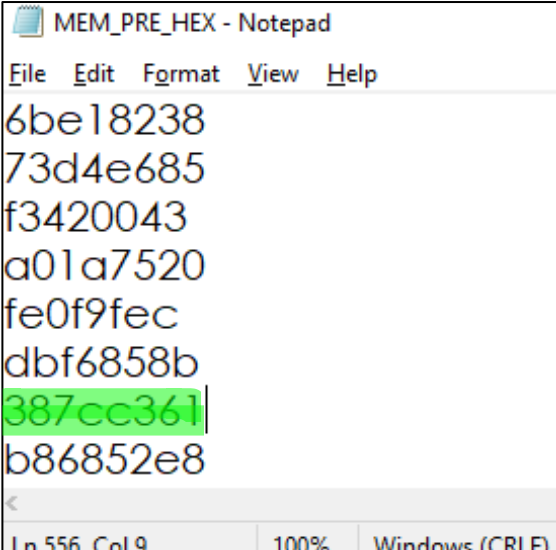
//Read sequence
repeat(100) begin
start_item(transaction);

transaction.RESET = 1'b1;
assert(transaction.randomize() with {OPERATION == 2'b00;})
| else `uvm_error("Sequence","Randomization failed")

finish_item(transaction);
end
```

After resetting the design, I added 100 read operations, to ensure that the stored values inside the memory are same as the preloading file. The test passed.

```
##### Driving the DUV #####
# - Current simulation time 10
# XX Resetting XX
#####
#
##### Scoreboard report #####
# - Current simulation time:30
# XX Resetting XX
#####
#
##### Driving the DUV #####
# - Current simulation time:30
# - Operation: Read
# - Address A:22b
#####
#
##### Scoreboard report #####
# - Current simulation time:50
# DUT output: DQ = 387cc361
# Expected output: DQ = 387cc361
# Test result: Passed
#####
#
##### Driving the DUV #####
# - Current simulation time:50
# - Operation: Read
# - Address A:212
#####
#
##### Scoreboard report #####
# - Current simulation time:70
# DUT output: DQ = 6cbac5c7
# Expected output: DQ = 6cbac5c7
# Test result: Passed
#####
```



MEM_PRE_HEX - Notepad

File Edit Format View Help

6be18238
73d4e685
f3420043
a01a7520
fe0f9fec
dbf6858b
387cc361
b86852e8

Ln 556, Col 9 100% Windows (CRLF)

- “Resetting the DUT after some random operations to make sure it will return to its initial values.”

```
//Reset
start_item(transaction);
transaction.RESET = 1'b0;
finish_item(transaction);

//Random sequence
repeat(100) begin
start_item(transaction);

transaction.RESET = 1'b1;
assert(transaction.randomize())
| else `uvm_error("Sequence","Randomization failed")

finish_item(transaction);
end

//Reset
start_item(transaction);
transaction.RESET = 1'b0;
finish_item(transaction);

//Read sequence
repeat(100) begin
start_item(transaction);

transaction.RESET = 1'b1;
assert(transaction.randomize() with {OPERATION == 2'b00;})
| else `uvm_error("Sequence","Randomization failed")

finish_item(transaction);
end
```

After resetting the design, I added 100 random operations, asserted the reset signal then added 100 read operations to ensure that the stored values inside the memory are the same as the preloading file. **The test passed.**

```
##### Driving the DUV #####
# - Current simulation time:2010
# - Operation: Write
# - Address C:2a1
# - Value:a72e4046
#####
##### Scoreboard report #####
# - Current simulation time:2030
#####
##### Driving the DUV #####
# - Current simulation time:2030
# XX Resetting XX
#####
##### Scoreboard report #####
# - Current simulation time:2050
# XX Resetting XX
#####
##### Driving the DUV #####
# - Current simulation time:2050
# - Operation: Read
# - Address A:149
#####
##### Scoreboard report #####
# - Current simulation time:2070
# DUT output: DQ = 92476c94
# Expected output: DQ = 92476c94
# Test result: Passed
#####
```

MEM_PRE_HEX - Notepad

File Edit Format View Help

```
4bd3522a
ec965e5b
92476c94
17c62aba
b98cc76b
66599d24
11f7633a
6ea05f74
```

Ln 330, Col 9 100% Windows (CRLF)

- “Write a random value into a certain memory location and read that value in the following cycle.”

