BASIC MEMORY PROJECT

## **1. Introduction**

This document provides an overview of the Memory Module project, which implements a synchronous memory system capable of basic read and write transactions with handshaking. The project involves two main components:

1. **Memory Design**: A memory module that supports data writing and reading with handshaking signals (valid, ready).
2. **Testbench**: A testbench to simulate the behavior of the memory module under various scenarios, ensuring correctness through different types of accesses.

### **1.1 Objective**

The goal of this project is to design and verify a memory module using handshaking signals for proper synchronization of read and write transactions. The testbench ensures all possible scenarios are tested to verify the functionality of the memory.

## **2. Memory Design Overview**

## **2.1. Functionality**

The **Memory module** performs basic read and write operations using the following functionality:

* **Write Operation**: When wr\_rd\_i is 1 (indicating a write), data (write\_i) is written to the memory location specified by addr\_i.
* **Read Operation**: When wr\_rd\_i is 0 (indicating a read), data is read from the memory location specified by addr\_i and placed on read\_o.
* **Handshaking**: The module uses a handshaking mechanism to synchronize the transaction between the memory and the testbench:
  + **valid\_i**: Asserted by the testbench to signal a valid transaction.
  + **ready\_o**: Asserted by the memory to indicate that it is ready to either read or write.

### **2.2. Key Features**

* Synchronous operation with a clock signal.
* Memory size and data width are configurable via parameters D (depth) and W (width).
* Reset functionality: Clears the memory and resets output signals.
* Error-free read and write operations: Uses handshaking to ensure proper data transfer.

### **2.3. Pin Description**

* **Inputs**:
  + clk\_i: Clock signal.
  + rst\_i: Reset signal.
  + addr\_i: Address to access in memory.
  + write\_i: Data to write into memory.
  + wr\_rd\_i: Read/write control signal (1 for write, 0 for read).
  + valid\_i: Handshaking signal indicating valid transaction.
* **Outputs**:
  + read\_o: Data read from memory.
  + ready\_o: Handshaking signal indicating memory readiness.

### **2.4. Memory Array**

The memory is an array of registers (mem[D-1:0]), where each register holds W bits of data. The array's size is configurable based on the parameter D (depth).

## **3. Testbench Overview**

### **3.1. Testbench Structure**

The **testbench** is designed to simulate various memory transactions and ensure that the memory module behaves as expected under different conditions. It covers the following scenarios:

* Front-door access (FD) to the memory with write and read operations.
* Back-door access (BD) using $readmemh and $writememh to directly write and read memory content.
* Testing specific quarters of the memory.
* Randomized read and write tests.
* Concurrent and consecutive write and read operations.

### **3.2. Key Features**

* **Clock Generation**: A clock (clk\_i) is generated with a period of 10 time units.
* **Reset Mechanism**: The reset is activated at the start and deactivated after two clock cycles.
* **Parameterized Test Cases**: The testbench uses $value$plusargs to choose between different test scenarios.
* **Handshaking in Test Cases**: Each test case uses the valid\_i and ready\_o signals to control the flow of transactions and ensure that the memory module handles data correctly.

### **3.3. Test Case Breakdown**

1. fd\_write and fd\_read: Write and read operations on the memory using front-door access.
2. bd\_write and bd\_read: Write and read operations using back-door access to load and store data.
3. Consecutive and Concurrent Operations: Test consecutive and parallel read/write operations to ensure memory can handle simultaneous accesses correctly.

## **4. Key Considerations for Design and Testing**

### **4.1. Handshaking Logic**

* The valid\_i signal is used to initiate transactions and indicate that data is valid. The memory responds to this signal with the ready\_o signal.
* The testbench waits for ready\_o to go high before proceeding to the next transaction, ensuring proper synchronization between memory operations.

### **4.2. Reset Logic**

* On reset (rst\_i), all memory locations are cleared, and the output signals are reset to their initial values.

### **4.3. Edge Cases and Error Handling**

* The design and testbench should ensure that edge cases like reading from an empty location, writing to a full location, and memory overflow are handled properly.

### **4.4. Functional Coverage**

* Ensure that all address locations are tested.
* Test Quaaddress accesses.
* Verify write and read operations at different locations in memory.
* Consecutive and Concurrent Execution.

## **5. Simulation and Debugging**

For this project, we use ModelSim software to simulate the design and verify the correctness of the memory module. ModelSim provides an efficient environment for simulating Verilog/SystemVerilog code, generating waveforms, and performing debugging.

