System Verilog Testbench Tutorial

Using Synopsys EDA Tools

Developed By

Abhishek Shetty

Guided By

Dr. Hamid Mahmoodi

Nano-Electronics & Computing Research Center
School of Engineering
San Francisco State University
San Francisco, CA

Fall 2011

Table of Contents

Tables and Figures:

1. Introduction

The Verification Process

The process of verification parallels the design creation process. A designer reads the hardware specification for a block, interprets the human language description, and creates the corresponding logic in a machine-readable form, usually RTL code written using Verilog or VHDL language. To do this, the user needs to understand the input format, the transformation function, and the format of the output. There is always ambiguity in this interpretation, perhaps because of ambiguities in the original document, missing details or conflicting descriptions. The SystemVerilog language provides three important benefits over Verilog.

1. Explicit design intent – SystemVerilog introduces several constructs that allow you to explicitly state what type of logic should be generated.
2. Conciseness of expressions – SystemVerilog includes commands that allow you to specify design behavior more concisely than previously possible.
3. A high level of abstraction for design – The SystemVerilog interface construct facilitates inter module communication.

These benefits of SystemVerilog enable you to rapidly develop your RTL code, easily maintain your code, and minimize the occurrence of situations where the RTL code simulates differently than the synthesized netlist. SystemVerilog allows you to design at a high level of abstraction. This results in improved code readability and portability. Advanced features such as interfaces, concise port naming, explicit hardware constructs, and special data types ease verification challenges.

Basic Testbench Functionality

The purpose of a Testbench is to determine the correctness of the design under test (DUT). The following steps accomplish this.

- Generate stimulus
- Apply stimulus to the DUT
- Capture the response
- Check for correctness
- Measure progress against the overall verification goals

2. Building Blocks of Testbench

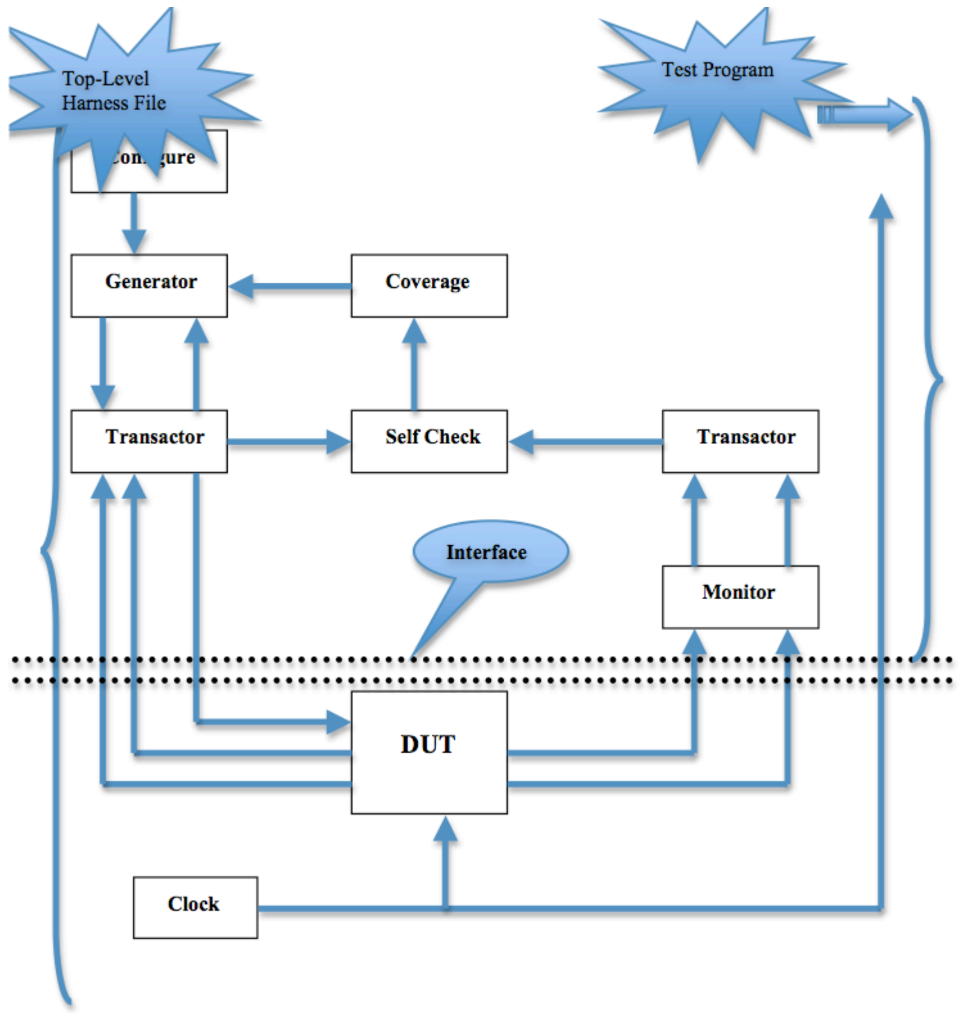


Fig 1

System Verilog Verification Features

These features assist in the creation of extensible, flexible test benches.

New Data Types:

```
int da[];       // dynamic array
int da[string];       // Associative array, indexed by string
int da[$];            // Queue

initial begin
    da = new[16]; // Create 16 elements
end
```

The string data type represents a variable-length text string, which is a unique feature of System Verilog.

System Verilog offers dynamic arrays, associative arrays and queues. The array can be resized if needed.

The queue provides much of the C++ STL deque type: elements can be added and removed from either end efficiently. Complex data structures can be created for score boarding a large design.

Classes

System Verilog provides an object-oriented programming model.

System Verilog classes support a single-inheritance model. There is no facility that permits conformance of a class to multiple functional interfaces, such as the interface feature of Java. System Verilog classes can be type-parameterized, providing the basic function of C++ templates. However, function templates and template specialization are not supported.

The polymorphism features are similar to those of C++: the programmer may specify write a virtual function to have a derived class gain control of the function. Encapsulation and data hiding is accomplished using the local and protected keywords, which must be applied to any item that is to be hidden. By default, all class properties are public. System Verilog class instances are created with the new keyword. A constructor denoted by function new can be defined. System Verilog supports garbage collection, so there is no facility to explicitly destroy class instances.

Example:

```
virtual class Memory;
    virtual function bit [31:0] read(bit [31:0] addr); endfunction
    virtual function void write(bit [31:0] addr, bit [31:0] data);
endfunction
endclass

class SRAM #(parameter AWIDTH=10) extends Memory;
   bit [31:0] mem [1<<AWIDTH];
   virtual function bit [31:0] read (bit [31:0] addr);
      return mem [addr];
   endfunction
   virtual function void write(bit [31:0] addr, bit [31:0] data);
      mem[addr] = data;
   endfunction
endclass
```

There are other important features like Randomization, Assertions, and Coverage, which is discussed in future sections.

2.1 Program Block – Encapsulates Test Code

The program block can read and write all signals in modules, and can call routines in modules, but a module have no visibility into a program. This is because your Testbench needs to see and control the design, but the design should not depend on anything in the Testbench.
A program can call a routine in a module to perform various actions. The routine can set values on internal signals, also known as "back- door load." Next, because the current SystemVerilog standard does not define how to force signals from a program block, you need to write a task in the design to do the force, and then call it from the program.

The Program block provides

- Entry point to test execution
- Scope for program-wide data and routines
- Race-free interaction between Testbench and design

Test code syntax for program block

```
Program automatic test (router_io. TB rtr_io);
      //Testbench code in initial block
    initial begin
        run ();
     end

     task run ();
    .......
    ....
    endtask: run
endprogram: test
```

Fig 2: Test Code: Program Block

By default, System Verilog programs are static, just like verilog module instances, tasks and functions. To allow dynamic allocation of memory like Vera and C++ always make programs "automatic".

Why to declare System Verilog program blocks as *"automatic"*?

Any data declared outside a module, interface, task or future are global in scope (can be used anywhere after its declaration) and have a static lifetime (means which exist for the whole elaboration and simulation time).

System Verilog allows specific data within a static task or function to be explicitly declared as automatic. Data declared as automatic have the lifetime of the call or block and are initialized on each entry to the call or block. By default programs in System Verilog have a static lifetime, meaning all variables defined have a static lifetime meaning all variables are defined static (allocated at compile time). This is done to maintain backward compatibility with Verilog modules, which have a static lifetime. However for all practical purposes programs need dynamic variables. This requires that we make the program block as *automatic.*

Why *'always'* Blocks Not allowed in a Program Block?

In System Verilog, you can put an *initial* blocks in a program, but not *always* blocks. This is bit opposite to the verilog and we have the reasons below:

- System Verilog programs are closer to program in C, with one entry point, than Verilog's many small blocks of concurrently executing hardware.
- In a design, always block might trigger on every positive edge of a clock from the start of simulation. In System Verilog, a Testbench has the steps of initialization, stimulate and respond to the design and then wrap up the simulation. An always block that runs continuously would not work in System Verilog.

2.2 The Interface

It is the mechanism to connect Testbench to the DUT just named as bundle of wires (e.g. connecting two hardware unit blocks with the help of physical wires). With the help of interface block: we can add new connections easily, no missed connections and port lists are compact. It also carries the directional information in the form of Modport (will be explained in counter example) and clocking blocks.

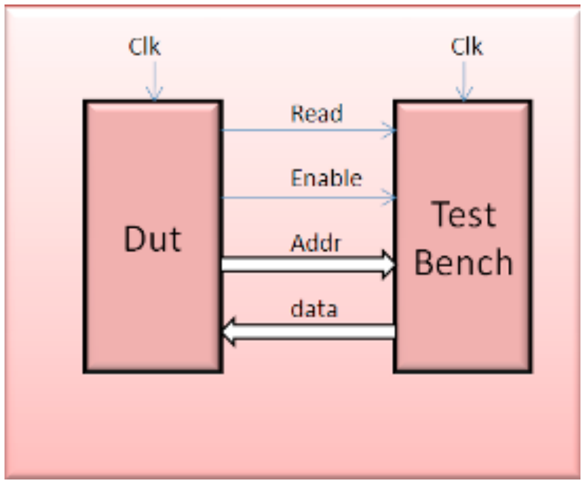
*** An interface encapsulates the communication between DUT and Testbench including**

- Connectivity (signals) – named bundle of wires
    - One or more bundles to connect
    - Can be reduced for different tests and devices
- Directional information (modports)
- Timing (Clocking blocks)
- Functionality (routines, assertions, initial/always blocks)

*** Solves many problems with traditional connections**

- Port lists for the connections are compact
- No missed connections
- Easy to add new connections
- New signals in the interface are automatically passed to test program or module, preventing connection problems.
- Changes are easily made in interface making it easy to integrate in higher levels
- Most port declaration work is duplicated in many modules.
- And most important is easy to change if design changes.

Interface – An Example



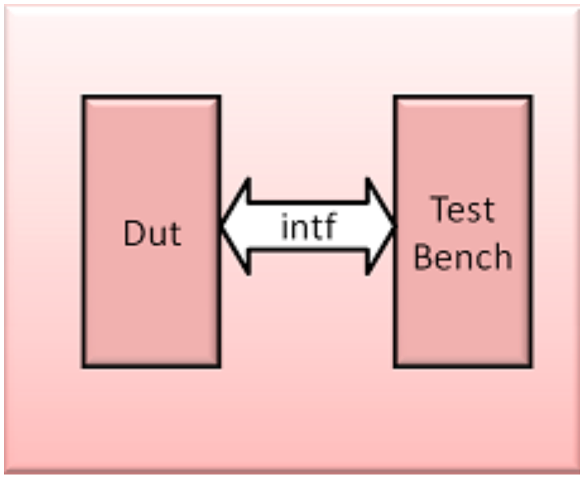


Fig 3

In the above example, all the signals clk, read, enable, data are done manually. During developing this connection, if an extra signal needs to be added then everything needs to be changed. In the next figues, System Verilog added a powerful feature called Interfaces. Interface encapsulates the interconnection and communication between blocks.

The interface instance is to be used in program block, the data type for the signals should be logic. The reason is that signals within a program block are almost always driven within a procedural block(initial). All signals driven within a procedural block must be of type reg, the synomn of which is logic. When a signal is declared to be logic, it can also be driven by a continuous assignment statement. This added flexibility of logic is generally desirable.

There is an exception to the above recommendation. If the signal is bi-directional signal(inout), or has multiple drivers, then the data must be wire (or any other form of wire-types).

TIP: Use wire type in case of multiple drivers. Use logic type in case of a single driver.

How do we achieve Synchronous Timing for different modules?

2.3 Clocking Blocks

- A clocking block assembles signals that are synchronous to a particular clock, and makes their timing explicit. A clocking block (CB) specifies clock signals and the timing and synchronization requirements of various blocks. A CB is helpful in separating clocking activities of a design from its data assignments activities and can be used in test benching.
- A CB assembles all the signals that are sampled or synchronized by a common clock and defines the timing behaviors with respect to the clock. It is defined by a *clocking – endclocking* keyword pair
- A CB only describes how the inputs and outputs are sampled and synchronized. It does not assign a value to a variable.
- Depending on the environment, a Testbench can contain one or more clocking blocks, each containing its own clock plus an arbitrary number of signals.

Clocking Block Example:

```
clocking cb @ (posedge clk)
   default input #1ns output #1ns;
   output reset_n;
   output din;
   output frame_n;
   output valid_n;
   input dout;
```

```
    input busy_n;
    input valido_n;
    input frameo_n;
endclocking: cb
```

Fig 4

- In the above example for CB, the ***default*** keyword defines the default skew for inputs and outputs. If not defined the default skew is ***default input #1step output #0***
- A ***1step*** is a special unit of time. It denotes a negative time delay from the edge of a clock when all inputs are steady and ready to be sampled.
- Each clocking block is associated with one clock with it.

An interface cannot contain module instances, but only instances of other interfaces. The advantages to using an interface are as follows:

- An interface is ideal for design reuse. When two blocks communicate with a specified protocol using more than two signals, consider using an interface.
- The interface takes the jumble of signals that you declare over and over in every module or program and puts it in a central location, reducing the possibility of misconnecting signals.
- To add a new signal, you just have to declare it once in the interface, not in higher-level modules, once again reducing errors.
- Modports allow a module to easily tap a subset of signals form an interface. We can also specify signal direction for additional checking.

Signal Direction Using Modport

```
interface route_io (input bit clock);
   logic reset_n;
    ........


clocking cb @ (posedge clock);
   default input #1ns output #1ns
   output reset_n;
   output valid_n;
    .......


endclocking
  modport DUT (input reset_n, input, din, output, dout, ...... );
  modport TB (clocking cb, output reset_n);
endinterface: router_io
```

```
module router (
router_io.DUT.db, input bit
clock);
..........

endmodule: router
```

```
program automatic test (router_io.TB rtr_io);
Initial
  rtr_io.reset_n <= 1'b0;
  rtr_io.valid_n <= '1;

  ........

endprogram: test
```

- It enforces signal access and direction with modport.

