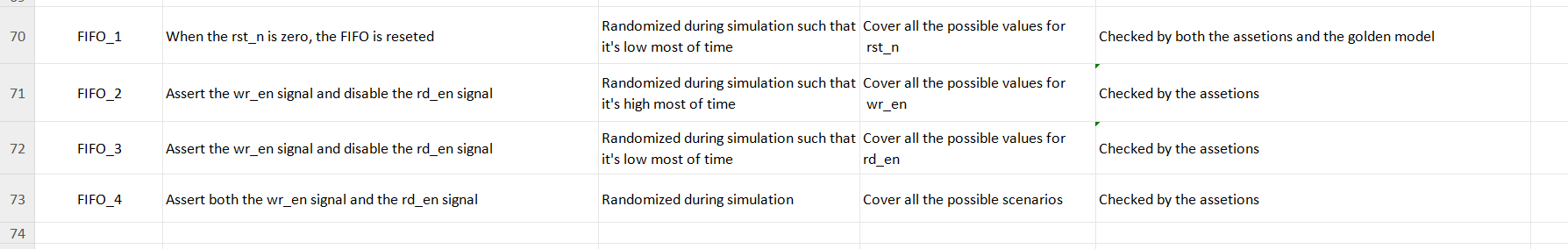


**UVM verification**

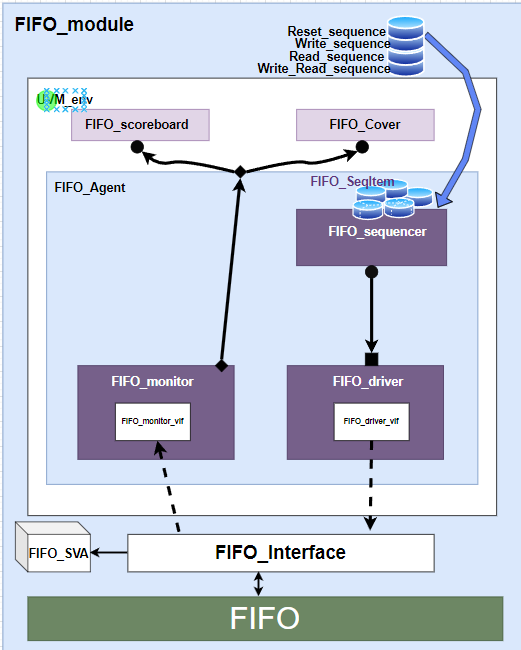
**Name：Ali Mohamed Mohamed Saied**

**Under supervision of : Eng. Kareem Wassem**

**1:Verification plan: (Overview)**

****

**2: UVM structure:**

****

**3:UVM testbench flow sequence:**

**A: Top module (FIFO\_top):**

This is the module which is responsible for generating the system clock signal (clk), instantiate the interface module passing to it the generated clock signal, instantiate the design (DUT) passing to it the just-instantiated interface instance.

**B: Interface (FIFO\_Interface):**

Inside this part, the system input and output signals are created and sent to the “modport” which determines which of this signal will be input and/or output depending on the “modport” type.

**C: Design(FIFO.v):**

It’s the module to be tested. It holds the main operation of the system.

**D: Assertions module(FIFO\_SVA):**

Inside this module we place sequences which we need to test their occurrence such as the assertion of certain signal once another signal is asserted. This module is reponsible for doing this job.

**E: UVM\_test (FIFO\_test):**

It’s the package where the testing strategy is stated. The sequence of the testing process is mentioned there, such that by reading it you can realize which signal to be asserted first and which one second,….etc.

It does that by instantiating the uvm\_sequence(mentioned nextly).

**F: UVM\_env(FIFO\_env):**

Inside this package we include the testing structure. We have various testing elements which are:

1-uvm\_scoreboard. 2-Coverage collector. 3-uvm\_agent.

These structures are classes each of them has a testing rule. Objects of these classes are decalared here in uvm\_env.

**G: UVM\_scoreboard(FIFO\_Scoreboard):**

It’s a class all its role is to check if the system output is correct depending on an available golden reference.