The simulation process involves running the 13 test case scenarios described in the testbench, verifying that the memory design behaves as expected in each scenario. Each test case is designed to test specific functionalities and edge cases, ensuring comprehensive coverage of the memory module.

#### **5.1. Test Case Scenarios in ModelSim**

We will simulate and verify the following test case scenarios using ModelSim:

1. Test Case 1: FD Write and FD Read (Single Location)
   1. Write to and read from a single memory location.
   2. Verification: Check that the correct data is written and read at the specified address.
2. Test Case 2: FD Write and FD Read (All Locations)
   1. Write to and read from all memory locations.
   2. Verification: Ensure all memory locations are correctly written and read.
3. Test Case 3: Access 1st Quarter FD
   1. Perform write and read operations for the first quarter of memory.
   2. Verification: Confirm correct operations within the first quarter of memory.
4. Test Case 4: Access 2nd Quarter FD
   1. Perform write and read operations for the second quarter of memory.
   2. Verification: Verify the memory operations in the second quarter.
5. Test Case 5: Access 3rd Quarter FD
   1. Perform write and read operations for the third quarter of memory.
   2. Verification: Ensure memory operations are correct in the third quarter.
6. Test Case 6: Access 4th Quarter FD
   1. Perform write and read operations for the fourth quarter of memory.
   2. Verification: Confirm operations in the final quarter of memory.
7. Test Case 7: FD Write and BD Read
   1. Write data via front-door access and read using back-door access.
   2. Verification: Ensure front-door write and back-door read are correctly synchronized.
8. Test Case 8: BD Write and FD Read
   1. Write data via back-door access and read using front-door access.
   2. Verification: Validate back-door write and front-door read functionality.
9. Test Case 9: FD Random Location Write and Read
   1. Perform write and read at random memory locations.
   2. Verification: Ensure that data is correctly written and read at random locations.
10. Test Case 10: Consecutive Write and Read
    1. Perform consecutive write and read operations on sequential memory locations.
    2. Verification: Check that consecutive locations are written and read correctly.
11. Test Case 11: Concurrent Write and Read
    1. Perform write and read operations concurrently on the same memory location.
    2. Verification: Confirm that concurrent operations are handled correctly without conflicts.
12. Test Case 12: BD Write and BD Read
    1. Perform write and read operations using back-door access.
    2. Verification: Validate that both write and read operations are correctly executed using back-door access.
13. Test Case 13: Read All Locations
    1. Read all memory locations to ensure data integrity.
    2. Verification: Check that all locations are correctly read without errors.

#### **5.2. Simulation Setup**

1. **Compiling the Design and Testbench**:
   1. In ModelSim, first, compile the design and testbench files to check for syntax or compilation errors.
   2. Use the command:

shell

Copy code

vcom memory\_nonb.v tb\_final.v

1. **Running the Simulation**:
   1. After compiling, run the simulation with the vsim command:

shell

Copy code

vsim work.tb

* 1. You can specify the test case to run by providing the correct value for $value$plusargs("value").

1. **Monitoring and Debugging**:
   1. Open **waveform viewer** in ModelSim to view signal transitions and check if the expected behavior matches the actual simulation results.
   2. Look for issues such as:
      1. Incorrect data written or read.
      2. Handshaking signals not operating as expected.
      3. Timing mismatches between write and read operations.

### **5.3. Debugging and Analysis**

After running the simulation for the 13 test cases, analyze the results:

1. **Check Waveforms**: Ensure the signal transitions (e.g., valid\_i, ready\_o, write\_i, read\_o) match the expected behavior defined in the test cases.
2. **Identify Errors**: If any issues occur, look at the logs or waveforms to locate the source of the error.
   1. For example, if a write operation fails, check if the memory location was accessed correctly.
   2. If the ready\_o signal doesn't assert when expected, verify the handshaking logic in the design.
3. **Correct Errors**: Based on the analysis, update the design or testbench to fix errors or improve the test coverage.

## **6. Conclusion**

The simulation in ModelSim allows for an in-depth verification of the memory design under various scenarios, ensuring correct functionality across different test cases. The testbench helps to automate the verification process, and ModelSim’s waveform viewer and debugging tools provide a robust environment to identify and correct any issues in the design.

This project successfully implements a synchronous memory module with handshaking for read and write operations. The provided testbench comprehensively covers various scenarios, ensuring the memory design works as intended. This design can be extended to handle larger memory sizes, more complex operations, and additional verification features as required.