Fig 5

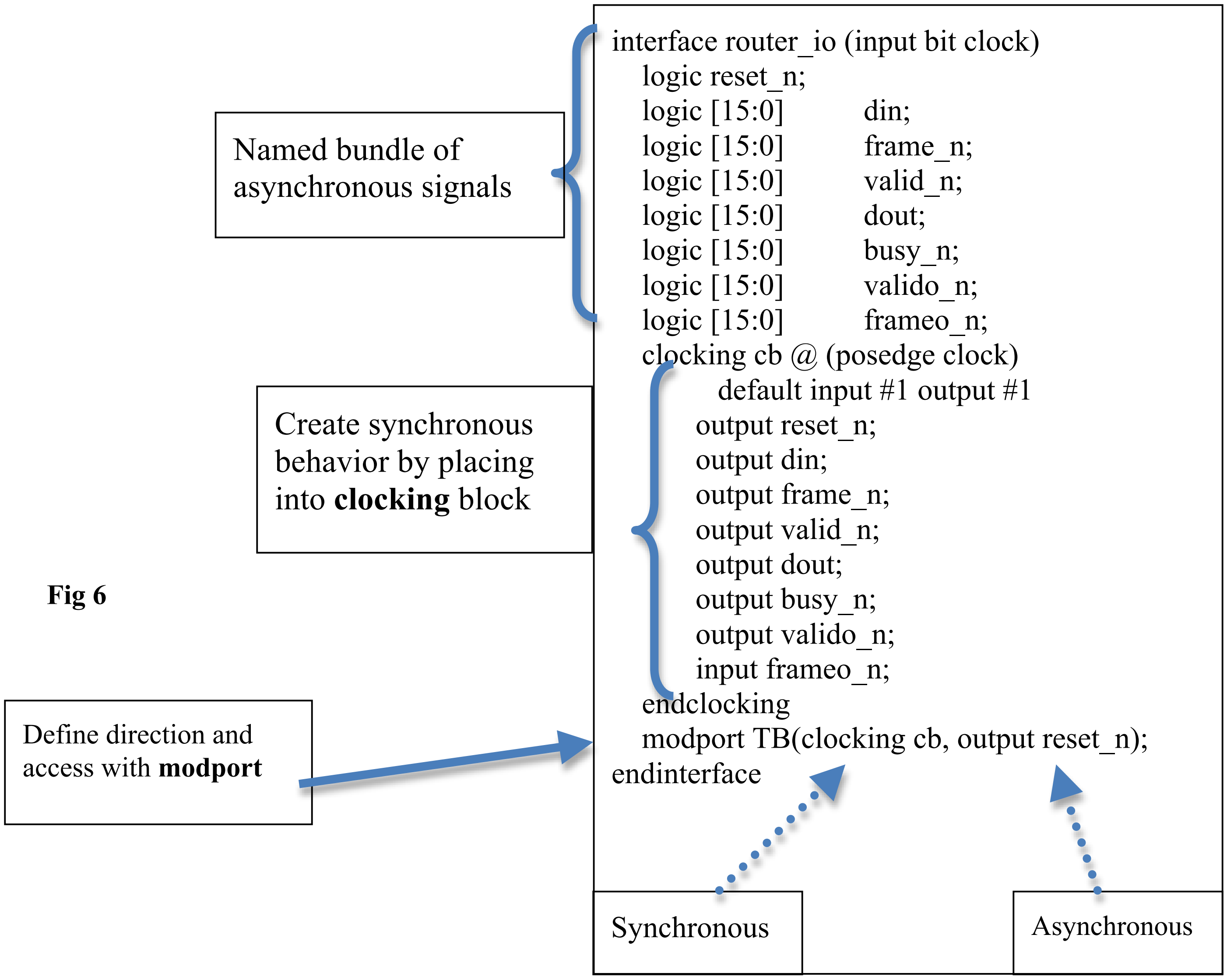
Referring to the above example:

- A new construct related to the interface is also added: ***modport***. This provides direction information for module interface ports and controls the use of tasks and functions within certain modules. The directions of ports are those seen from the perspective module or program.
- You can define a different view of the interface signals that each module or program sees on its interface ports
- Definition is made within the interface, and it may have any number of modports.

- Modports do not contain vector sizes or data types (common error) – only whether the connecting module sees a signal as ***input***, ***output***, ***inout*** or ***ref*** port.

2.4 Module Top: This file carries the top-level image of your whole design showing all the modules connected to it and the ports being used for the design. The interface and test programs are instantiated here in the harness files. Looking at into a top level harness files gives an detailed picture of any design, as to what are the functional parameters, interfaces etc.

A Complete Interface



```
interface router_io (input bit clock)
   logic reset_n;
   logic [15:0]        din;
   logic [15:0]        frame_n;
   logic [15:0]        valid_n;
   logic [15:0]        dout;
   logic [15:0]        busy_n;
   logic [15:0]        valido_n;
   logic [15:0]        frameo_n;
   clocking cb @ (posedge clock)
          default input #1 output #1
        output reset_n;
        output din;
        output frame_n;
        output valid_n;
        output dout;
        output busy_n;
        output valido_n;
        input frameo_n;
   endclocking
   modport TB(clocking cb, output reset_n);
endinterface
```

Named bundle of asynchronous signals
Create synchronous behavior by placing into **clocking** block
Define direction and access with **modport**
Synchronous
Asynchronous

Fig 6

2.5 Descriptions of some of the intermediate blocks

Environment - Environment contains the instances of the entire verification component and Component connectivity is also done. Steps required for execution of each component are done in this.

Coverage - Checks completeness of the Testbench. This can be improved by the usage of Assertions, which helps to check the coverage of the Testbench and generate suitable reports of the test coverage. The concept of coverage can get more complex, when we deal with the concept of functional coverage, cover groups and cover points. With the coverage points, we can generate coverage report of your design and know the strength of your verification.

Transactors – Transactor does the high level operations like burst-operations into individual commands, sub-layer protocol in layered protocol like PciExpress Transaction layer over PciExpress Data Link Layer, TCP/IP over Ethernet etc. It also handles the DUT configuration operations. This layer also provides necessary information to coverage model about the stimulus generated. Stimulus generated in generator is high level like Packet is with good crc, length is 5 and da is 8<92>h0. This high level stimulus is converted into low-level data using packing. This low level data is just a array of bits or bytes. Creates test scenarios and tests for the functionality and identifies the transaction through the interface.

Drivers - The drivers translate the operations produced by the generator into the actual inputs for the design under verification. Generators create inputs at a high level of abstraction namely, as transactions like read write operation. The drivers convert this input into actual design inputs, as defined in the specification of the designs interface. If the generator generates read operation, then read task is called, in that, the DUT input pin "read_write" is asserted.

Monitor – Monitor reports the protocol violation and identifies all the transactions. Monitors are two types, Passive and active. Passive monitors do not drive any signals. Active monitors can drive the DUT signals. Sometimes this is also referred as receiver. Monitor converts the state of the design and its outputs to a transaction abstraction level so it can be stored in a 'score-boards' database to be checked later on. Monitor converts the pin level activities in to high level.

Checker: The monitor only monitors the interface protocol. It doesn't check the whether the data is same as expected data or not, as interface has nothing to do with the data. Checker converts the low level data to high-level data and validated the data. This operation of converting low-level data to high-level data is called Unpacking, which is reverse of packing operation. For example, if data is collected from all the commands of the burst operation and then the data is converted in to raw data, and all the sub fields information are extracted from the data and compared against the expected values. The comparison state is sent to scoreboard.

Note: These intermediate blocks are optional and can get more complex depending on the complexity of the DUT.

In the below example for up_counter, we have designed three additional blocks namely program block, interface block and the Testbench. Some of the new features used in the code are being "bolded" out to easily notice.

The Generator, Agent, Driver, Monitor and Checker are all classes, modelled as Transactors. They are instantiated inside the Environment class (refer fig.1). For simplicity, the test is at the top of the hierarchy, as is the program that instantiates the Environment Class. The functional coverage definition can be put inside or outside the Environment class.

2.6 Working with Threads... Concept of FORK and JOIN

Classic Verilog has two ways of grouping statements - with a **begin...end** or **fork...join**. Statements in a beg b in...end run sequentially, while those in fork...join executes in parallel and the statements within the fork...join block has to be finished before the rest of the block can continue.

A System Verilog fork...join block always causes the process executing the fork statement to block until the termination of all forked processes. With the addition of the **join_any** and **join_none** keywords, SystemVerilog provides three choices for specifying when the parent (forking) process resumes execution. Refer the below diagram which shows the functioning of different types of fork and joins.

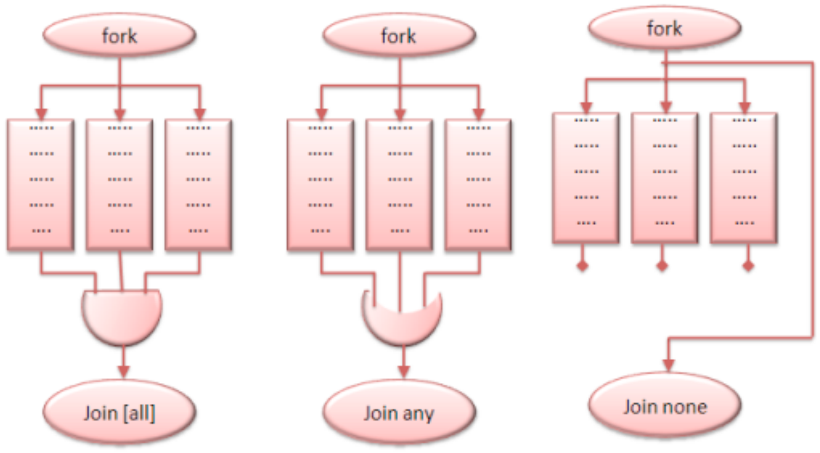


Fig 7: Fork and Join

How to kill process in fork/join?

Ans: Threads can be terminated at any time using a keyword disable.

Disable will be used with a label assigned to fork statement.

Eg: Example Code Snippet using Interaction of begin…end and fork...join

```
initial begin
        $display ("@%0d: start fork … join example", $time);
       #10 $display ("@%0d: start fork … join example", $time);
    fork
        display ("@%0d: parallel start", $time);
       #50 display ("@%0d: parallel after #50", $time);
       #10 display ("@%0d: parallel after #10", $time);
    begin
        #30 display ("@%0d: sequential after #30", $time);
       #10 display ("@%0d: sequential after #10", $time);
    end
    join
        display ("@%0d: after join", $time);
       display ("@%0d: final after #80", $time);
    end
```

Output:

```
@0: start fork … join example
@10: sequential after #10
@10: parallel start
@20: parallel after #10
@40: sequential after #30
@50: sequential after #10
@60: parallel after #50
@60: after join
@140: final after #80
```

2.7 Randomization: What to randomize

When you think of randomizing the stimulus to a design, the first things you may think of are the data fields. These are the easiest to create – just call $random. The problem is that this approach has a very low payback in terms of bugs found: you only find data-path bugs, perhaps with bit-level mistakes. The test is still inherently directed. The challenging bugs are in the control logic. As a result, you need to randomize all decision points in your DUT. Wherever control paths diverge, randomization increases the probability that you'll take a different path in each test case.

Refer the randomization_check () task in FIFO example later in the tutorial to get a better idea on the concept of randomization.

Difference between rand and randc?

The variables in the class can be declared random using the keywords: rand and randc. Dynamic and associative arrays can be declared using rand or randc keywords.

Variables declared with rand keywords are standard random variables. Their values are uniformly distributed over their range. Values declared with randc keyword are randomly distributed. They are cyclic in nature. They support only bit or enumerated data types. The size is limited.

2.8 Events

In Verilog a thread waits for an event with the @operator. This operator is edge sensitive, so it always blocks, waiting for the event to change. Another thread triggers the event with the -> operator, unblocking the first thread.

System Verilog lets you play more with the event type variables. If you have two event type variables event1 and event2, you can:

1. Assign one to the other: event1 = event2;

In this case event1 is said to be merged with event2.

2. Reclaim an event: event = null;

An event type can be *reset,* by assigning a null value.

3. Compare two events: event1 = null;

Usual comparison operators can be used to compare the event with another event or null.

2.8.1 Waiting for an Even Trigger

This specific event gets triggered only with the occurrence of that specific event. Instead of edge sensitive block e1, using the level sensitive wait.(e1.triggered) as shown below.

Example Code Snippet using Events:

```
event e1, e2;
initial begin
   $display (“\n @%0d: 1: before trigger”, $time);
   -> e1;
   wait (e2.triggered);
   $display(“@%0d: 1 : after trigger”, $time);
end

initial begin
   $display (“\n @%0d: 2: before trigger”, $time);
   -> e2;
   wait (e1.triggered);
   $display(“@%0d: 2 : after trigger”, $time);
end
```

Output:

```
@0 : 1 : before trigger
@0 : 2 : before trigger
@0 : 1 : after trigger
@0 : 2 : after trigger
```

2.8.2 Passing Events

Events can also be passed as arguments to other routines or modules.

Lets see how do we pass an event through a Generator, which is similar to the example what we used for FIFO.

```
class Generator;
   event done;
   function new (event done) ;     // Pass event from the  SV Testbench
        this.done = done;
   endfucntion


task reset;
   fork begin
        .........                 // perform Reset operations on the design
  -> done;                // Tell the test that resetting is done
  end
endtask
endclass
program automatic test;
    event gen_done;
    Generator gen;
   initial begin
          gen = new (gen_done);      // Creating a new construct
          gen.reset;      // Calls the reset function available with the class Generator
          wait(gen_done.triggered);
   end
endprogram
```

2.8.3 Semaphores

A semaphore allows you to control access to a resource. Semaphores can be used a testbench when you have a resource, such as a bus, that may have multiple requestors from inside the testbench but, as part of the physical design, can only have one driver. In System Verilog, a thread that requests a key when one is not available always block.

Semaphores can be used in a testbench when you have a resource, such as a bus, that may have multiple requestors from inside the testbench but as part of the physical design, can only has one driver. There are three basic operations for a semaphore. We create a semaphore with one or more keys using the new method get one or more keys with ***get,*** and return one or more keys with ***put.***

Eg: Semaphore

```
program automatic test (...)
    semaphore sem;
    initial begin
          sem = new(1);    //Allocate with 1 key
          fork
              ......
              ......
          join
      end
      task send;
            sem.get(1);
            ....
            ....
            sem.put(1);
      endtask
   endprogram
```

2.9 Tasks and Functions

Task and Function declarations are similar as in Verilog but following rules hold good for system verilog.