**H: Coverage collector(FIFO\_Cover):**

It’s the class responsible for making sure that all the required bits have experienced toggling at least one time(coverpoint), and also make sure that the required cross-coverage between various signals occurs.

**I: uvm\_agent(FIFO\_Agent):**

It’s a class responsible for including both the stimulus holder for the system (uvm\_driver), and the result holder for the checker classes(uvm\_scoreboard and coverage-collector).

**I: uvm\_sequence\_item(FIFO\_SeqItem):**

It’s the class in which the values for the system input signals are created, some of these signals values may have some depend on some given “constraint” .

**K: uvm\_sequence(FIFO\_Write\_seq, FIFO\_Read\_Seq,…..etc.):**

A class responsible for starting the process of generation of the system input signals, so that it can force some of these signals to certain values directly, while other signals are “randomized”.

**L: uvm\_driver(FIFO\_driver):**

It’s class responsible for holding the generated input signals values from where they are generated (uvm\_sequence\_item) to the design so that it can operate.

**M: uvm\_sequencer(FIFO\_Sequencer):**

This class work as arbiter between the uvm\_sequence and the uvm\_driver.

**N: uvm\_monitor(FIFO\_Monitor):**

A class responsible for holding the system output signals values from the interface to the place where they will be tested if they are correct or not (uvm\_scoreboard and coverage-collector).

**4: Testing results:**

1. **Bug report:**

1. Put brackets around the conditions in any if statement containing more than one condition gathered together to prevent false execution.
2. Adding this code snippet to the always block specified for “count”:

|  |
| --- |
| else if(({wr\_en, rd\_en} == 2'b11)) begin              if(count < FIFO\_DEPTH)                  count <= count + 1;              if(count != 0)                  count <= count - 1;          end |

1. Adding this code snippet to the always block speceified for “rd\_en” and delete the assignment statement of “underflow”:

|  |
| --- |
| else begin          if (empty & rd\_en)              underflow <= 1;          else              underflow <= 0;      end |

1. Modify the assignment statement of “almostempty” to be:

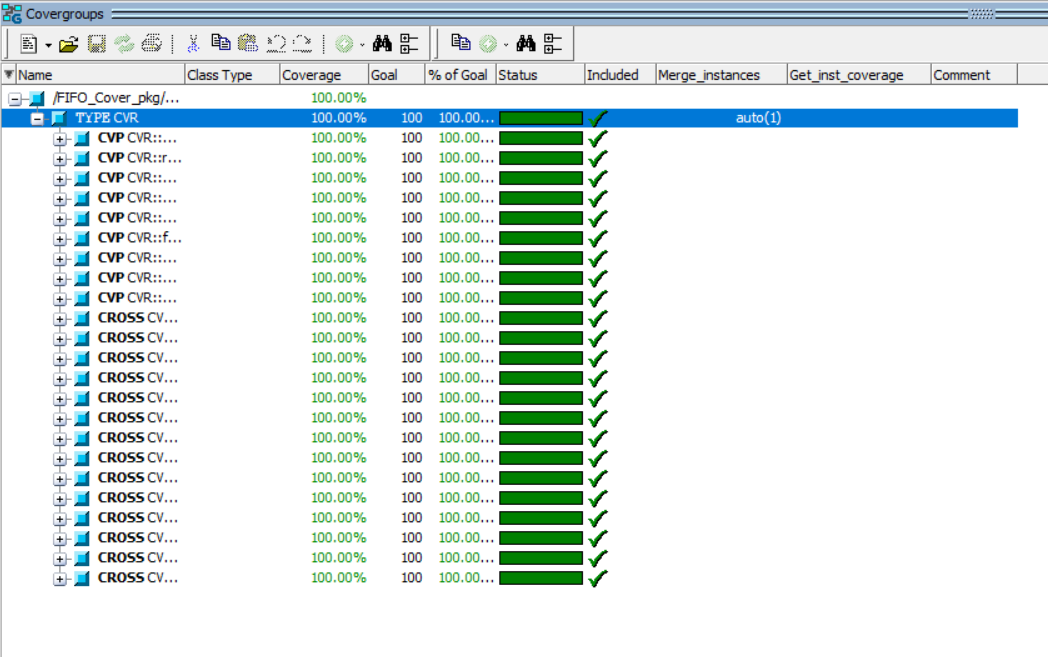
|  |
| --- |
| assign almostfull = (count == FIFO\_DEPTH-1)? 1 : 0; |

