iTest Bangladesh

Linked List DUT Test Plan Report

Prepared By:

Test Plan: Zavin
Verification and Report Written By: A.S.M. Sarwar Jahan

Contents

Introduction

The Linked List DUT Test Plan is designed to verify the functionality and correctness of a dynamically implemented linked list in SystemVerilog. This report outlines the approach, methodology, and results of testing the Linked List DUT to ensure its ability to handle key operations such as insertion, deletion, traversal, reversal, and length computation.

The goal of this testing process is to validate that the implementation adheres to the expected behavior under various scenarios, including both typical and edge cases. The testing is conducted using a combination of random testing (to explore unexpected scenarios) and directed testing (to focus on specific operations and edge cases).

This test plan employs a stimulus-driven verification environment that generates input sequences and monitors the DUT's output. A comparison between the expected and actual outputs ensures accurate functionality. The results of these tests are analyzed to confirm that the linked list meets the requirements and performs reliably under all conditions.

By identifying any potential issues and validating the correctness of the implementation, this report aims to ensure the robustness and usability of the Linked List DUT for real-world applications.

Design Overview

The Linked List DUT is implemented in SystemVerilog to model a dynamic data structure capable of performing fundamental operations such as insertion, deletion, traversal, reversal, and length computation. The design comprises the following key components:

1. Node Class

- Represents a single element in the linked list.
- Each node contains:
    - Data: A bit[0:7] value that stores the information for the node.
    - _next: A pointer to the next node in the linked list.
- Constructor: Initializes the node with a given data value and an optional reference to the next node.

2. Linked List Class

- Implements the core logic and operations of the linked list.
- Attributes:
    - head: A pointer to the first node of the list, representing the start of the linked list.
- Functions:
    - new(list_q data): Constructs a linked list from a queue of input data.

- traverse(): Returns all elements of the list in order.
- find(data): Searches for a specific data value in the list and returns its position(s).
- insert(prev_data, next_data): Inserts a new node with next_data after the first occurrence of prev_data.
- delete(data): Removes the first occurrence of a node with the specified data.
- len(): Computes and returns the number of nodes in the list.
- reverse(): Reverses the order of the nodes in the list.

3. DUT (Device Under Test) Class

- Encapsulates the linked_list class and serves as the testable unit.
- Functions:
  - run(opcode_u op,
  - list_q data): Executes the requested operation (NEW, INSERT, DELETE, etc.) based on the operation code (opcode_u) and the input data queue.

Test Plan and Methodology

General Overview

The verification of the **Linked List DUT** relies on systematically comparing the outputs from the DUT and the Monitor class. Inputs are generated using the **Stimulus** class and fed into both the DUT and Monitor class. The DUT processes these inputs using the implemented linked list operations, while the Monitor calculates the expected results based on basic queue operations in SystemVerilog. By comparing the DUT's output with the expected output from the Monitor, we can determine the correctness of each operation. A match between the outputs indicates a test pass, while a mismatch signifies a test failure. This process allows us to verify whether all operations of the linked list are functioning as intended.

Types of Testing

1. Constrained Random Testing

Definition:
Constrained random testing involves generating random inputs within defined constraints to test various scenarios in a systematic yet unpredictable manner. It combines the flexibility of randomness with the control of constraints, allowing targeted yet diverse testing. For random test, we used access+r -sv_seed command {random} command in run window.

Methodology:

- In the **Stimulus** class, different input sequences are generated:
    - **Alternating Sequences**: A sequence alternating between two values (e.g., {4, 5, 4, 5}).
    - **Same Sequences**: A sequence with the same value repeated (e.g., {4, 4, 4}).
    - **One, Two, and Three Element Sequences**: Short sequences with 1, 2, or 3 elements.
- Randomization is applied at multiple levels:
    - Selection of the input sequence is randomized.
    - Selection of the opcodes (NEW, INSERT, DELETE, etc.) is randomized.
    - For operations like INSERT and FIND, the target value can either:
        - Exist within the current queue.
        - Be a completely random value outside the queue.
- The random sequences and operations are fed into both the DUT and Monitor, ensuring a comprehensive exploration of possible scenarios.

Advantages:

- **Increased Coverage**: Tests diverse scenarios, including unexpected edge cases.
- **Efficiency**: Automates the testing process and reduces manual effort.
- **Scalability**: Easily extendable to include additional constraints or operations.