- Any port is seen as **input** default port direction, unless explicitly declared as other types. Eg: in, out, inout, ref.
- Unless declared, data types of ports are of **logic** type.
- There is **no need** to use **begin..end** when more then one statement is used inside a task.
- A task can be terminated before endtask, by usage of return statement.
- **Wire** data type cannot be used inside the port list.

Eg: Task

```
module task_intro ();

initial begin
#1 doInit(4,5);
#1 doInit(9,6);
#1 $finish;
end


task doInit(input bit [3:0] count, delay);
automatic reg [7:0] a;
```

```
if (count > 5) begin
$display ( " @ %t ns  Returning from task", $time);
return;
end
#5 $display ("@ %t ns Value passed is %d\n", $time, count);
endtask

endmodule
```

Output:

```
@6 ns Value passed is 4
@7 ns Returning from task
```

Eg: Function

```
module task_intro ();

bit a;

initial begin
#1 a = doInit(4,5);
#1 a = doInit(9,6);
#1 $finish;
end

function bit unsigned doInit(bit [3:0] count, add);
automatic reg [7:0] b;
if (count > 5) begin
$display ( " @ %t ns  Returning from task", $time);
return 0;
end
b = add;
#5 $display ("@ %t ns Value passed is %d\n", $time, count + b);
doInit = 1;
endfunction

endmodule
```

Output:

```
@1 ns Value passed is   9
@2 ns Returning from function
```

With the above examples we can clearly define where task and functions can be used effectively.

3. System Verilog Assertion (SVA):

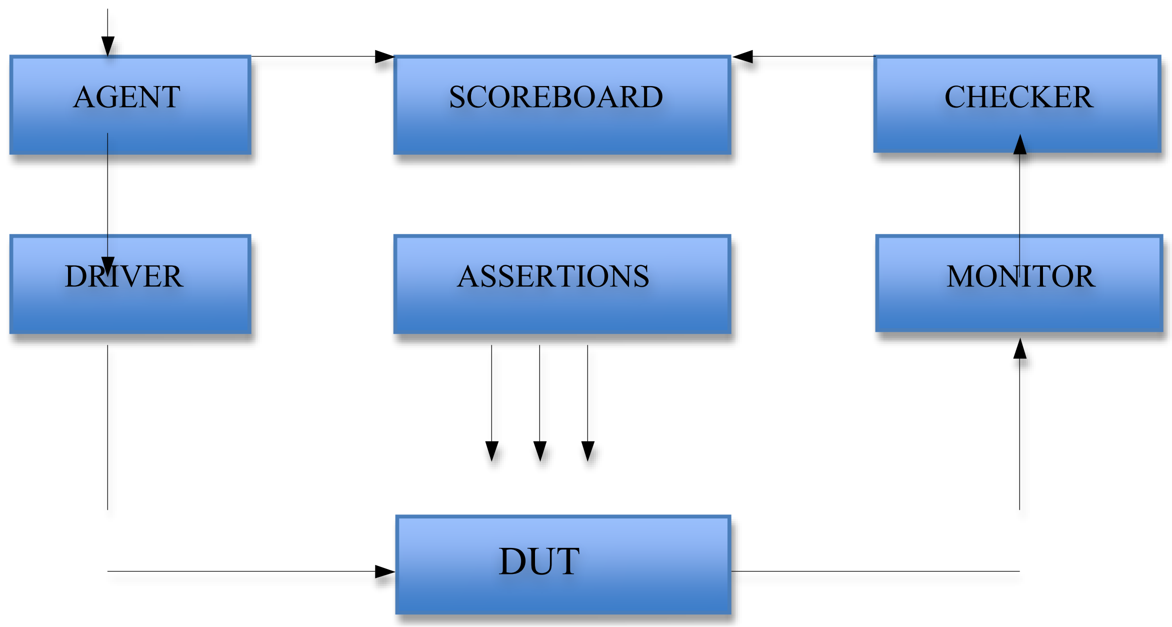


Fig 8: Assertions Block Diagram

What is an Assertion?

Assertions are mechanism or tool used by HDL's (VHDL and Verilog) to detect a design's expected behavior.

- If a property that is being checked for in a simulation does not behave the way we expect it to, the assertion fails.
- If a property that is forbidden from happening in a design happens during simulation, the assertion fails.
- It helps capturing the designer's interpretation of the specification.
- Describes the property of the design
- Assertion doesn't help in designing any entity but it checks for the behavior of the design.

Eg: Output of decoder should not have more than 1 bit turned ON at any time.

Where should the Assertions be used?

- Between modules, DUT and Testbench to check communication between the modules and stimulus constraints.
- It can also be used inside individual modules to verify the design, corner-cases and verify the assumptions.

How does Assertions looks like????

SVA Syntax Example:

assert property ******** ***keywords*********

(@(posedge clk) $rose(req) |-> ##[1:3] $rose(ack));

In the above example, when there is a positive edge on the Request (req) signal, then make sure that between 1 and 3 clock cycles later, there is a positive edge on acknowledge (ack) signal.
Here the designer knows that the acknowledge signal should go high within 1 to 3 cycles as soon as the Request signal has gone high at the positive edge.

3.1 Types of Assertions:

3.1.1 Immediate Assertions:

Immediate Assertions check the state of a condition at a given point in time.

- Based on simulation event semantics.
- Test expression is evaluated just like any other. Verilog expression with a procedural block. Those are not temporal in nature and are evaluated immediately.
- Have to be placed in a procedural block definition.
- Used only with dynamic simulation

A sample immediate assertion is shown below:

```
always_comb
begin
        a_ia: assert (a && b);
end
```

The immediate assertion a_ia is written as part of a procedural block and it follows the same event schedule of signal “a” and “b”. The always block executes if either signal “a” or signal “b” changes. The keyword that differentiates the immediate assertion from the concurrent assertion is ***property.***

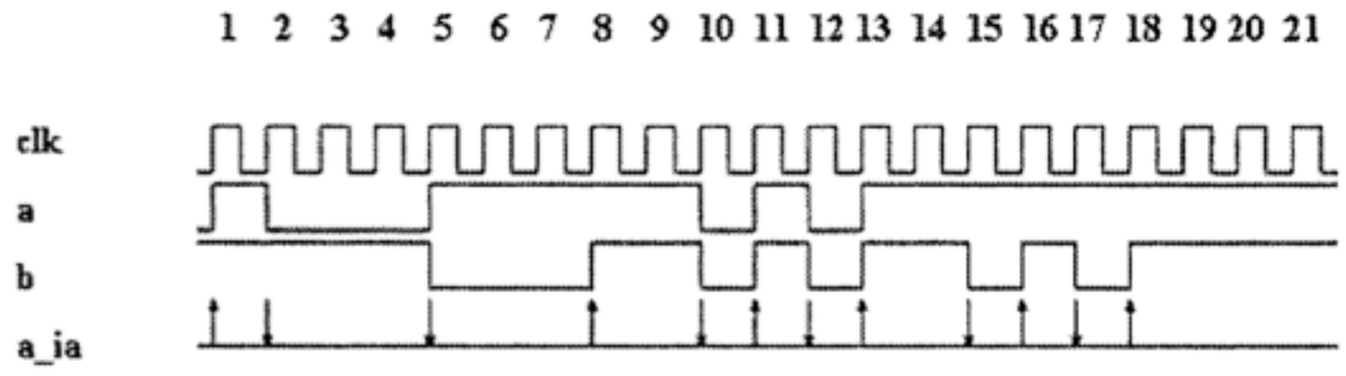


Fig 9

3.1.2 Concurrent Assertions:

Concurrent Assertions provide a means whereby a condition can be checked overtime.

- Based on Clock Cycles.
- Test expression is evaluated at clock edges based on the sampled values of the variables involved.
- Sampling of variables is done in the " observed region" of the scheduler.
- Can be placed in a procedural block, a module, an interface or a program definition.
- Can be used with both static and dynamic verification tool.

A sample of Concurrent Assertion:

a_cc: assert property (@ (posedge clk) not (a && b)) ;

Above example shows the result of concurrent assertion a_cc. All successes are shown with an up arrow and all features are shown with a down arrow. The key concept in this example is that property being verified on every positive edge of the clock irrespective of whether or not signal "a" and signal "b" changes.

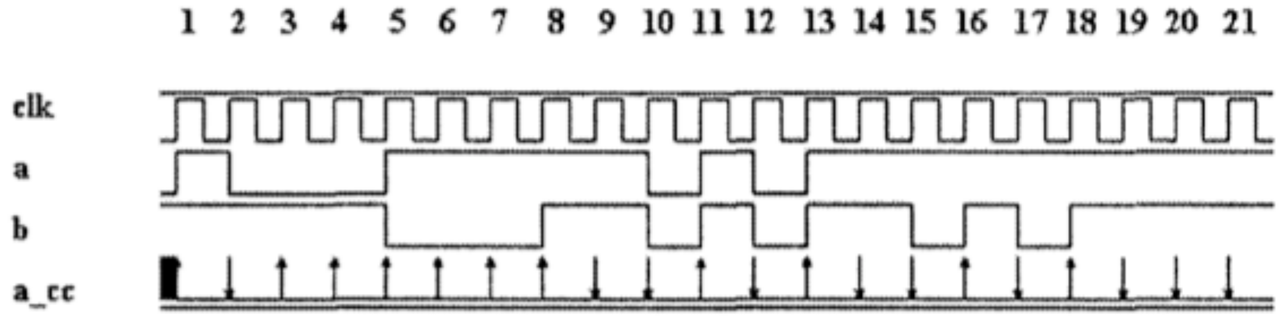


Fig 10

A simple Action block:

The System Verilog language is defined such that, every time an assertion check fails, the simulator is expected to print out an error message by default. The simulator need not print anything upon a success of an assertion. A user can also print a custom error or success message using the **"action block"** in the assert statement. The basic syntax of an **action block** is shown below.

```
assertion name :
        assert property (property_name)
                <success message>;
        else
                <fail messages>;
```

For example:

```
property p7;
  @(posedge clk)      a##2 b;
endproperty

a7: assert property (p7)
    $display (" Property p7 succeeded\n");
    else
    $display ("property p7 failed\n");
```

3.2 Implication Operator:

In the above example property p7 we can notice:

- the property looks for a valid start of the sequence on every positive edge of the clock. In this case, it looks for signal "a" to be high on every positive clock edge.
- If signal "a" is not high on any given positive clock edge, an error is issued by the checker. This error just means that we did not get a valid starting point for the checker at this clock. Whole these errors are benign; they can log a lot of error messages over time, since the check is performed on every clock edge. To avoid these errors, some kind of gating technique needs to be defined, which will ignore the check if a valid starting point is not present.

SVA provides a technique to achieve this goal. This technique is called **Implication.** Implication is equivalent to an if-then structure. The left hand side of the implication is called the **"antecedent"** and the right hand side is called the **"consequent"**. The antecedent is the gating condition. If the antecedent succeeds, then the consequent is evaluated. If the antecedent does not succeed, then the property is assumed to succeed by default. This is called a "vacuous success".
While implication avoids unnecessary error messages, it can produce vacuous successes. ***The implication construct can be used only with property definitions. It cannot be used only with property definitions.***

3.3 Two Types of Implication:

3.3.1 Overlapped Implication:

It is denoted by the symbol |->. If there is a match on the antecedent, then the consequent expression is evaluated in the same clock cycle. A simple example is shown below in property p8. This property checks if signal "a" is high on a given positive clock edge, then signal "b" should also be high on the same clock edge.

```
    property p8;
       @ (posedge clk)     a | -> b;
    endproperty
a8: assert property (p8);
```

3.3.2 Non-overlapped Implication:

It is denoted by the symbol |=>. If there is a match on the antecedent, then the consequent expression is evaluated in the next clock cycle. A delay of one clock cycle is assumed for the evaluation of the consequent expression. A simple example is shown below in property p9. This property checks that, if signal "a" is high on a given positive clock edge, then signal "b" should be high on the next clock edge.

```
    property p9;
      @(posedge clk)      a|=> b;
    endproperty
    a9:     assert property(p9);
```

The "$past" construct

SVA provides a built in system task called **$past** that is capable of getting values of signals from previous clock cycles. By default, it provides the value of the signal from the previous clock cycle. The simple syntax of the construct is as follows.

$past (signal_name, number of clock cycles)

This task can be used effectively to verify that the path taken by the design to get to the state in this current clock cycle is valid. Property p19 checks that in the given clock edge, if the expression (c && d) is true, then 2 cycles before that, the expression (a && b) was true.

```
property p19;
   @(posedge clk)      (c && d)        | - >
         ($past ((a&&b), 2) == 1'b1);
endproperty
a19: assert property (p19);
```

The "$past" construct

The simple syntax of the construct is as follows.

$past (signal_name, number of clock cycles, gating signal)

```
property p20;
         @(posedge clk)          (c && d) |->
                  ($past {{a&& b), 2, e) == 1'b1);
endproperty
a20: assert property(p20);
```

3.4 Mailboxes:

A mailbox is a communication mechanism that allows messages to be exchanged between processes or threads. Data can be sent to a mailbox by one process and retrieved by another.

Mailbox is a built-in class that provides the following methods:

- Create a mailbox: new()
- Place a message in a mailbox: put()
- Try to place a message in a mailbox without blocking: try_put()
- Retrieve a message from a mailbox: get() or peek()
- Try to retrieve a message from a mailbox without blocking: try_get() or try_peek()
- Retrieve the number of messages in the mailbox: num()

Eg: Generator using mailboxes

```
task generator(int n, mailbox mbx);
    Transacion t;
     repeat (...) begin
     t = new();
     .....
     mbx.put(t);
  end
endtask
```

Note: There are many other operators, which come in handy to perform specific operations. Need to explore the features of SVA depending on the complexity of the RTL design.

3.5 Structured Flow to develop a System Verilog Testbench:

The basic components which needs to be included in the System Verilog Testbench is as follows:

- Random Stimulus (Packet)
- Generator (The Generator should generate the random stimulus from Random Stimulus Packet Class). Generator should send this Stimulus to Driver through some means (using Mailbox or Callbacks).
- The Driver should driver this stimulus to DUT using Virtual Interface
- The Driver should send this stimulus to the Scoreboard using Mailbox or Callback
- The Monitor should monitor the output port/s and re-assemble the Actual Data and send it to Scoreboard using Mailbox or Callback as an Actual Data
- The scoreboard should compare the cActual Versus Expected Data.

3.5 Guide to write a System Verilog Verification Testbench for Synchronous FIFO:

Lets answer some of the basic questions, which need to be answered before starting to design our Testbench for FIFO Verilog code.