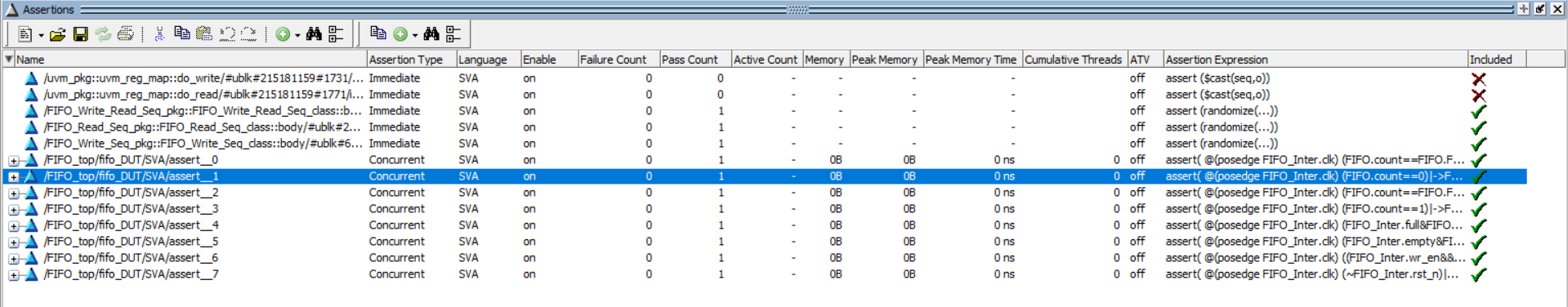
1. Reset signals overflow, wr\_ack and underflow .

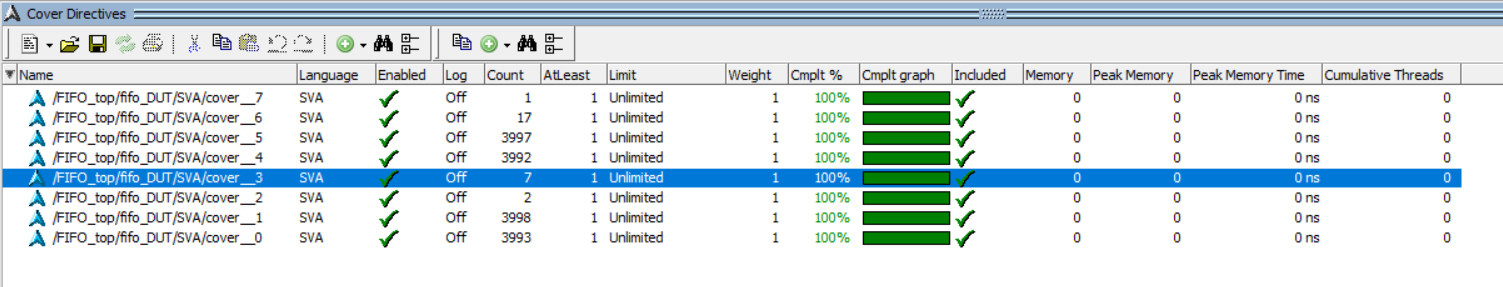
6)The data\_out wasn’t reset-ted when the rst\_n signal is low.

