Lab 1
Memory Verification using Tasks & Functions

Module ID : CX-301

Design Verification

Instructor : Dr. Abid Rafique

Version 1.2

Document History

The changes and versions of the document are outlined below:

Version	State / Changes	Date	Author
1.0	Initial Draft	Jan, 2024	Qamar Moavia
1.1	Modified with new exercises	feb, 2025	Qamar Moavia

Table of Contents

Objectives

By the end of this lab, students will be able to answer the following questions:

- What is the purpose of testbench?
- What are SystemVerilog Tasks and Functions? How are they helpful in Verification?
- How SystemVerilog Loops, Tasks and Functions helps in writing different test cases faster?

Tools

- SystemVerilog
- Synopsys VCS

Instructions for Lab Tasks

The required files for this lab can be found on the

```
share_folder/CX-300-DesignVerification/Labs/Lab1
```

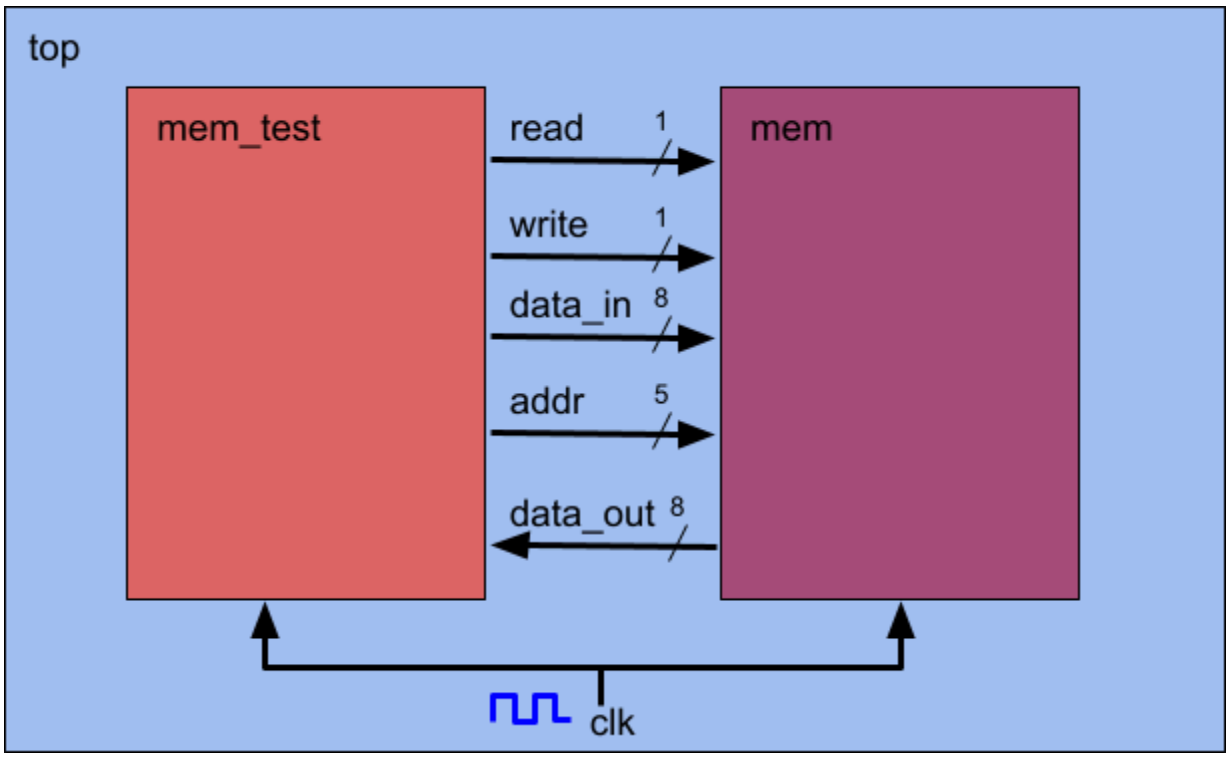
The submission must follow the hierarchy below, with the folder named and the file names exactly as listed below.

```
./lab1/
      ├── Task1/
      │    ├── design/
      │    │    ├── mem.sv
      │    ├── testbench/
      │    │    ├── mem_test.sv
      │    │    ├── top.sv
      │    ├── // screenshots of outputs/waveforms are must
      ├── Task2-4/
      │    ├── design/
      │    │    ├── mem.sv
      │    ├── testbench/
      │    │    ├── mem_test.sv
      │    │    ├── top.sv
      │    ├── // screenshots of outputs/waveforms are must
```

Along with that you also need to upload your solution on the github as well, and share the link

Testbench Architecture

Previously, we instantiated the **DUT** directly inside the testbench. In this lab, we are using a **top-level module** that contains both the **mem_test** and the **memory DUT**, which simplifies the testbench. This approach improves modularity, making the **top module** more straightforward and easier to manage, while also enhancing scalability and reusability for future projects.