Q. What is the intention of this design? (Buffering the TX and RX or whatever. See the code, as you don't have design specs)

Ans: We need to know what we are expecting from the design. What are the intended inputs and what outputs are being expected from the design? Will there be any specific conditions, where the design might fail for certain inputs?

Q. What is the functionality implemented in the design?

Ans: Basic Functionalities of FIFO are like READ, WRITE and RESET etc.

Q. What are the channels available to access the DUT?

Ans: List down all the inputs and outputs, which connects the Design Under Test (Eg: FIFO) and your system verilog Testbench.

Q. What should be the nature of your stimulus?

Ans: As we need to implement functionalities like READ, WRITE, RESET etc.., we need to decide what stimulus need to provided to perform these operations and also should answer, do I have to encapsulate Stimulus at a higher level and then break them down at the driver level.

Q. What should be monitored to verify the functionality?

Ans: List down the output ports to verify your design functionality. For FIFO, we will verify cases for example like read only, write only, if the fifo is full are we able to write, if the fifo is empty are we able to read, I both read and write and asserted what will be expected, what will happen to reset of fifo and so on.

Q. How I will interface my Testbench with DUT? Off-course I need an interface block. How many interfaces are there?

Ans: Interface block is nothing but can easily imagined as wires, which need to connect two separate blocks to communicate with each other. So in our present design of FIFO Testbench, how many such interfaces are needed to communicate our FIFO DUT with the Testbench block.

Hint: We will of-course use a single interface block, but multiple modports. One modport for DUT (All the inputs to the DUT are outputs of this clocking block modport). Another modport for Driver (The driver will need only wires to drive the inputs of the DUT. Therefore this modport should have only DUT inputs) and last modport as Monitor (), this modport will have all the FIFO signals

Q. What should be my bandwidth to provide the stimulus to guarantee the availability of data to DUT? Are there any specific requirements?

Ans: Frequency bandwidth has to be selected very carefully, as we don’t want any data to be lost during the process of READ and WRITE at every clock edge. Enough time should be given to the driver to perform the operations, and sufficient bandwidth has to be considered.

Q. Do I need to insert any monitor inside the DUT for any functionality coverage? Do I need assertions? Where will I put those assertions? Inside the RTL or at interface or at monitor or where?

Ans: Refer Assertions section to get a brief idea. Yes Assertions are necessary to do a functional coverage test. Assertions need to be triggered inside the monitor block to check for specific conditions.

Q. How many testcases do I need?

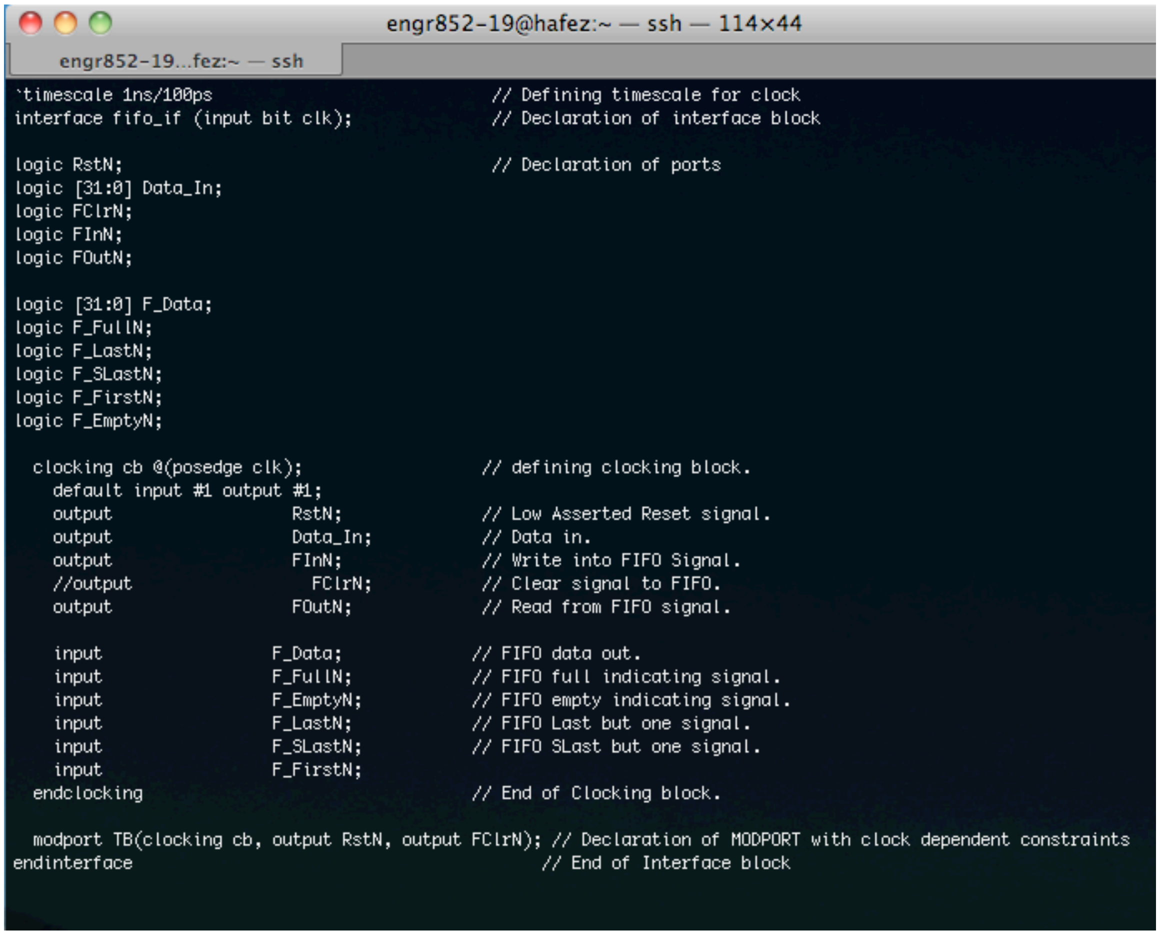
Ans: Always remember your Testbench checks the functionality of your design, and each corner has to be tested; hence every functionality needs a testcase to be tested.

The last and most important point is: How should I build the individual components (Generators, Drivers, Monitors and Scoreboard) in such a way that if any behavioral change needed, I should be able to change the behavior directly from test, without touching a single line of code inside the Testbench. Remember this, Testcase is the behavioral part and Testbench environment is the structural part. Now we have all the necessary details about the requirements, lets begin to write the different layers of our Testbench.

3.6 Step-by-Step Layered Approach for System Verilog Testbench:

1. Write a module called "interface.sv" which acts as an interface of the DUT with one MODPORT.

You guys will be familiar by now, what is the functionality of interface block, how the block looks like and importance of MODPORT. On completion of your interface block, it should look as below in Fig 11: ***interface.sv***



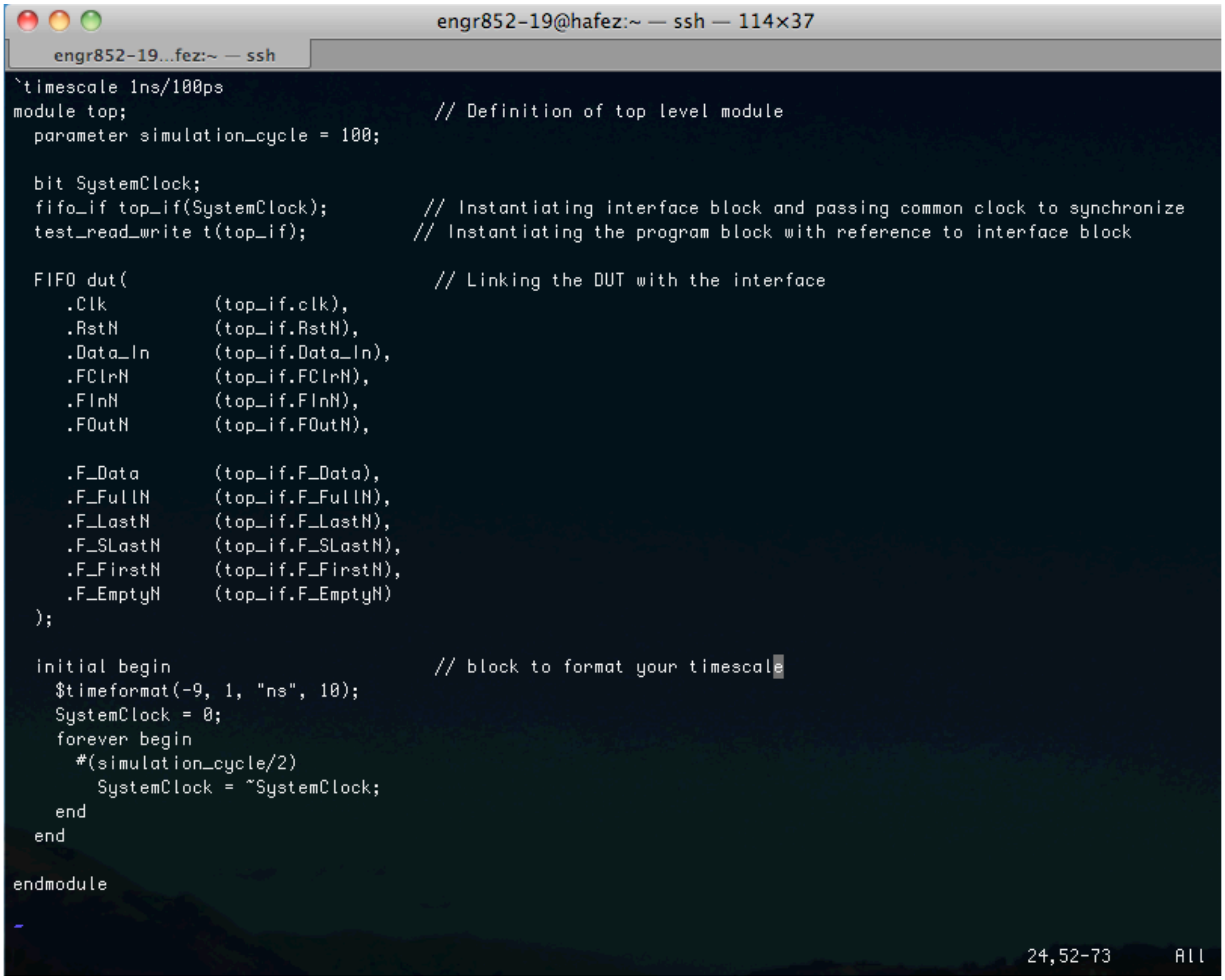
```
`timescale 1ns/100ps                              // Defining timescale for clock
interface fifo_if (input bit clk);                // Declaration of interface block

logic RstN;                                       // Declaration of ports
logic [31:0] Data_In;
logic FClrN;
logic FInN;
logic FOutN;

logic [31:0] F_Data;
logic F_FullN;
logic F_LastN;
logic F_SLastN;
logic F_FirstN;
logic F_EmptyN;

  clocking cb @(posedge clk);                    // defining clocking block.
    default input #1 output #1;
    output                  RstN;                // Low Asserted Reset signal.
    output                  Data_In;             // Data in.
    output                  FInN;                // Write into FIFO Signal.
    //output                  FClrN;             // Clear signal to FIFO.
    output                  FOutN;               // Read from FIFO signal.

    input                 F_Data;               // FIFO data out.
    input                 F_FullN;              // FIFO full indicating signal.
    input                 F_EmptyN;             // FIFO empty indicating signal.
    input                 F_LastN;              // FIFO Last but one signal.
    input                 F_SLastN;             // FIFO SLast but one signal.
    input                 F_FirstN;
  endclocking                                   // End of Clocking block.

  modport TB(clocking cb, output RstN, output FClrN); // Declaration of MODPORT with clock dependent constraints
endinterface                                          // End of Interface block
```

Fig 11

2. Develop a top-level module and instantiate the “interface” developed above, the DUT (Eg: FIFO Verilog code) and program.

The top-level module as we know should comprise the interface instantiation, the DUT and the program block instantiation. (Program block will be discussed next). Refer Fig 12

top.sv

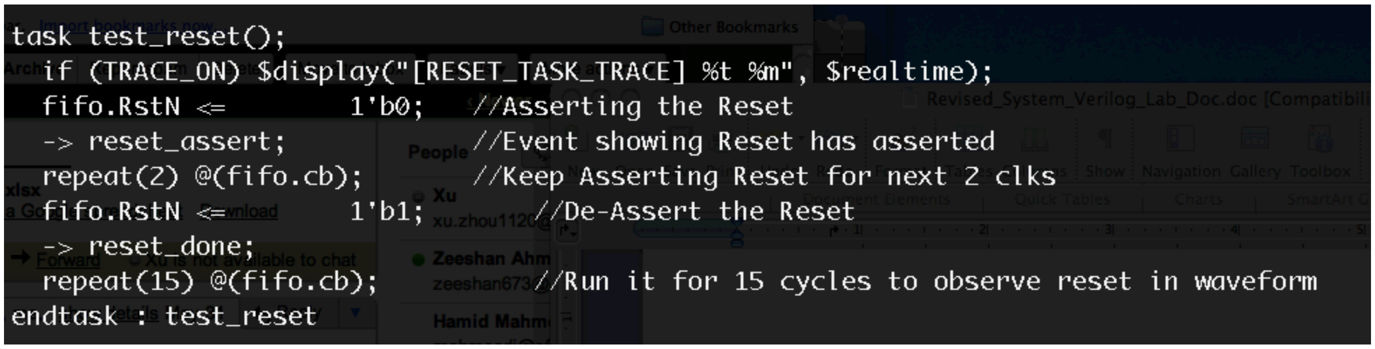


```
`timescale 1ns/100ps
module top;                              // Definition of top level module
  parameter simulation_cycle = 100;

  bit SystemClock;
  fifo_if top_if(SystemClock);          // Instantiating interface block and passing common clock to synchronize
  test_read_write t(top_if);           // Instantiating the program block with reference to interface block

  FIFO dut(                              // Linking the DUT with the interface
     .Clk          (top_if.clk),
     .RstN         (top_if.RstN),
     .Data_In      (top_if.Data_In),
     .FClrN        (top_if.FClrN),
     .FInN         (top_if.FInN),
     .FOutN        (top_if.FOutN),

     .F_Data       (top_if.F_Data),
     .F_FullN      (top_if.F_FullN),
     .F_LastN      (top_if.F_LastN),
     .F_SLastN     (top_if.F_SLastN),
     .F_FirstN     (top_if.F_FirstN),
     .F_EmptyN     (top_if.F_EmptyN)
  );

  initial begin                          // block to format your timescale
    $timeformat(-9, 1, "ns", 10);
    SystemClock = 0;
    forever begin
      #(simulation_cycle/2)
        SystemClock = ~SystemClock;
    end
  end

endmodule

~
                                                                                          24,52-73      All
```

Fig 12

3. Develop a program block, which contains all the necessary blocks like reset, monitor, drivers, generators etc. It’s a good practice to write separate tasks, so it can be called whenever it is required. All your testcases fall inside this block.