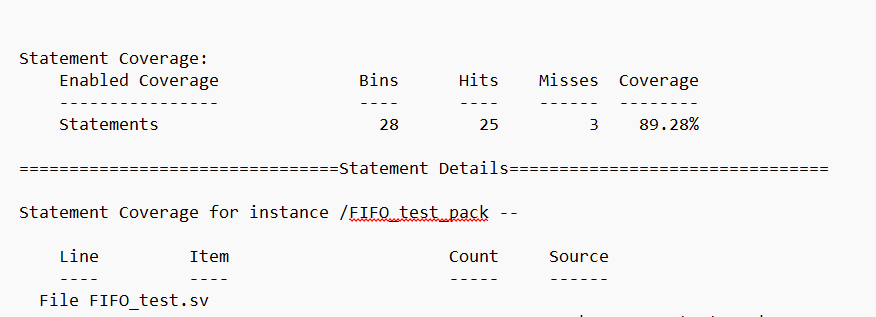
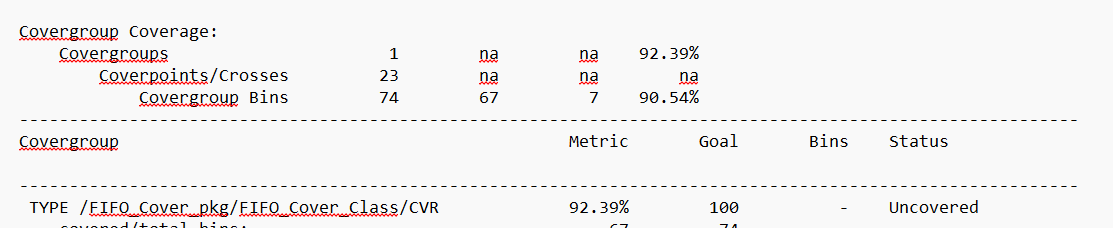
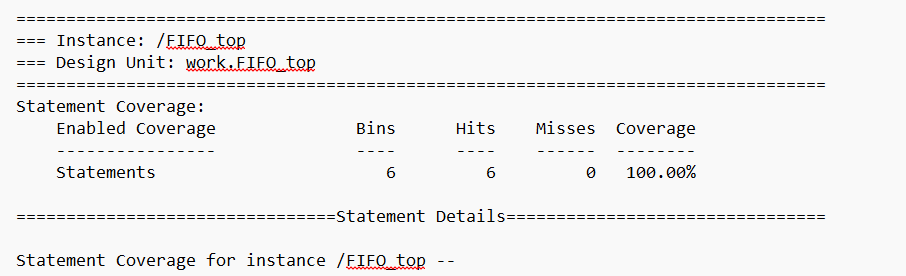
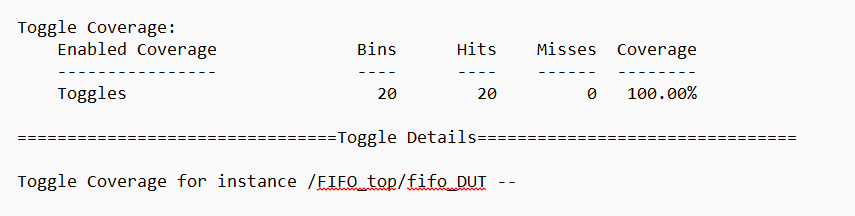
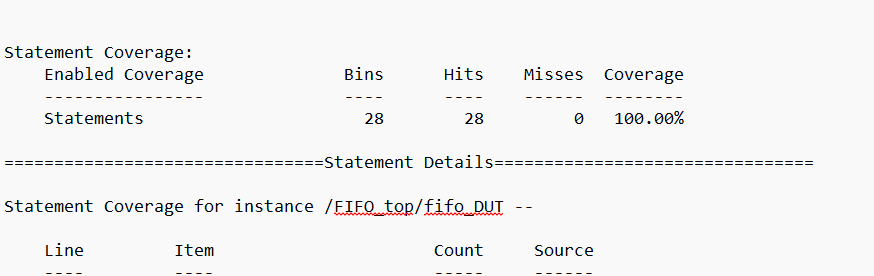
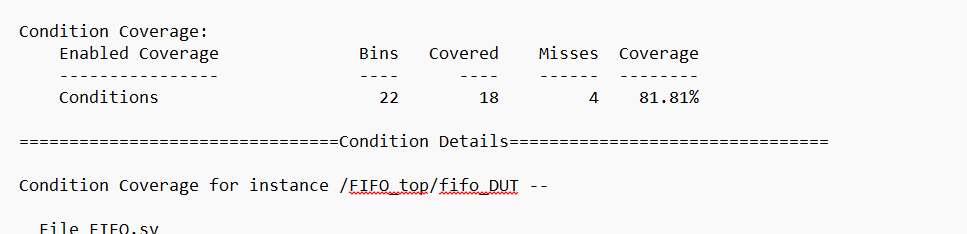