2. Directed Testing

Definition:
Directed testing involves crafting specific, deterministic test cases for well-defined inputs and expected outputs. This method ensures thorough testing of known scenarios, edge cases, and boundary conditions.

Methodology:

- A total of **46 directed test cases** were executed.
- Tests covered each linked list operation:
    - **NEW**: Tests for creating lists of varying lengths and contents.
    - **TRAVERSE**: Verifies that the linked list elements are retrieved in order.
    - **INSERT**: Validates insertion of elements at specific positions.
    - **DELETE**: Ensures proper deletion of the first occurrence of a given element.

    - **FIND**: Checks for the presence and index of an element in the list.
    - **REVERSE**: Tests the reversal of the linked list's order.
    - **LEN**: Confirms the count of nodes in the list.
- Edge and boundary cases were explicitly tested:
    - Operations on an empty list.
    - Duplicate elements in the list.
    - Insertions at the beginning or end of the list.
    - Deletions of non-existent elements.

Advantages:

- **Targeted Validation**: Ensures specific scenarios are covered thoroughly.
- **Reproducibility**: Provides repeatable test cases with predictable outcomes.
- **Edge Case Testing**: Validates behavior under unusual or boundary conditions.

Test Cases and Observations:

Analysis of Random Test Results:

1. Overall Performance:
   Most of the test cases passed successfully during random testing. The loop for constrained random tests was executed 20 times, and the results provide significant insights.

2. Identified Issues:

    - Length Calculation Error (Error 1):
      The DUT's length calculation consistently reports one more element than the actual count. Consequently, all length-related operations fail.
      Example:
      Input Queue: {4, 5, 4, 5, 4, 5, 4}
      Expected Length: {7}
      DUT Output Length: {8}

    - Deletion Error for Repeated Values (Error 2):
      When attempting to delete a specific value that occurs multiple times in the queue, the DUT erroneously deletes all instances of that value rather than just the first occurrence.
      Example:
      Input Queue: {4, 5, 4, 5, 4, 5, 4}
      Value to Delete: 5
      Expected Queue After Deletion: {4, 4, 5, 4, 5, 4}
      DUT Output: {4, 4, 4, 4}

Test Case No.	Operation	Input Queue/Data	Expected Output	Actual Output	Result	Remarks
1	NEW	{3}	{3}	{3}	PASSED	Successfully initialized queue.
2	REVERSE	{3}	{3}	{3}	PASSED	Reverse operation correctly handled for a single element.
3	NEW	{4, 5, 4, 5, 4, 5, 4}	{4, 5, 4, 5, 4, 5, 4}	{4, 5, 4, 5, 4, 5, 4}	PASSED	Successfully initialized queue.
4	LEN	{4, 5, 4, 5, 4, 5, 4}	{7}	{8}	FAILED	Length calculation incorrect; LEN logic needs to be fixed.
5	NEW	{3, 5, 4, 5, 4, 5, 4}	{3, 5, 4, 5, 4, 5, 4}	{3, 5, 4, 5, 4, 5, 4}	PASSED	Successfully initialized queue.
6	LEN	{3, 5, 4, 5, 4, 5, 4}	{7}	{8}	FAILED	Length calculation incorrect; LEN logic needs to be fixed.
7	NEW	{30, 26, 24, 38, 4, 5, 4}	{30, 26, 24, 38, 4, 5, 4}	{30, 26, 24, 38, 4, 5, 4}	PASSED	Successfully initialized queue.
8	REVERSE	{30, 26, 24, 38, 4, 5, 4}	{4, 5, 4, 38, 24, 26, 30}	{4, 5, 4, 38, 24, 26, 30}	PASSED	Reverse operation successfully handled.
9	NEW	{4, 6, 2, 38, 4, 5, 4}	{4, 6, 2, 38, 4, 5, 4}	{4, 6, 2, 38, 4, 5, 4}	PASSED	Successfully initialized queue.
10	INSERT	{4, 6, 2, 38, 4, 5, 4}, 6, 9	{4, 6, 9, 2, 38, 4, 5, 4}	{4, 6, 9, 2, 38, 4, 5, 4}	PASSED	Insert operation successfully handled.
11	NEW	{4, 4, 4, 4, 4, 5, 4}	{4, 4, 4, 4, 4, 5, 4}	{4, 4, 4, 4, 4, 5, 4}	PASSED	Successfully initialized queue.