(i) Develop a task called **"test_reset"** inside the **program block**, which performs assertion operation when the waveforms are successfully asserted. Refer Fig 13.

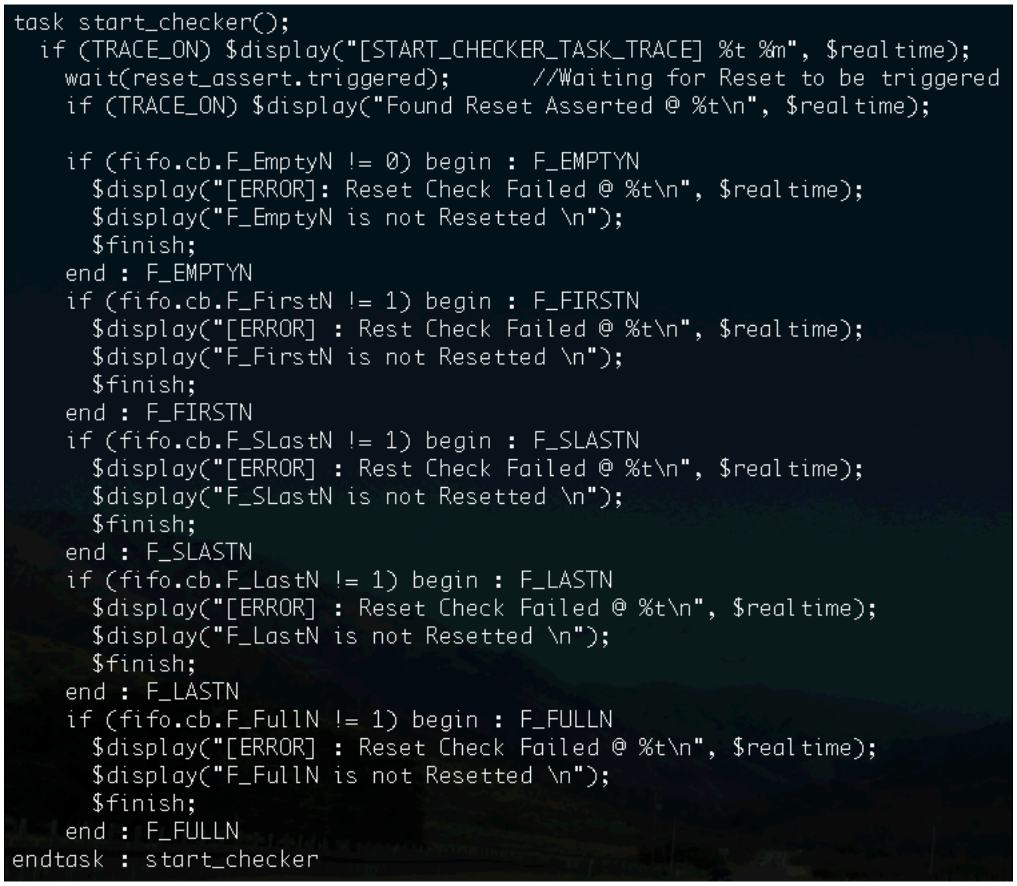


```
task test_reset();
  if (TRACE_ON) $display("[RESET_TASK_TRACE] %t %m", $realtime);
  fifo.RstN <=        1'b0;   //Asserting the Reset
  -> reset_assert;            //Event showing Reset has asserted
  repeat(2) @(fifo.cb);       //Keep Asserting Reset for next 2 clks
  fifo.RstN <=        1'b1;     //De-Assert the Reset
  -> reset_done;
  repeat(15) @(fifo.cb);      //Run it for 15 cycles to observe reset in waveform
endtask : test_reset
```

Fig 13

(ii) Develop a task called **"start_checker"** which checks all the possible parameters function properly on specific operation. For example, at reset operation all the parameters of FIFO have to be reset.

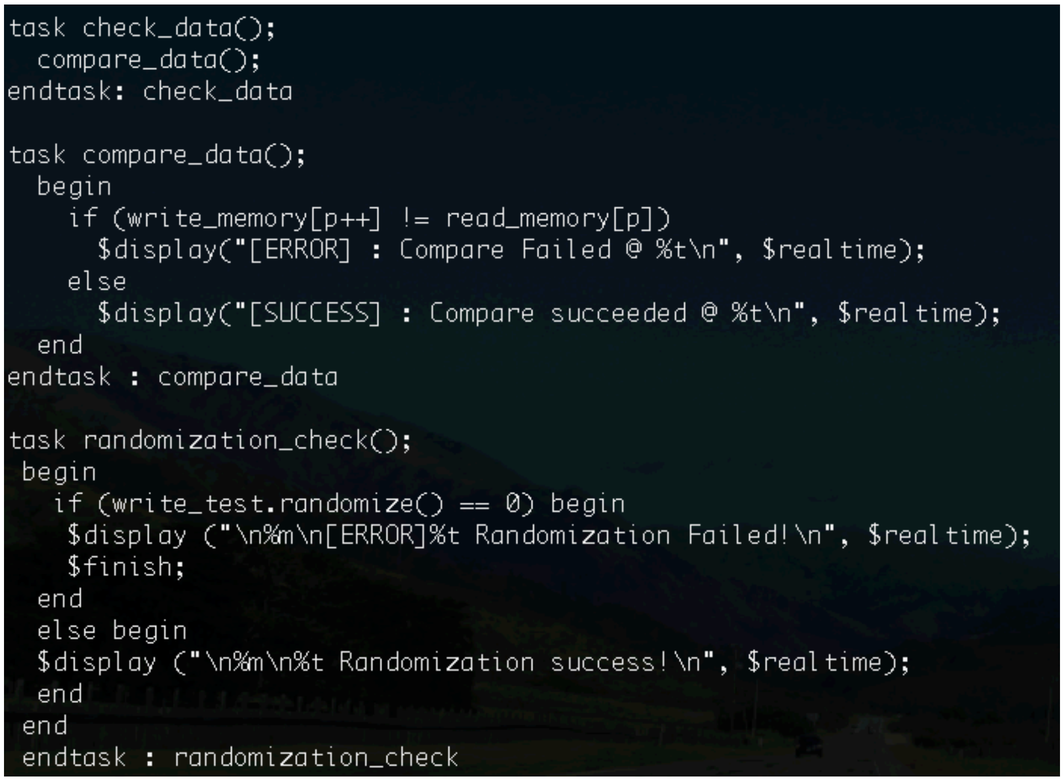
Note: The below example of the block, shown in Fig 14 where the parameters are accessed via the clocking block variable **"cb",** which is nothing but puts all the parameters sync up with the single clock.



```
task start_checker();
  if (TRACE_ON) $display("[START_CHECKER_TASK_TRACE] %t %m", $realtime);
    wait(reset_assert.triggered);      //Waiting for Reset to be triggered
    if (TRACE_ON) $display("Found Reset Asserted @ %t\n", $realtime);

    if (fifo.cb.F_EmptyN != 0) begin : F_EMPTYN
      $display("[ERROR]: Reset Check Failed @ %t\n", $realtime);
      $display("F_EmptyN is not Resetted \n");
      $finish;
    end : F_EMPTYN
    if (fifo.cb.F_FirstN != 1) begin : F_FIRSTN
      $display("[ERROR] : Rest Check Failed @ %t\n", $realtime);
      $display("F_FirstN is not Resetted \n");
      $finish;
    end : F_FIRSTN
    if (fifo.cb.F_SLastN != 1) begin : F_SLASTN
      $display("[ERROR] : Rest Check Failed @ %t\n", $realtime);
      $display("F_SLastN is not Resetted \n");
      $finish;
    end : F_SLASTN
    if (fifo.cb.F_LastN != 1) begin : F_LASTN
      $display("[ERROR] : Reset Check Failed @ %t\n", $realtime);
      $display("F_LastN is not Resetted \n");
      $finish;
    end : F_LASTN
    if (fifo.cb.F_FullN != 1) begin : F_FULLN
      $display("[ERROR] : Reset Check Failed @ %t\n", $realtime);
      $display("F_FullN is not Resetted \n");
      $finish;
    end : F_FULLN
endtask : start_checker
```

Fig 14

(iii) Develop another task called **“check_data”,** which basically does a comparison of two data. For example, to verify the data sent and received through FIFO is identical we can use this task in our verification. Refer Fig 15



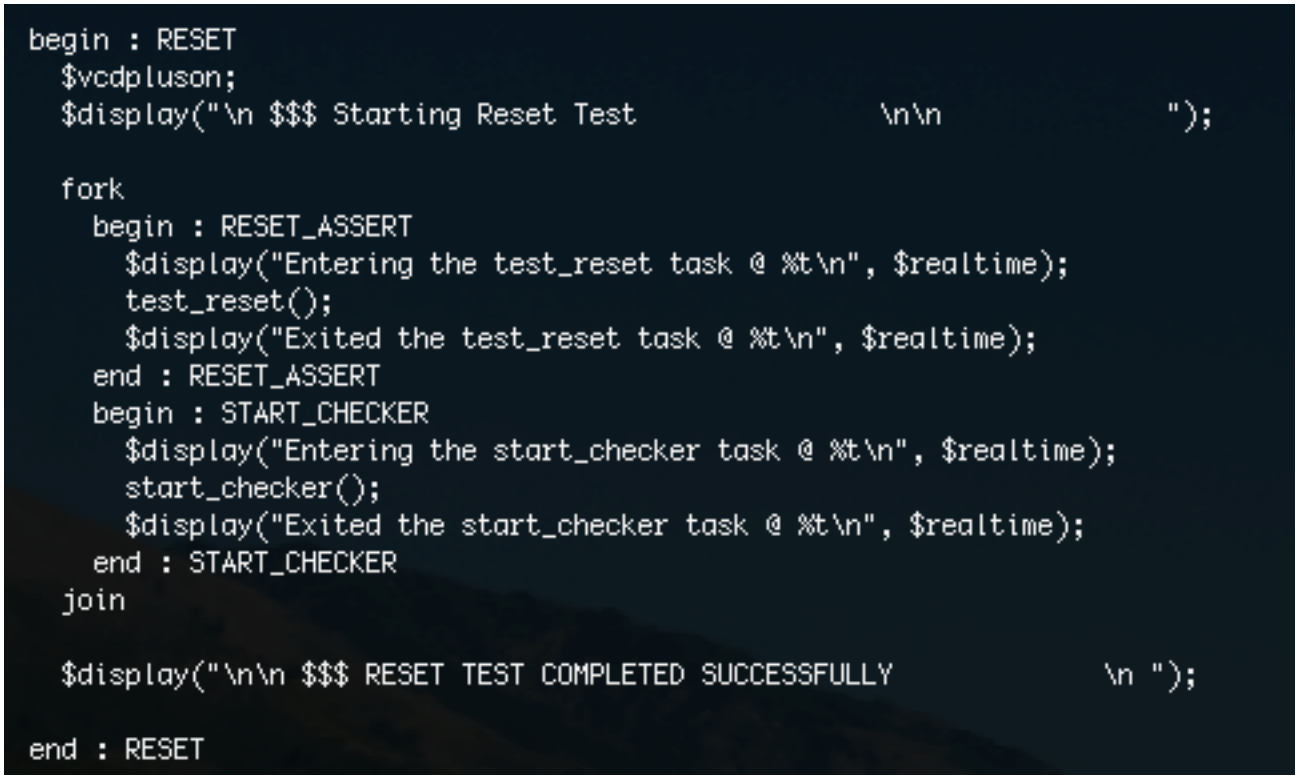
```
task check_data();
  compare_data();
endtask: check_data

task compare_data();
  begin
    if (write_memory[p++] != read_memory[p])
      $display("[ERROR] : Compare Failed @ %t\n", $realtime);
    else
      $display("[SUCCESS] : Compare succeeded @ %t\n", $realtime);
  end
endtask : compare_data

task randomization_check();
 begin
   if (write_test.randomize() == 0) begin
    $display ("\n%m\n[ERROR]%t Randomization Failed!\n", $realtime);
    $finish;
  end
  else begin
  $display ("\n%m\n%t Randomization success!\n", $realtime);
  end
 end
 endtask : randomization_check
```

Fig 15

(iv) Then finally write a small block of code, which calls all these above defined tasks in order to perform operation. Refer Fig 14

Note: This sequence of task calls are written inside a **fork & join** loop, which is basically blocking and the code inside the fork .. join block should be completed before executing the later steps. For more information on concept of fork & join, refer the tutorial above.