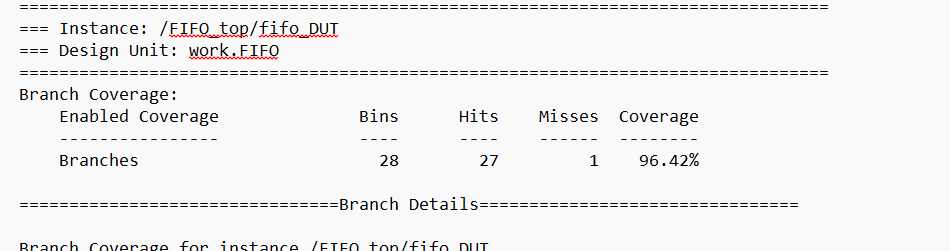
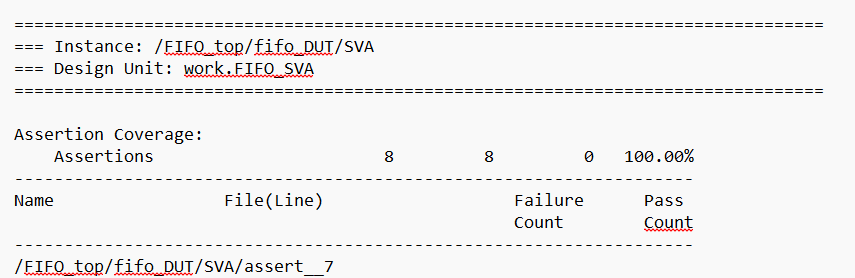
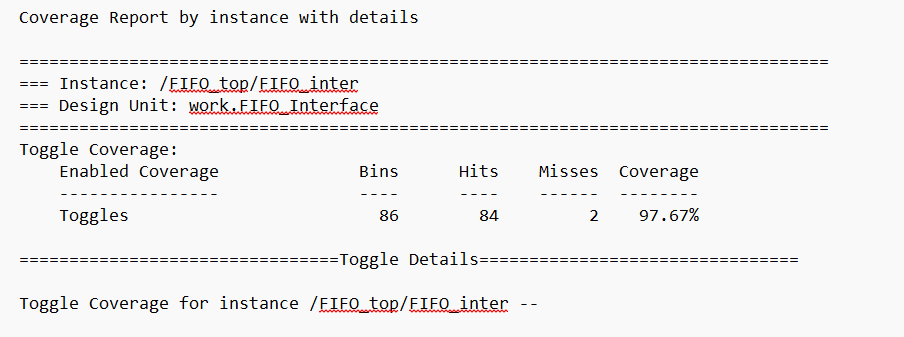
1. **:Code coverage report, Functional coverage report, and Sequential domain coverage:**