```
//Reset
start_item(transaction);
transaction.RESET = 1'b0;
finish_item(transaction);

/*
Write into a certain location, then read from the same location.
*/
//Write
start_item(transaction);
transaction.RESET      = 1'b1;
transaction.OPERATION = 2'b01;
transaction.addC       = 'b1111;
transaction.DQ_temp    = 'b1;
finish_item(transaction);

//Read
start_item(transaction);
transaction.RESET      = 1'b1;
transaction.OPERATION = 2'b00;
transaction.addA       = 'b1111;
finish_item(transaction);
```

Although I used directed testing here as the operation is simple, I think if the number of randomized transactions is very high or using the randc key word the same result will be achieved. Here I stored the value “1” in location “f” and retrieved the same value in the following cycle. **Test passed.**

```
##### Driving the DUV #####
# - Current simulation time:30
# - Operation: Write
# - Address C:00f
# - Value:00000001
#####
##### Scoreboard report #####
# - Current simulation time:50
#####
##### Driving the DUV #####
# - Current simulation time:50
# - Operation: Read
# - Address A:00f
#####
##### Scoreboard report #####
# - Current simulation time:70
# DUT output: DQ = 1
# Expected output: DQ = 1
# Test result: Passed
#####
```

- "Writing two values into certain locations then perform adding operation on them, then retrieving the stored result."
- "Writing two values into certain locations then perform subtracting operation on them, then retrieving the stored result." *I used directed testing for simplicity*

```
##### Driving the DUV #####
# - Current simulation time:30
# - Operation: Addition
# - Address A:006
# - Address B:007
# - Address C:00f
#####
##### Scoreboard report #####
# - Current simulation time:50
# DUT output: Overflow = 0
# Test result: Passed
#####
##### Driving the DUV #####
# - Current simulation time:50
# - Operation: Read
# - Address A:00f
#####
##### Scoreboard report #####
# - Current simulation time:70
# DUT output: DQ = ed62e7d5
# Expected output: DQ = ed62e7d5
# Test result: Passed
#####
```

MEM_PRE_HEX - Notepad

File Edit Format View Help

231b808d
e57d8df6
977213e3
16eea6fc
d67440d9
d434406b
2567f78b
73b79ea4

<

Ln 8, Col 9 100% Windows (CRLF)

```
##### Driving the DUV #####
# - Current simulation time:70
# - Operation: Subtraction
# - Address A:000
# - Address B:001
# - Address C:00f
#####
##### Scoreboard report #####
# - Current simulation time:90
#####
##### Driving the DUV #####
# - Current simulation time:90
# - Operation: Read
# - Address A:00f
#####
##### Scoreboard report #####
# - Current simulation time:110
# DUT output: DQ = fd334f7
# Expected output: DQ = fd334f7
# Test result: Passed
#####
```

MEM_PRE_HEX - Notepad

File Edit Format View Help

9f277b3e
8f544647
33e1c448
231b808d
e57d8df6
977213e3
16eea6fc
d67440d9

<

Ln 1, Col 9 100% Windows (CRLF)

Both tests passed successfully.

- “Adding two large values so that the addition operation will assert the overflow flag.”

Test passed. (Covered using random sequence)

- “Adding two values and storing the result in the same location as one of the operands.”

Many approaches may achieve the same result as directed testing (constraints or maybe randomizing large number of transactions).

Test passed.