These above tasks can be called anywhere throughout your Testbench to perform specific functionalities



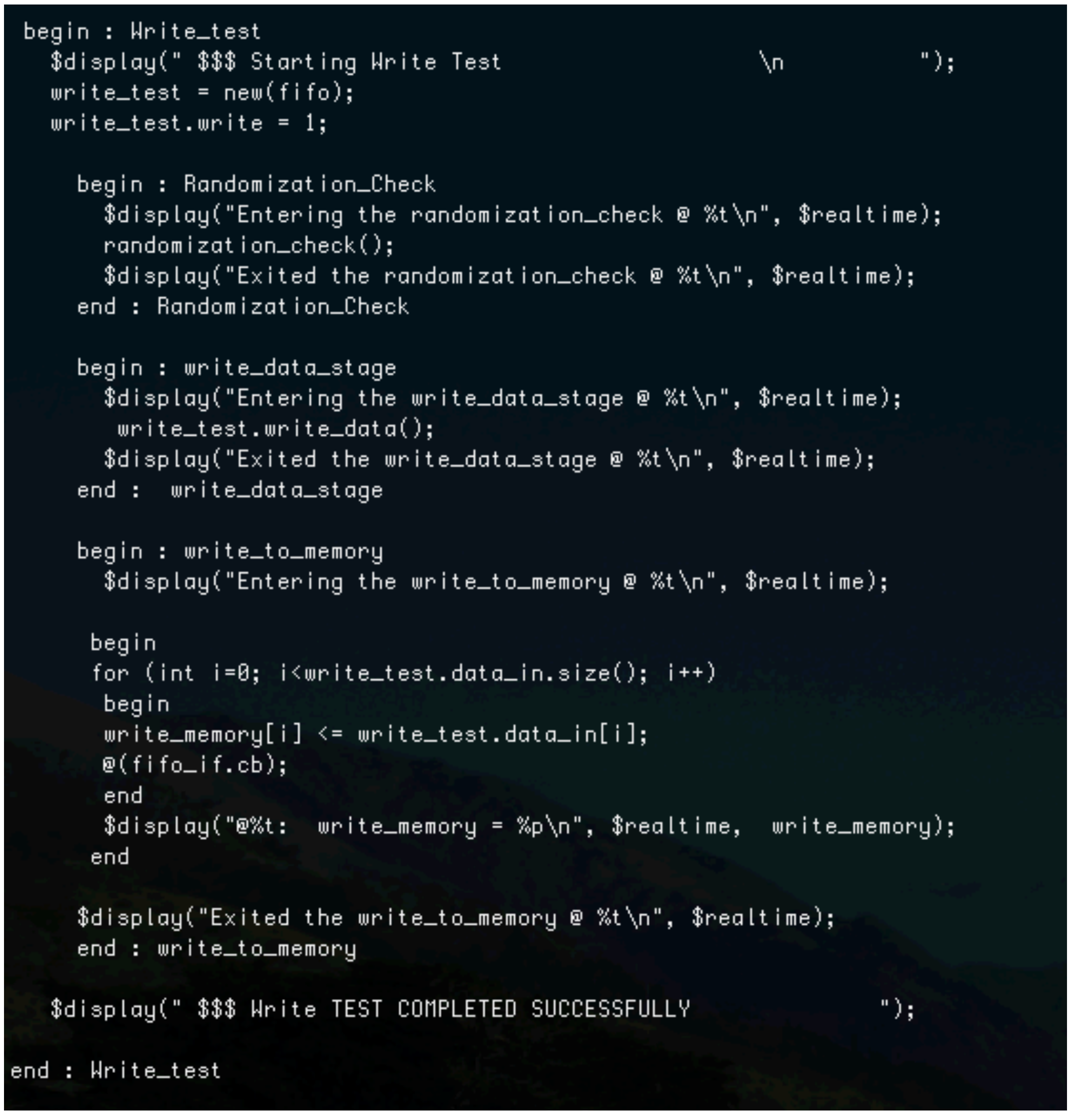
```
begin : RESET
  $vcdpluson;
  $display("\n $$$ Starting Reset Test               \n\n          ");

  fork
    begin : RESET_ASSERT
      $display("Entering the test_reset task @ %t\n", $realtime);
      test_reset();
      $display("Exited the test_reset task @ %t\n", $realtime);
    end : RESET_ASSERT
    begin : START_CHECKER
      $display("Entering the start_checker task @ %t\n", $realtime);
      start_checker();
      $display("Exited the start_checker task @ %t\n", $realtime);
    end : START_CHECKER
  join

  $display("\n\n $$$ RESET TEST COMPLETED SUCCESSFULLY             \n ");

end : RESET
```

Fig 16

(v) Then finally add all the necessary functionalities needed inside your Testbench. For example in FIFO, we need to test the writing process into FIFO and test the reading process from FIFO with different conditions like what happens when read operation is performed when FIFO is empty, write operation is done when FIFO is full etc.

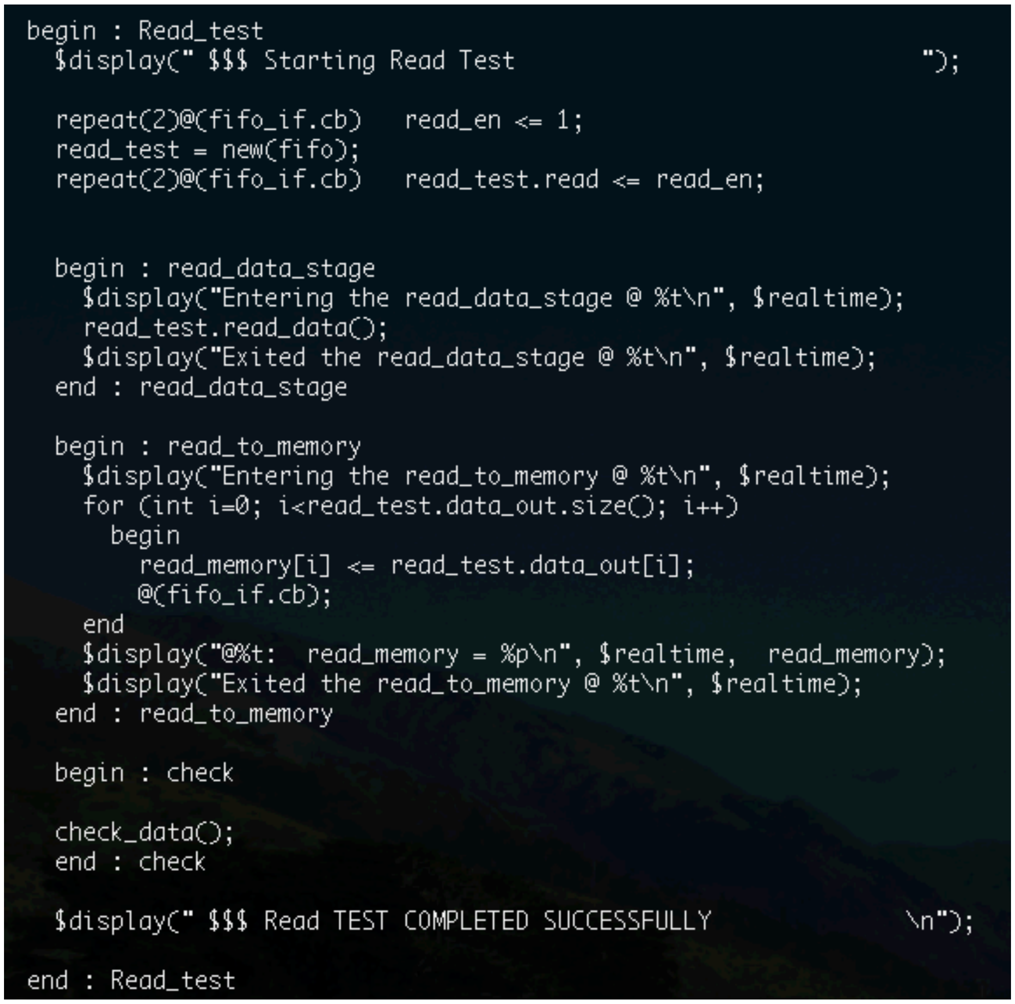
Refer the below code Fig 16 blocks to refine your idea on how to approach the test coverage of your design.

In Fig 17 & 18, the process of data writing into the FIFO memory and reading data from FIFO memory is demonstrated.



```
begin : Write_test
  $display(" $$$ Starting Write Test                   \n          ");
  write_test = new(fifo);
  write_test.write = 1;

    begin : Randomization_Check
      $display("Entering the randomization_check @ %t\n", $realtime);
      randomization_check();
      $display("Exited the randomization_check @ %t\n", $realtime);
    end : Randomization_Check

    begin : write_data_stage
      $display("Entering the write_data_stage @ %t\n", $realtime);
       write_test.write_data();
      $display("Exited the write_data_stage @ %t\n", $realtime);
    end :  write_data_stage

    begin : write_to_memory
      $display("Entering the write_to_memory @ %t\n", $realtime);

     begin
     for (int i=0; i<write_test.data_in.size(); i++)
      begin
      write_memory[i] <= write_test.data_in[i];
      @(fifo_if.cb);
      end
      $display("@%t:  write_memory = %p\n", $realtime,  write_memory);
     end

    $display("Exited the write_to_memory @ %t\n", $realtime);
    end : write_to_memory

  $display(" $$$ Write TEST COMPLETED SUCCESSFULLY              ");

end : Write_test
```

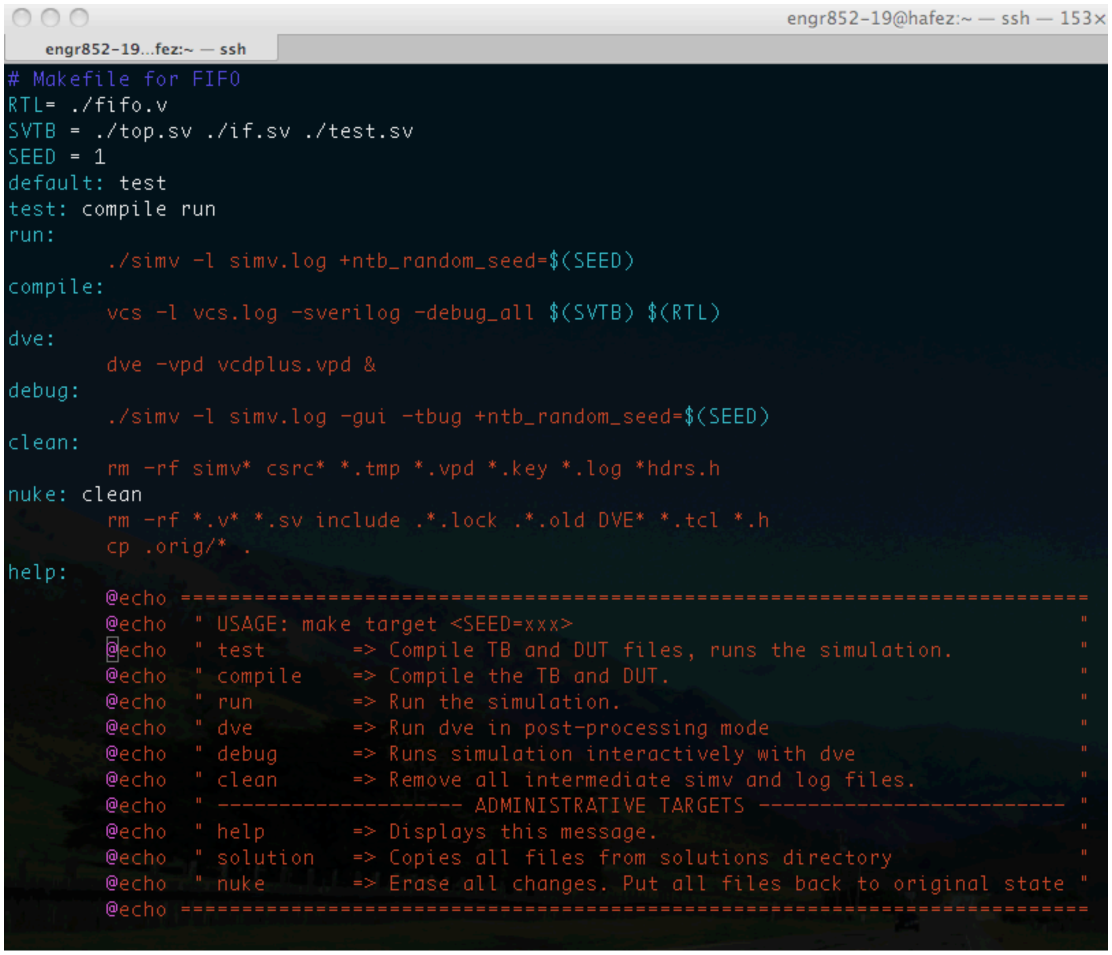
Fig 17



```
begin : Read_test
  $display(" $$$ Starting Read Test                              ");

  repeat(2)@(fifo_if.cb)   read_en <= 1;
  read_test = new(fifo);
  repeat(2)@(fifo_if.cb)   read_test.read <= read_en;


  begin : read_data_stage
    $display("Entering the read_data_stage @ %t\n", $realtime);
    read_test.read_data();
    $display("Exited the read_data_stage @ %t\n", $realtime);
  end : read_data_stage

  begin : read_to_memory
    $display("Entering the read_to_memory @ %t\n", $realtime);
    for (int i=0; i<read_test.data_out.size(); i++)
      begin
        read_memory[i] <= read_test.data_out[i];
        @(fifo_if.cb);
    end
    $display("@%t:  read_memory = %p\n", $realtime,  read_memory);
    $display("Exited the read_to_memory @ %t\n", $realtime);
  end : read_to_memory

  begin : check

  check_data();
  end : check

  $display(" $$$ Read TEST COMPLETED SUCCESSFULLY             \n");

end : Read_test
```

Fig 18

4. Once the basic tasks have been built inside another module called assertion_bind.sva, develop a “FIFO_check” module inside program block in which all the signals to be monitored using assertions.
5. Make the Unix "Makefile" so that the whole process will be automated as show below in Fig 17. If you carefully look into the Makefile, we can see the variables are being assigned to the commands and these variables get linked up when you call the ***“make”*** command. (To know how does Assertions works, look at the Assertions part of tutorial and some examples at the end of this tutorial)



```
# Makefile for FIFO
RTL= ./fifo.v
SVTB = ./top.sv ./if.sv ./test.sv
SEED = 1
default: test
test: compile run
run:
        ./simv -l simv.log +ntb_random_seed=$(SEED)
compile:
        vcs -l vcs.log -sverilog -debug_all $(SVTB) $(RTL)
dve:
        dve -vpd vcdplus.vpd &
debug:
        ./simv -l simv.log -gui -tbug +ntb_random_seed=$(SEED)
clean:
        rm -rf simv* csrc* *.tmp *.vpd *.key *.log *hdrs.h
nuke: clean
        rm -rf *.v* *.sv include .*.lock .*.old DVE* *.tcl *.h
        cp .orig/* .
help:
        @echo ==========================================================================
        @echo  " USAGE: make target <SEED=xxx>                                            "
        @echo  " test        => Compile TB and DUT files, runs the simulation.            "
        @echo  " compile     => Compile the TB and DUT.                                   "
        @echo  " run         => Run the simulation.                                       "
        @echo  " dve         => Run dve in post-processing mode                           "
        @echo  " debug       => Runs simulation interactively with dve                    "
        @echo  " clean       => Remove all intermediate simv and log files.               "
        @echo  " -------------------- ADMINISTRATIVE TARGETS ------------------------- "
        @echo  " help        => Displays this message.                                    "
        @echo  " solution    => Copies all files from solutions directory                 "
        @echo  " nuke        => Erase all changes. Put all files back to original state "
        @echo ==========================================================================
```

Fig 19: Makefile

Once all the blocks are set and being linked up with the help of ***Makefile***, we can execute all these files to check the functionality of our Testbench by executing the below commands.

Type ***ssh -X <username>@hafez.sfsu.edu*** on the shell window to login to your respective accounts**.**
On execution of above command, it prompts to enter your password. Note: Entry of password is not physically visible to the user. On successful login, user is now ready inside the server.
To activate the C shell, type the below command as shown. Without executing this command, all other commands seem to be invalid to the interpreter and throw an error message upon execution.

- ***[engr852-19@hafez ~]$ csh***

Source the Synopsys setup files using below command

- ***[engr852-19@hafez~]$source /packages/synopsys/setup/synopsys_setup.tcl***

I strongly suggest, giving a try and writing your own Testbench with the above references. Parallely, you can copy the solutions into your home directory by running the below command.