****

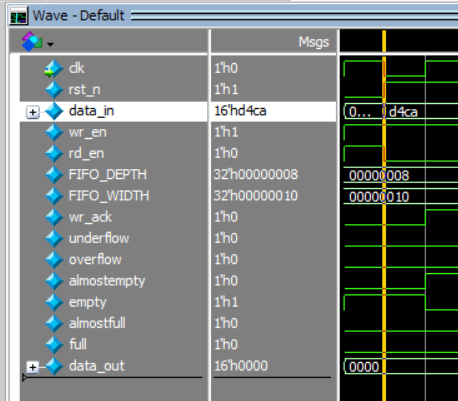
****

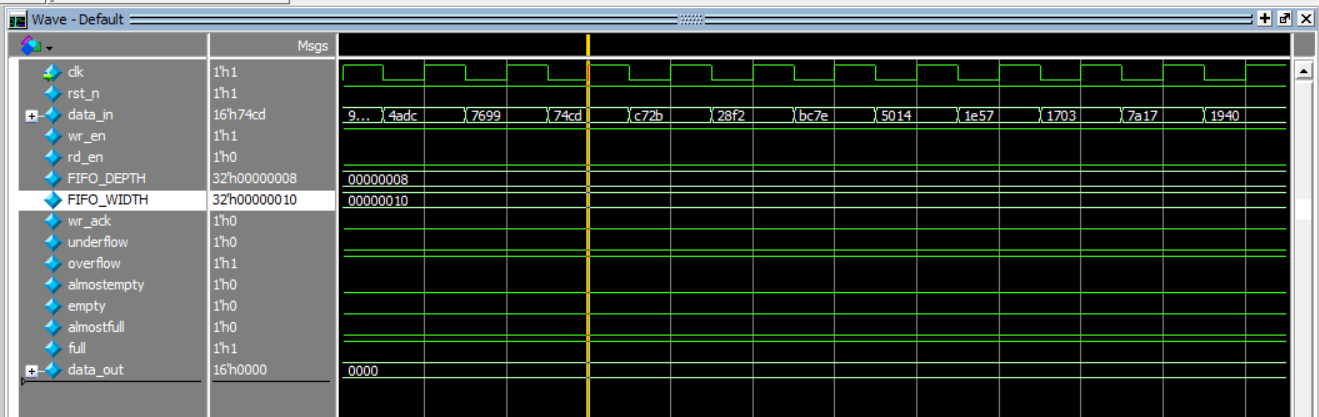
****

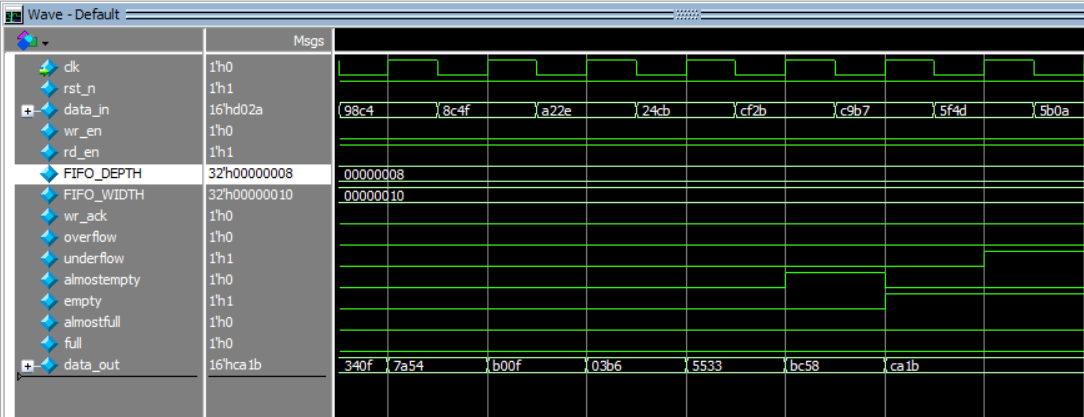
****

****

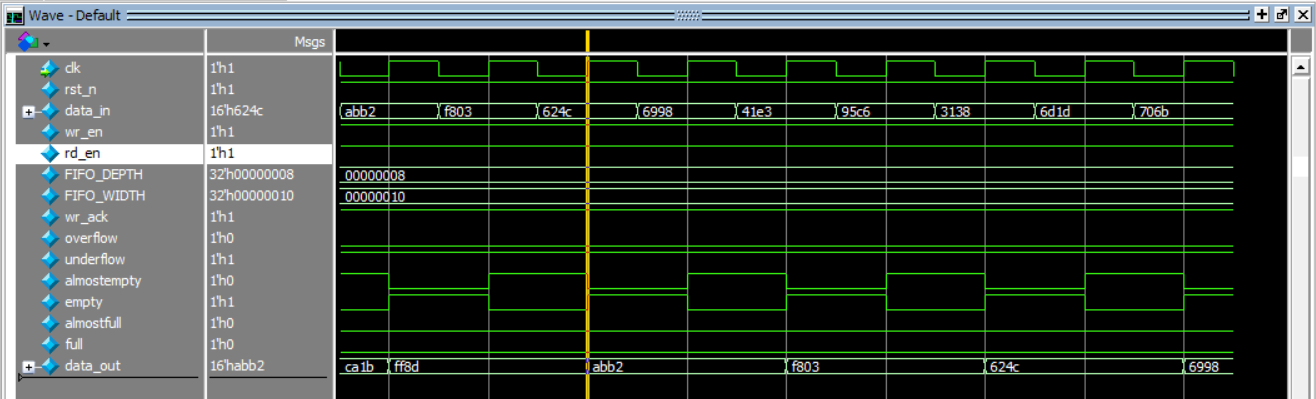
1. **:QuestaSim snippets:**
2. **Reset process:**