DUT Specifications:

Throughout the SystemVerilog labs, we will be working with a Memory Design as our DUT (Design Under Test). A key responsibility of a verification engineer is to thoroughly understand the design before verifying it. To achieve this, design specifications are always shared with the verification team first. This ensures a clear understanding of the DUT, enabling effective verification.

Below are the design specifications for the memory design that will be used across the SystemVerilog labs.

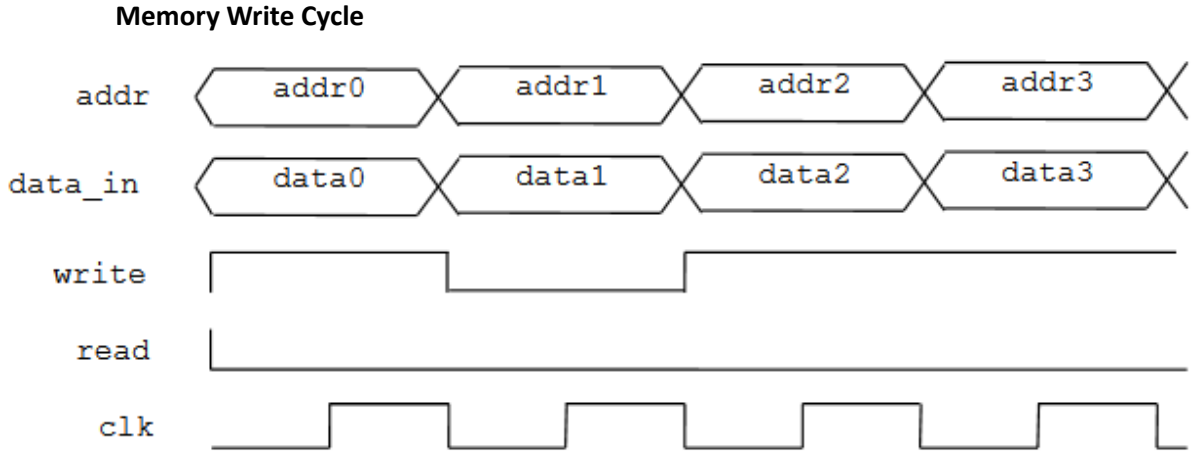
Input/Outputs:

- **`addr`** is a 5-bit logic vector.
- **`data_in`** and **`data_out`** are both 8-bit logic vectors.
- **`read`**, **`write`**, and **`clk`** are single-bit logic signals.

Operations:

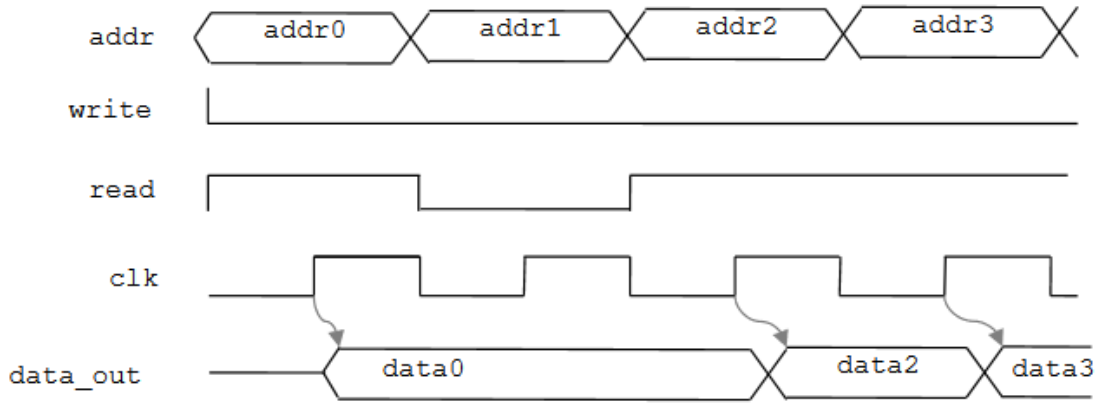
- **Memory write:**

  On the positive edge of **`clk`**, if **`write = 1`**, the value of **`data_in`** is stored in **`memory[addr]`**.