- ***[engr852-19@hafez~]$ cp -rpf /packages/synopsys/setup/verification_flow/System_Verilog/fifo_example ./***

Execute the below command where all the above said design files (Eg: FIFO design files) are being located in your Unix directory.

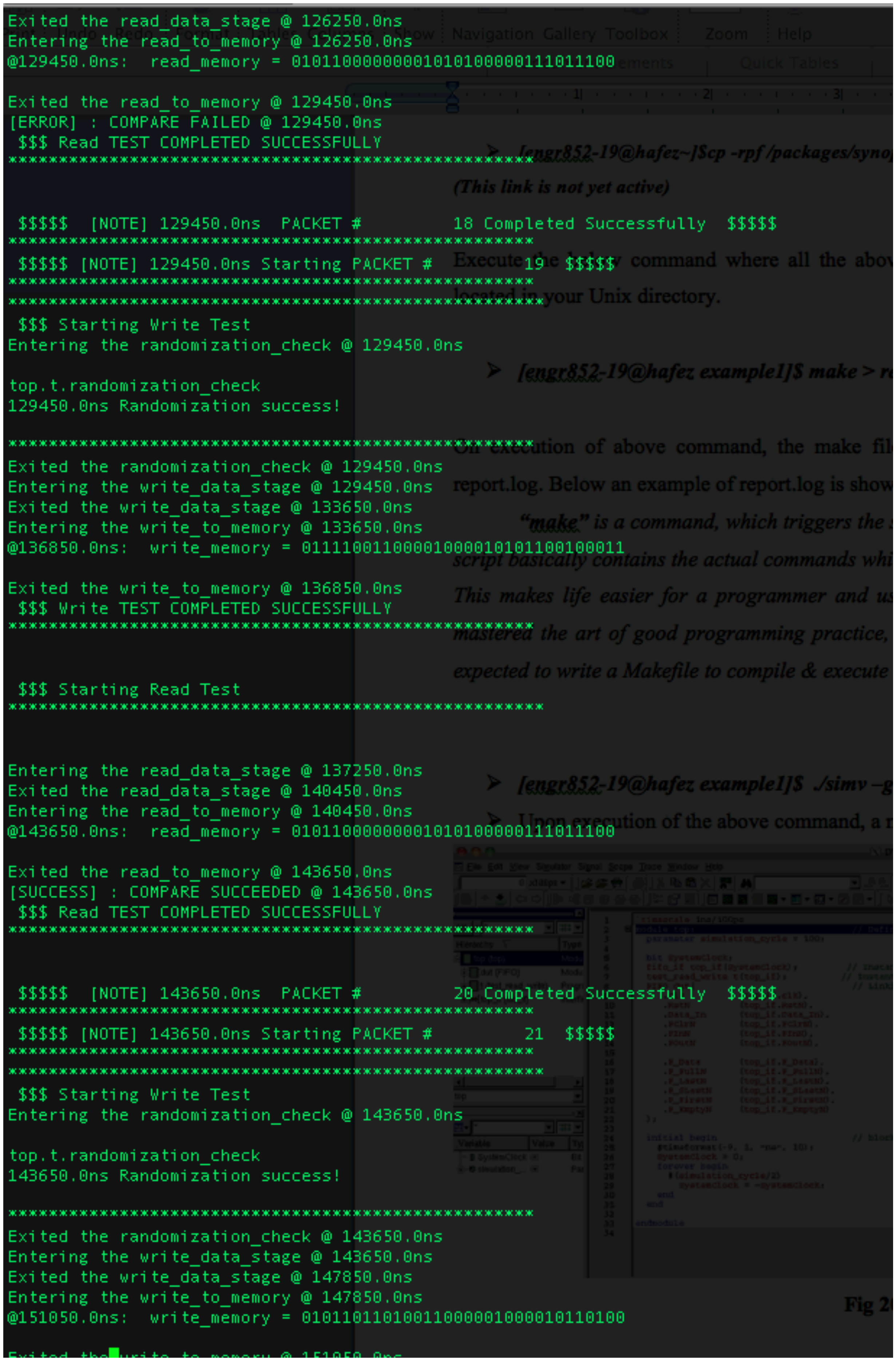
- ***[engr852-19@hafez example1]$ make > report.log***

On execution of above command, the make file is being executed and saves the output in report.log file. Below an example of report.log is shown for FIFO example.

"make" *is a command, which triggers the script called "Makefile" as shown above in Fig 19. This script basically contains the actual commands which needs to be executed, and are assigned to a variable. This makes life easier for a programmer and user to understand the design hierarchy. You have just mastered the art of good programming practice, by writing a Makefile for your design. A developer is expected to write a Makefile to compile & execute his/her project.*

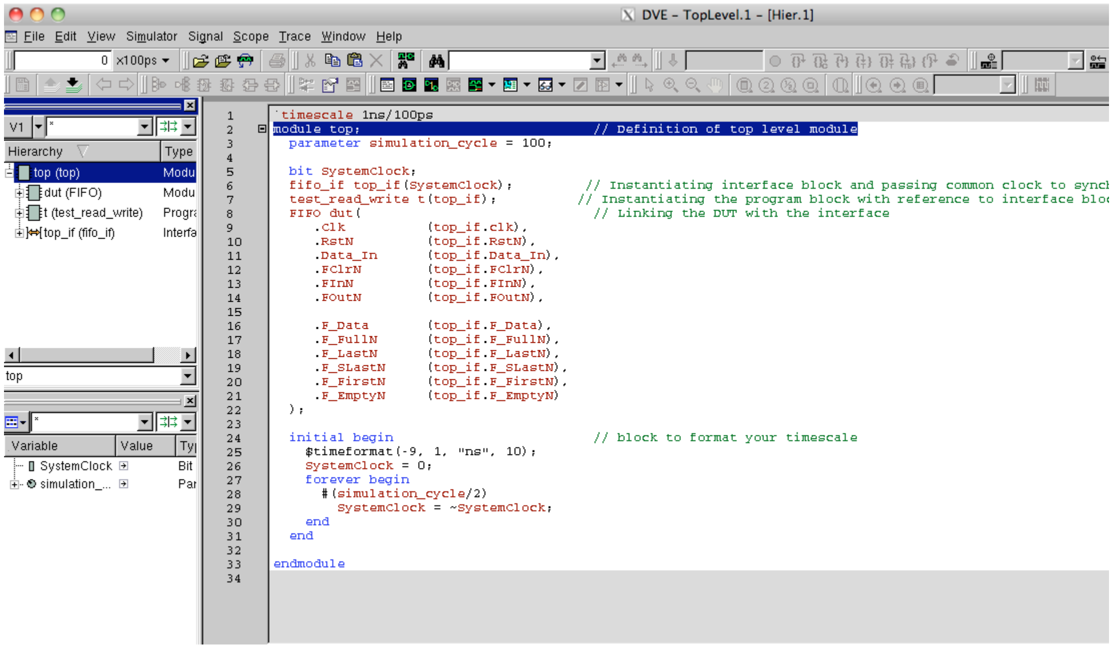
The *report.log* gives exact idea about the execution flow and which process is taking place at specific timing cycle. To make a successful testbench for a design, the requirements of both the testbench and the design should be common. If there is any change in the requirements, the testbench might not be able to test the design's all features successfully. With suitable error and success commands, we can check if the testbench works fine for the given design.

The testbench developer can write suitable display messages, which comes on the report.log when executed and helps to debug if any of the requirements are not matching.



```
Exited the read_data_stage @ 126250.0ns
Entering the read_to_memory @ 126250.0ns
@129450.0ns:  read_memory = 01011000000001010100000111011100

Exited the read_to_memory @ 129450.0ns
[ERROR] : COMPARE FAILED @ 129450.0ns
 $$$ Read TEST COMPLETED SUCCESSFULLY
*****************************************************


 $$$$$  [NOTE] 129450.0ns  PACKET #         18 Completed Successfully  $$$$$
*****************************************************
 $$$$$ [NOTE] 129450.0ns Starting PACKET #       19  $$$$$
*****************************************************
******************************************************
 $$$ Starting Write Test
Entering the randomization_check @ 129450.0ns

top.t.randomization_check
129450.0ns Randomization success!

*****************************************************
Exited the randomization_check @ 129450.0ns
Entering the write_data_stage @ 129450.0ns
Exited the write_data_stage @ 133650.0ns
Entering the write_to_memory @ 133650.0ns
@136850.0ns:  write_memory = 01111001100001000010101100100011

Exited the write_to_memory @ 136850.0ns
 $$$ Write TEST COMPLETED SUCCESSFULLY
*****************************************************


 $$$ Starting Read Test
******************************************************


Entering the read_data_stage @ 137250.0ns
Exited the read_data_stage @ 140450.0ns
Entering the read_to_memory @ 140450.0ns
@143650.0ns:  read_memory = 01011000000001010100000111011100

Exited the read_to_memory @ 143650.0ns
[SUCCESS] : COMPARE SUCCEEDED @ 143650.0ns
 $$$ Read TEST COMPLETED SUCCESSFULLY
*****************************************************


 $$$$$  [NOTE] 143650.0ns  PACKET #         20 Completed Successfully  $$$$$
*****************************************************
 $$$$$ [NOTE] 143650.0ns Starting PACKET #       21  $$$$$
*****************************************************
******************************************************
 $$$ Starting Write Test
Entering the randomization_check @ 143650.0ns

top.t.randomization_check
143650.0ns Randomization success!

*****************************************************
Exited the randomization_check @ 143650.0ns
Entering the write_data_stage @ 143650.0ns
Exited the write_data_stage @ 147850.0ns
Entering the write_to_memory @ 147850.0ns
@151050.0ns:  write_memory = 01011011010011000001000010110100
```

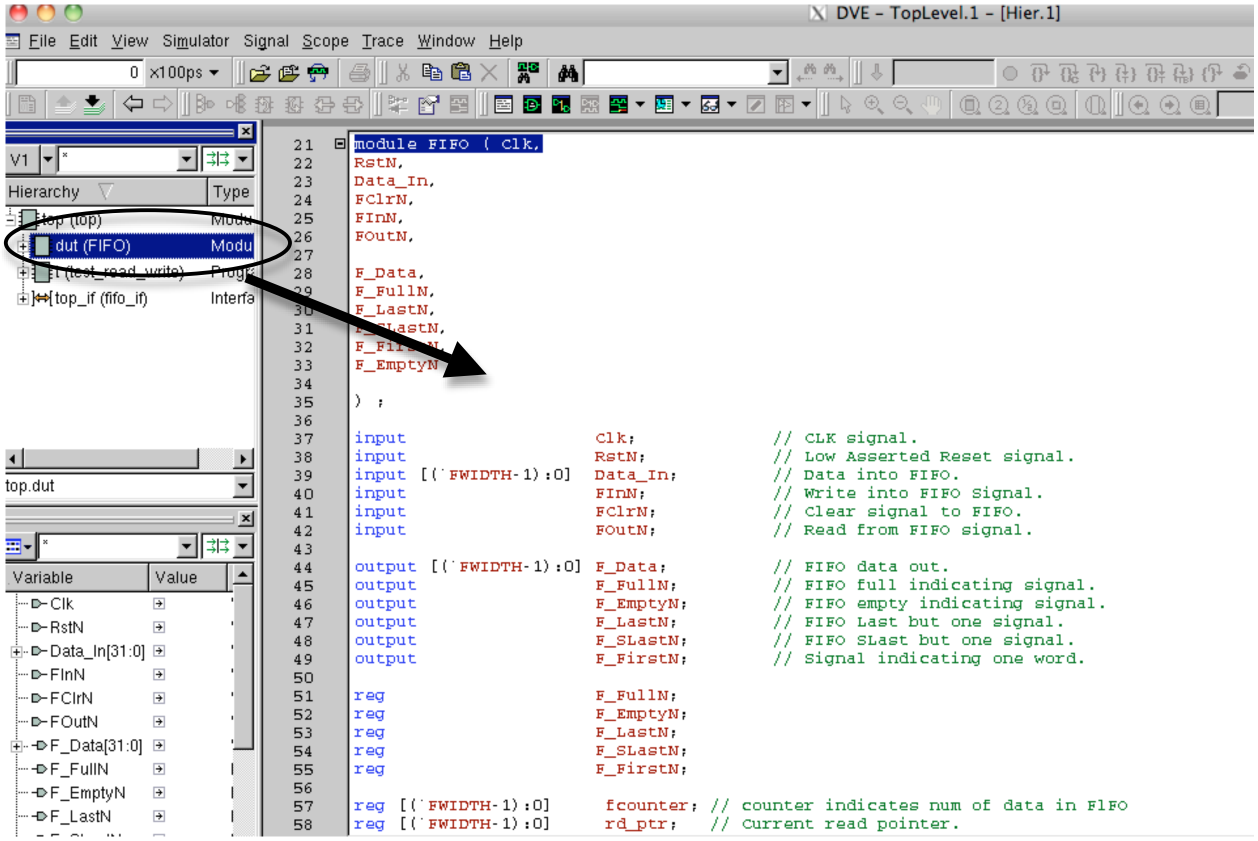
- ➢ ***[engr852-19@hafez example1]$ ./simv –gui***
- ➢ Upon execution of the above command, a new window opens as shown below



```
`timescale 1ns/100ps
module top;                                   // Definition of top level module
  parameter simulation_cycle = 100;

  bit SystemClock;
  fifo_if top_if(SystemClock);              // Instantiating interface block and passing common clock to synch
  test_read_write t(top_if);               // Instantiating the program block with reference to interface bloc
  FIFO dut(                                  // Linking the DUT with the interface
     .Clk          (top_if.clk),
     .RstN         (top_if.RstN),
     .Data_In      (top_if.Data_In),
     .FClrN        (top_if.FClrN),
     .FInN         (top_if.FInN),
     .FOutN        (top_if.FOutN),

     .F_Data       (top_if.F_Data),
     .F_FullN      (top_if.F_FullN),
     .F_LastN      (top_if.F_LastN),
     .F_SLastN     (top_if.F_SLastN),
     .F_FirstN     (top_if.F_FirstN),
     .F_EmptyN     (top_if.F_EmptyN)
  );

  initial begin                              // block to format your timescale
    $timeformat(-9, 1, "ns", 10);
    SystemClock = 0;
    forever begin
      #(simulation_cycle/2)
        SystemClock = ~SystemClock;
    end
  end

endmodule
```