12	LEN	{4, 4, 4, 4, 4, 5, 4}	{7}	{8}	FAILED	Length calculation incorrect; LEN logic needs to be fixed.
13	DELETE	{4, 5, 4, 5, 4, 5, 4}, 5	{4, 4, 5, 4, 5, 4}	{4, 4, 4, 4}	FAILED	Deletion logic incorrect for multiple occurrences of a value.
14	NEW	{4, 4, 4, 4, 4, 4, 4, 4, 4}	{4, 4, 4, 4, 4, 4, 4, 4, 4}	{4, 4, 4, 4, 4, 4, 4, 4, 4}	PASSED	Successfully initialized queue.
15	FIND	{4, 4, 4, 4, 4, 4, 4, 4, 4}, 87	{255}	{255}	PASSED	Find operation correctly handled for a non-existent value.
16	INSERT	{3, 5, 4, 5, 4, 5, 4}, 3, 100	{3, 100, 5, 4, 5, 4, 5, 4}	{3, 100, 5, 4, 5, 4, 5, 4}	PASSED	Insert operation successfully handled.
17	LEN	{3, 6, 2, 5, 4, 5, 4, 4, 4, 5}	{10}	{11}	FAILED	Length calculation incorrect; LEN logic needs to be fixed.
18	FIND	{3, 6, 2, 5, 4, 5, 4, 4, 4, 5}, 57	{255}	{255}	PASSED	Find operation correctly handled for a non-existent value.
19	TRAVERSE	{3, 75, 46, 58, 87, 95, 79, 45, 4, 5}	{3, 75, 46, 58, 87, 95, 79, 45, 4, 5}	{3, 75, 46, 58, 87, 95, 79, 45, 4, 5}	PASSED	Successfully traversed the queue.
20	INSERT	{88, 75, 46, 58, 87, 95, 79, 45, 4, 5}, 4, 26	{88, 75, 46, 58, 87, 95, 79, 45, 4, 26, 5}	{88, 75, 46, 58, 87, 95, 79, 45, 4, 26, 5}	PASSED	Insert operation successfully handled.

Analysis of Direct Test Result:

A total of 46 direct tests were executed following Zavin's test plan. While most of the test cases passed successfully, a few bugs were identified. Only the issues are mentioned below to keep the report concise.

Identified Errors:

1. **Error 3**: **Insertion into an Empty List**
   Attempting to insert a value into an empty list caused a fatal error. Insertion into an empty list was not possible.
   Reference Test Case: Test Case 13

2. **Error 4**: **Deletion from an Empty List**
   Deleting an item from an empty list caused an error.
   Reference Test Case: Test Case 25

3. **Error 5**: **Deleting the First Element in a Queue with Duplicate Values**
   When deleting the first element in a queue where all elements are identical, the operation caused an error.
   Reference Test Case: Test Case 24

4. **Error 6**: **Finding an Element Not Present in the List**
   If an element is not found in the list, the expected index is 255.
   Reference Test Case: Test Case 30

5. **Error 7**: **Reversing a Null Queue**
   Reversing an empty (null) queue caused an error.
   Note: This test case was not included in Zavin's test plan but was discovered during additional testing.

Till now, this errors have been found, I will work more to find more bugs!!! 

Conclusion

The verification process of the linked list operations was conducted thoroughly, using both constrained random tests and directed tests based on Zavin's test plan. While most of the test cases passed successfully, several critical issues were identified. These issues highlight areas where the DUT behavior diverges from the expected behavior, particularly in scenarios involving edge cases such as operations on empty lists, handling duplicate elements, and accurate index determination.

The identified issues are vital for improving the robustness and reliability of the linked list implementation. Addressing these bugs will ensure that the linked list operations conform to the specified requirements and perform reliably in all edge cases.

In conclusion, the testing process successfully validated many of the linked list functionalities while also uncovering critical issues, emphasizing the importance of rigorous testing and edge-case analysis in verification methodologies. The next steps should involve fixing the identified errors and re-verifying the DUT to ensure compliance with the expected behavior.

Appendix:

Code link