```
##### Driving the DUV #####
# - Current simulation time:30
# - Operation: Addition
# - Address A:000
# - Address B:001
# - Address C:000
#####
#
##### Scoreboard report #####
# - Current simulation time:50
# DUT output: Overflow = 1
# Test result: Passed
#####
#
##### Driving the DUV #####
# - Current simulation time:50
# - Operation: Read
# - Address A:000
#####
#
##### Scoreboard report #####
# - Current simulation time:70
# DUT output: DQ = 2e7bcl85
# Expected output: DQ = 2e7bcl85
# Test result: Passed
#####
```

- Coverage result: (10000 random transactions)

#CS_cover_group#	UVM_sv_unit(fast)	SVCovergroup	- +acc=...	100.00%	100.00%
Operations	UVM_sv_unit(fast)	SVCoverpoint	- +acc=...	100.00%	100.00%
addA	UVM_sv_unit(fast)	SVCoverpoint	- +acc=...	100.00%	100.00%
addB	UVM_sv_unit(fast)	SVCoverpoint	- +acc=...	100.00%	100.00%
addC	UVM_sv_unit(fast)	SVCoverpoint	- +acc=...	100.00%	100.00%
Overflow	UVM_sv_unit(fast)	SVCoverpoint	- +acc=...	100.00%	100.00%
Add_OF	UVM_sv_unit(fast)	SVCross	- +acc=...	100.00%	100.00%

- ✓ Four operations covered.
- ✓ Different addresses covered.
- ✓ Overflow signal different values.
- ✓ Cross coverage between addition and overflow signal.

○ Comparing final memory contains with preloaded file

Using Python, I developed simple script to read both (preloaded file and final memory contains) and compare them.

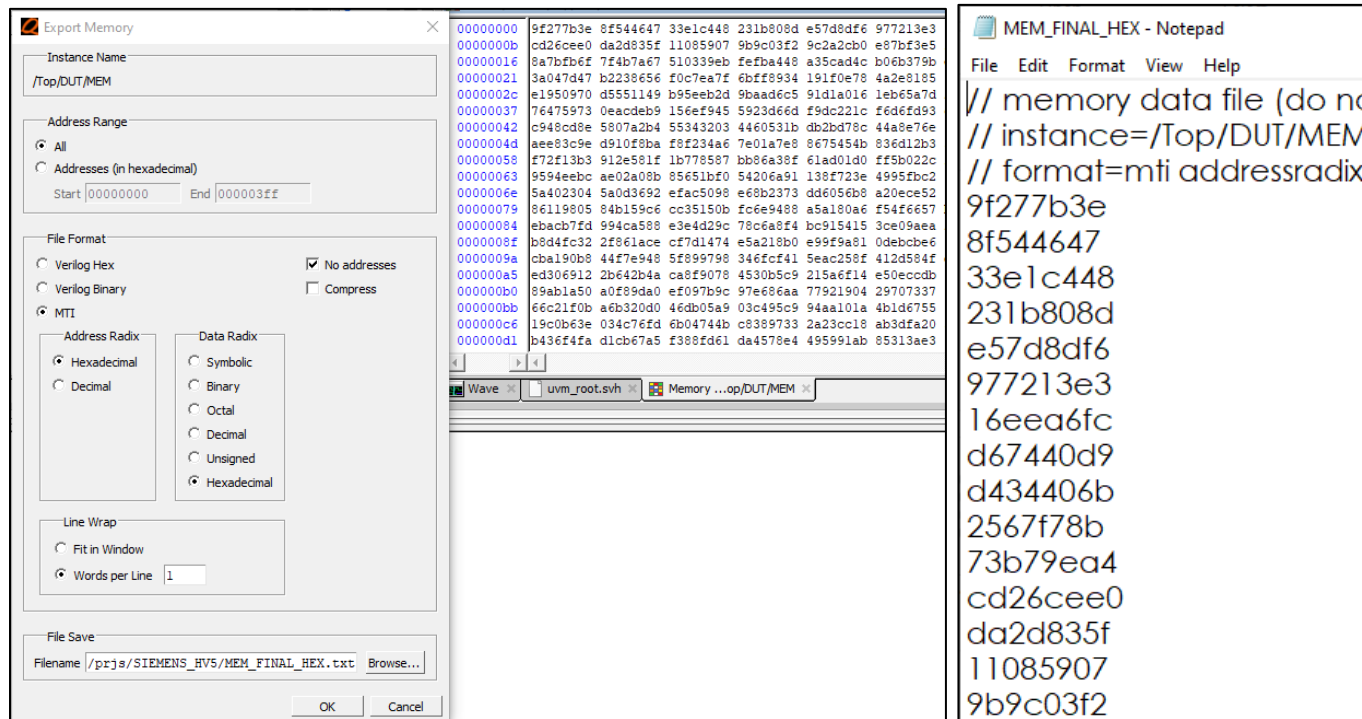


Figure 9 Exporting final memory contains.

```
#Defining two lists
preloaded_values = []
final_values = []

#open text files
with open("MEM_PRE_HEX.txt", "r") as file:
    preloaded_values = file.readlines()

with open("MEM_FINAL_HEX.txt", "r") as file:
    for line in file:
        if not line or line.startswith("//"):
            continue
        else:
            final_values.append(line)

#compare contents
numMatchValues = 0
numDiffValues = 0

for index in range(len(preloaded_values)):
    if preloaded_values[index] == final_values[index]:
        numMatchValues +=1
    else:
        numDiffValues +=1

#print results
print(f"Number of modified values: {numDiffValues}\nNumber of unchanged values: {numMatchValues}")
```

Figure 10 Comparison script

When the number of transactions is very large, it's expected that the number of unchanged values will reach zero.

```
Number of modified values: 1024  
Number of unchanged values: 0  
PS E:\EDU\prjs\SIEMENS_HV5>
```

Figure 11 Comparison result in case of large number of transactions

```
Number of modified values: 144  
Number of unchanged values: 880  
PS E:\EDU\prjs\SIEMENS_HV5>
```

Figure 12 Comparison result in case of 200 transactions

Thanks for your time.

Abdulrahman Nour Eldeen