Fig 20

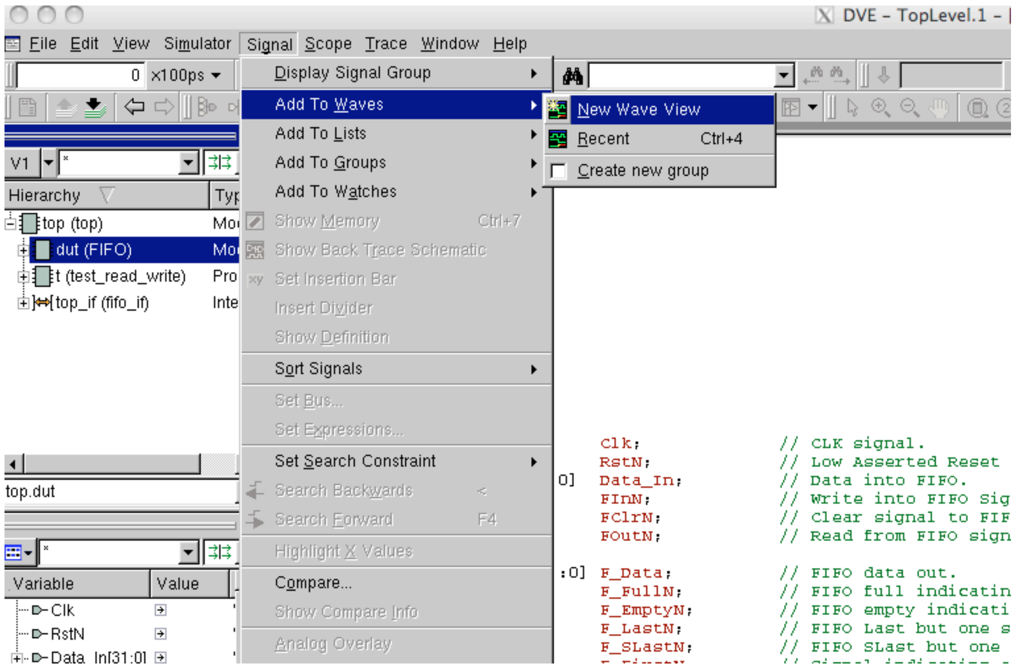
➢ Click the DUT of the design and drag it to the white space provided in the center as shown below, where you can view your code and modify it if necessary during the run.



```
module FIFO ( Clk,
RstN,
Data_In,
FClrN,
FInN,
FOutN,

F_Data,
F_FullN,
F_LastN,
F_SLastN,
F_FirstN,
F_EmptyN

) ;

input                   Clk;            // CLK signal.
input                   RstN;           // Low Asserted Reset signal.
input  [(`FWIDTH-1):0]  Data_In;        // Data into FIFO.
input                   FInN;           // Write into FIFO Signal.
input                   FClrN;          // Clear signal to FIFO.
input                   FOutN;          // Read from FIFO signal.

output  [(`FWIDTH-1):0] F_Data;         // FIFO data out.
output                  F_FullN;        // FIFO full indicating signal.
output                  F_EmptyN;       // FIFO empty indicating signal.
output                  F_LastN;        // FIFO Last but one signal.
output                  F_SLastN;       // FIFO SLast but one signal.
output                  F_FirstN;       // Signal indicating one word.

reg                     F_FullN;
reg                     F_EmptyN;
reg                     F_LastN;
reg                     F_SLastN;
reg                     F_FirstN;

reg  [(`FWIDTH-1):0]     fcounter; // counter indicates num of data in FIFO
reg  [(`FWIDTH-1):0]     rd_ptr;   // Current read pointer.
```

Fig 21

➢ Once DUT is loaded, click on tab Signal > Add to waves > New wave view to open new window showing the signals of your design.



```
   clk;              // CLK signal.
   RstN;             // Low Asserted Reset
0] Data_In;          // Data into FIFO.
   FInN;             // Write into FIFO sig
   FClrN;            // Clear signal to FIF
   FOutN;            // Read from FIFO sign

:0] F_Data;          // FIFO data out.
   F_FullN;          // FIFO full indicatin
   F_EmptyN;         // FIFO empty indicati
   F_LastN;          // FIFO Last but one s
   F_SLastN;         // FIFO SLast but one
```

Fig 22

➢ On the new window, click the specific buttons highlighted to simulate your design, scale your outputs to the exact window.

Simulator

Scaling

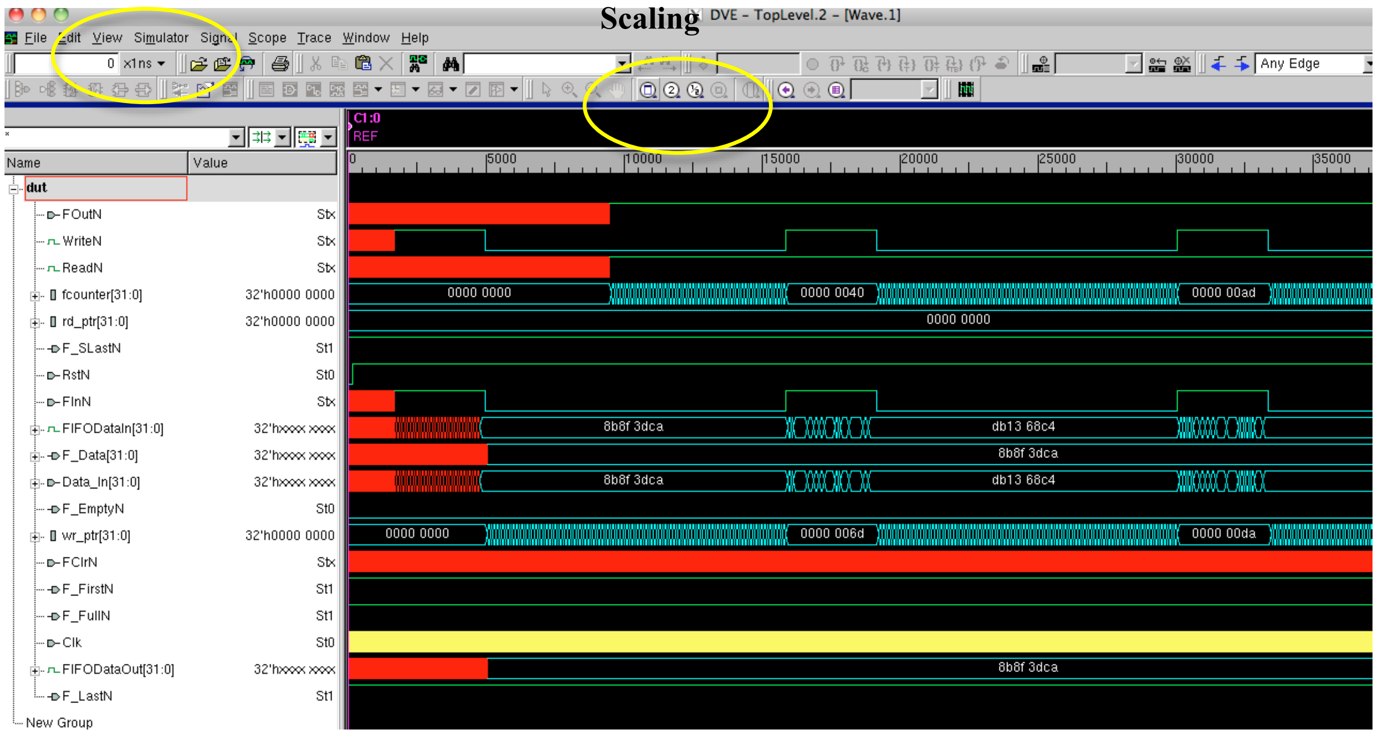

DVE - TopLevel.2 - [Wave.1]
File Edit View Simulator Signal Scope Trace Window Help
x1ns
Any Edge
C1:0
REF
Name
Value
dut
FOutN StX
WriteN StX
ReadN StX
fcounter[31:0] 32'h0000 0000
rd_ptr[31:0] 32'h0000 0000
F_SLastN St1
RstN St0
FInN StX
FIFODataIn[31:0] 32'hxxxx xxxx
F_Data[31:0] 32'hxxxx xxxx
Data_In[31:0] 32'hxxxx xxxx
F_EmptyN St0
wr_ptr[31:0] 32'h0000 0000
FClrN StX
F_FirstN St1
F_FullN St1
Clk St0
FIFODataOut[31:0] 32'hxxxx xxxx
F_LastN St1
New Group
0 5000 10000 15000 20000 25000 30000 35000
0000 0000
0000 0040
0000 00ad
8b8f 3dca
db13 68c4
0000 006d
0000 00da

Fig 23

Note: You can verify the outputs of the signals generated as per your design, if there any errors do the necessary changes, recompile the design and repeat the above steps again to re-verify it.

3.8 Advanced Topics of System Verilog:

Some of the important topics are not being covered in the FIFO example. Below we have used another example of Router to cover all the major aspects of System Verilog concepts.

Scoreboard: Scoreboard is used to store the expected output of the device under test. It implements the same functionality as DUT. It uses higher level of constructs. Dynamic data types and dynamic memory allocations in SystemVerilog make us easy to write scoreboards.

Scoreboard is used to keep track of how many transactions were initiated; out of which how many are passed or failed.

Access the router example from the directory copied from Hafez server.

> cd SV_router_example/

SV_router_example directory consists of folders named

1. labs
2. rtl
3. solutions

I recommend you to check out the different lab solutions available in the *solutions* directory. As we proceed with labs from 1 to 6 in *solutions* folder, more system verilog concepts are being dealt with. Refer the textbooks mentioned at the end of this tutorial to get deeper knowledge about these concepts. At the end of lab 6, we get a completed router testbench in System Verilog.

Note: Students who wish to give a try answering the labs; there is a hands-on ***labs*** directory, to try your own solutions for the router design.

System verilog functional coverage Introduction:

Coverage is defined as the percentage of verification objectives that have been met. It is used as a metric for evaluating the progress of a verification project in order to reduce the number of simulation cycles spent in verifying a design. There are two type of the coverage metrics code coverage and functional Code coverage and function coverage tells the verification engineer if the test plan goals have been met or not.

The SystemVerilog functional coverage constructs enable the following:

1. Coverage of variables and expressions, as well as cross coverage.
2. Automatic as well as user-defined coverage bins.

3. Filtering conditions at multiple levels.
4. Events and sequences to automatically trigger coverage samples.
5. Procedural activation and query of coverage
6. Optional directives to control and regulate coverage

Scoreboard and Functional Coverage:

The main goal of a verification environment is to reach 100% coverage of the defined functional coverage spec in the verification plan. Based on functional coverage analysis, the random based tests are than constrained to focus on corner cases to get do complete functional check. Coverage is a generic term for measuring progress to complete design verification. Simulations slowly paint the canvas of the design, as we try to cover all of the legal combinations. The coverage tools gather information during a simulation and then post process it to produce a coverage report. You can use this report to look for coverage holes and then modify existing tests or create new ones to fill the holes.

Covergroup: Covergroup is like a user-defined type that encapsulates and specifies the coverage. It can be defined in a package, module, program, interface or class once defined multiple instances can be created using new Parameters to new () enable customization of different instances. In all cases, we must explicitly instantiate it to start sampling. If the cover group is defined in a class, you do not make a separate name when we instance it. Cover group comprises of cover points, options, formal arguments, and an optional trigger. A cover group encompasses one or more data points, all of which are sampled at the same time.

Syntax Eg:

```
covergroup cg;
// Coverage points
endgroup: cg
//create instance
cg cg_inst = new;
```

Eg 1: *How to develop functional coverage block*

1. Create a covergroup inside classes
2. Covergroup encapsulates coverage bins definitions (state, transition, cross correlation)
3. Coverage bins sample timing definition
4. Coverage attributes

```
covergroup fcov (ref bit [3:0] sa, da) @ (condition)
   coverpoint sa;
   coverpoint da;
endgroup: fcov
bit [3:0] sa, da;
```

```
real coverage = 0.0;
fcov port_fc = new ( sa, da);

initial while (coverage < 99.9) begin
    ....
    sa = pkt_ref.sa;
    da = pkt_ref.da;
    ...
coverage = $get_coverage();
coverage = port_fc.get_inst_coverage ();
end
```

The covergroup can be embedded in a class and program block too. The advantage of creating the covergroup inside the class definition is that the covergroup automatically has access to all properties of the class without having an I/O argument.

VCS can automatically create across coverage bins

```
Eg: covergroup cov1() @ (router.cb);
         coverpoint sa;
         coverpoint da;
         cross sa, da;
     endgroup: cov()
```

The two major parts of functional coverage are the sampled data values and the time when they are sampled. When new values are ready (such as when a transaction has completed), your testbench triggers the cover group. This can be done directly with the sample function, as shown in Sample 9.5, or by using a blocking expression in the covergroup definition. The blocking expression can use a wait or @ to block on signals or events. Use sample if you want to explicitly trigger coverage from procedural code, if there is no existing signal or event that tells when to sample, or if there are multiple instances of a cover group that trigger separately. Use the blocking statement in the covergroup declaration if you want to tap into existing events or signals to trigger coverage.

To calculate the coverage for a point, you first have to determine the total number of possible values, also known as the domain. There may be one value per bin or multiple values. Coverage is the number of sampled values divided by the number of bins in the domain. A cover point that is a 3-bit variable has the domain 0:7 and is normally divided into eight bins. If, during simulation, values belonging to seven bins are sampled, the report will show 7/8 or 87.5% coverage for this point. All these points are combined to show the coverage for the entire group, and then all the groups are combined to give a coverage percentage for all the simulation databases.

Note: For various techniques in functional coverage using callbacks and triggers refer to the textbook available on Hafez server.

Example: Router

Description of the router design:

1. The router has 16 input ports and 16 output ports. Each input has and output has 3 signals, serial data, frame and valid. These signals are represented in a bit-vector format, din[15:0], frame_n[15:0], valid_n[15:0], dout[15:0], valido_n[15:0] and frameo_n[15:0].
2. To drive individual port, the specific bit position corresponding to the port number must be specified. For example, input port 3 is to be driven, and then the corresponding signals shall be din[3], frame_n[3] and valid_n[3].
3. To sample an individual port, the specific bit position corresponding to the port number shall be specified. For example, if output port 7 is to be sampled, then the corresponding signals shall be dout[7], frameo_n[7] and valido_n[7].
4. Packets are sent in variable length, composed of header and payload.
5. Packets can be routed from any input port to any output port.

4. References & Books

[1] "*System Verilog for Verification, A Guide to Learning the Testbench Language Features*", by Chris Spears, Second Edition.

[2] www.testbench.in

[3] *"System Verilog for Design, A Guide to Using System Verilog for Hardware Design and Modeling"*, by Stuart Sutherland, Simon and Peter Flake.

[4] http://www.systemverilog.in/systemverilog.php

[5] http://www.design-reuse.com/articles/22264/system-verilog-ovm-verification-reusability.html

[6] http://events.dvcon.org/2011/proceedings/papers/01_3.pdf