****

1. **Write process:**
2. **Read process:**

****

1. **Write\_Read\_Process:**

****

1. **Assertions table:**

|  |  |
| --- | --- |
| **Feature** | **Assertion** |
| When the FIFO is full, the full flag must be asserted. | assert property (@(posedge FIFO\_Inter.clk) ((FIFO.count == FIFO.FIFO\_DEPTH) |-> FIFO\_Inter.full));  cover property (@(posedge FIFO\_Inter.clk) ((FIFO.count == FIFO.FIFO\_DEPTH) |-> FIFO\_Inter.full)); |
| When the FIFO is empty, the empty flag must be asserted. | assert property (@(posedge FIFO\_Inter.clk) ((FIFO.count == 0) |-> FIFO\_Inter.empty));  cover property (@(posedge FIFO\_Inter.clk) ((FIFO.count == 0) |-> FIFO\_Inter.empty)); |
| When the FIFO is almost full, the almostfull flag must be asserted. | assert property (@(posedge FIFO\_Inter.clk) ((FIFO.count == FIFO.FIFO\_DEPTH-1) |-> FIFO\_Inter.almostfull));  cover property (@(posedge FIFO\_Inter.clk) ((FIFO.count == FIFO.FIFO\_DEPTH-1) |-> FIFO\_Inter.almostfull)); |
| When the FIFO is almost empty, the almostempty flag must be asserted. | assert property (@(posedge FIFO\_Inter.clk) ((FIFO.count == 1) |-> FIFO\_Inter.almostempty));  cover property (@(posedge FIFO\_Inter.clk) ((FIFO.count == 1) |-> FIFO\_Inter.almostempty)); |
| When a write request is introduced while the full flag is asserted, the overflow flag must be asserted. | assert property (@(posedge FIFO\_Inter.clk) ((FIFO\_Inter.full & FIFO\_Inter.wr\_en) |=> FIFO\_Inter.overflow));  cover property (@(posedge FIFO\_Inter.clk) ((FIFO\_Inter.full & FIFO\_Inter.wr\_en) |=> FIFO\_Inter.overflow)); |
| When a read request is introduced while the empty flag is asserted, the underflow flag must be asserted. | assert property (@(posedge FIFO\_Inter.clk) ((FIFO\_Inter.empty & FIFO\_Inter.rd\_en) |=> FIFO\_Inter.underflow));  cover property (@(posedge FIFO\_Inter.clk) ((FIFO\_Inter.empty & FIFO\_Inter.rd\_en) |=> FIFO\_Inter.underflow)); |
| When a write request is introduced while there is empty spaces in the FIFO, the wr\_ack flag must be asserted. | assert property (@(posedge FIFO\_Inter.clk) ((FIFO\_Inter.wr\_en && (FIFO.count < FIFO.FIFO\_DEPTH)) |=> FIFO\_Inter.wr\_ack));  cover property (@(posedge FIFO\_Inter.clk) ((FIFO\_Inter.wr\_en && (FIFO.count < FIFO.FIFO\_DEPTH)) |=> FIFO\_Inter.wr\_ack)); |
| When reset is asserted, the counters: wr\_ptr, rd\_ptr, and count, must be cleared. | assert property (@(posedge FIFO\_Inter.clk) ((!FIFO\_Inter.rst\_n) |=> ((!FIFO.wr\_ptr) & (!FIFO.rd\_ptr) & (!FIFO.count))));  cover property (@(posedge FIFO\_Inter.clk) ((!FIFO\_Inter.rst\_n) |=> ((!FIFO.wr\_ptr) & (!FIFO.rd\_ptr) & (!FIFO.count)))); |