- **Memory read:**

  On the positive edge of **`clk`**, if **`read = 1`**, the value stored at **`memory[addr]`** is assigned to **`data_out`**.



Constraints:

The input signals **`read`** and **`write`** should never be simultaneously high.

TASK 1: Write Testbench to Verify the Memory

Write a SystemVerilog Testbench to verify the functionality of the memory. In that make sure you verify the memory with the following two test cases that are described below:

1. Clearing the whole memory by writing zero to every address location, and then reading back and checking the data read from each location of memory should be zero.
2. Writing the data equal to the address to every address location (data 0 to address 0, data 1 to address 1 and so on), and then reading back and checking the data read matches the data written.

Using Tasks and Functions in Memory Verification:

In advanced verification, covering a large number of test cases at the signal level is impractical and time-consuming. Instead, verification engineers shift to **transaction-level modeling (TLM)**, focusing on high-level operations (e.g., read, write) rather than individual signals. This abstraction allows for the rapid creation and easy reuse of test cases. Let's explore a simple example to better understand this:

Signal-Level Approach:

To write data = 10 at address = 9, one might do something like this:

```
@(negedge clk) write <= 1; read <= 0; data_in <= 10; address <= 9;
```

Later, if we want to write data = 7 to address = 25, we would need to repeat a similar line:

```
@(negedge clk) write <= 1; read <= 0; data_in <= 7; address <= 25;
```

While this might work for simple designs, such as the memory you're verifying in this lab, it becomes impractical for more complex designs like UART, AXI Interconnect, Processor Core, or DMA.

For this reason, verification engineers avoid working directly at the signal level. Instead, they first set up the testbench by creating reusable **tasks** and **functions** (such as `read_mem`, `write_mem`, `print_status`, `checker`, etc.), which can later be used to write test cases and check the output of the design.

By using tasks and functions, engineers can automate common operations like memory reads and writes, making the testbench more efficient and scalable. This approach not only reduces verification time but also increases maintainability and readability.

In the next few lab tasks we will write SystemVerilog Tasks & Functions to help us in verifying the memory.

TASK 2 : Writing Tasks for Memory Read and Write

Create stimulus tasks using SystemVerilog subprogram enhancements and verify the supplied memory design.
Working in the **`Task1`** directory, perform the following.

3. For this lab, the memory design is already written (**`mem.sv`**). You are going to complete the memory testbench in the **`mem_test.sv`** file.

   You will be using the following SystemVerilog subprogram constructs:
   - **`void`** function
   - Argument passing by name (**`.name(name)`**)
   - Default formal arguments with default values
4. Declare a task (**`write_mem`**) to write to the memory as follows:
   a. Define input arguments for the address and data values.
   b. Assign **`addr`**, and **`data_in`** from the arguments, and drive **`read`** and **`write`** synchronized to the clock. Hint: drive signals on the inactive clock edge to avoid race conditions.

c. Define an input argument debug with a default value of 0. If **`debug`** = 1, display the write address and data values.

5. Declare a task (**`read_mem`**) to read the memory as follows:
   a. Define an input argument for the address value and an output argument for the read data. Remember to use blocking assignment for assigning read data to output argument, so that assignment is complete upon return to the caller.
   b. Assign **`addr`** from the argument, and drive **`read`** and **`write`**, synchronized to the clock. **Hint:** drive signals on the inactive clock edge to avoid race conditions.
   c. Assign the output data argument from **`data_out`** at an appropriate time. There is a short propagation delay between the rising edge of **`clk`** and **`data_out`** updating.
   d. Input argument **`debug`** with a default value of 0. If **`debug`** = 1, display the read address and data values.

TASK 3 : Completing the Test Cases & Running the Simulation

6. Complete the **"Clearing the Memory"** test by writing zero to every address location, and then reading back and checking the data read matches the data written.
7. Complete the **"Data = Address**" test by writing data equal to the address to every address location, and then reading back and checking the data read matches the data written.
8. Write a **void** function (**printstatus**) with an input argument **status** which indicates the number of errors encountered in the tests. If **status** = 0, the test passed.
9. Use **VCS** to compile and run the simulation. Remember to also compile the **top.sv** module, which instantiates and connects both design and testbench.
10. Debug your subprograms as required until you are happy the design is verified.

TASK 4 : Adding More Test Cases

11. Add another test case **“Data = Random**” test which writes random data to an address then read back from the same address to verify its correctness. Then it moves on to the next address. Repeat this for all addresses.

Good Luck 🙂