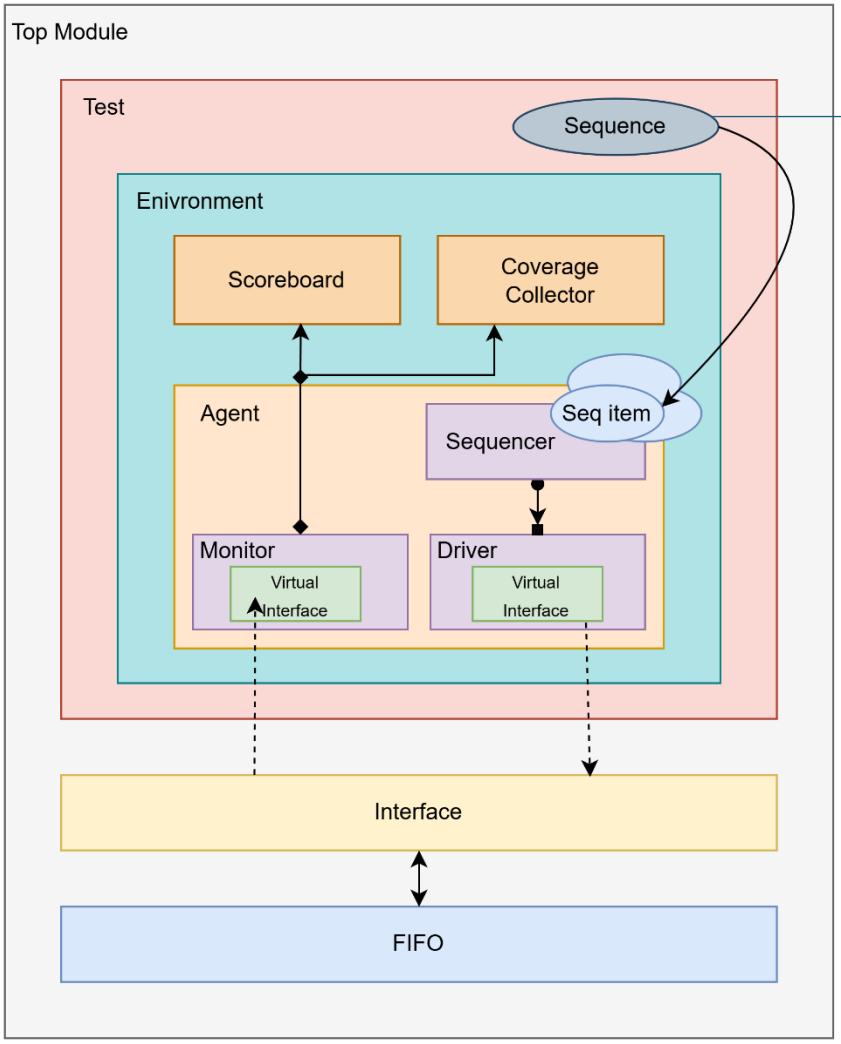
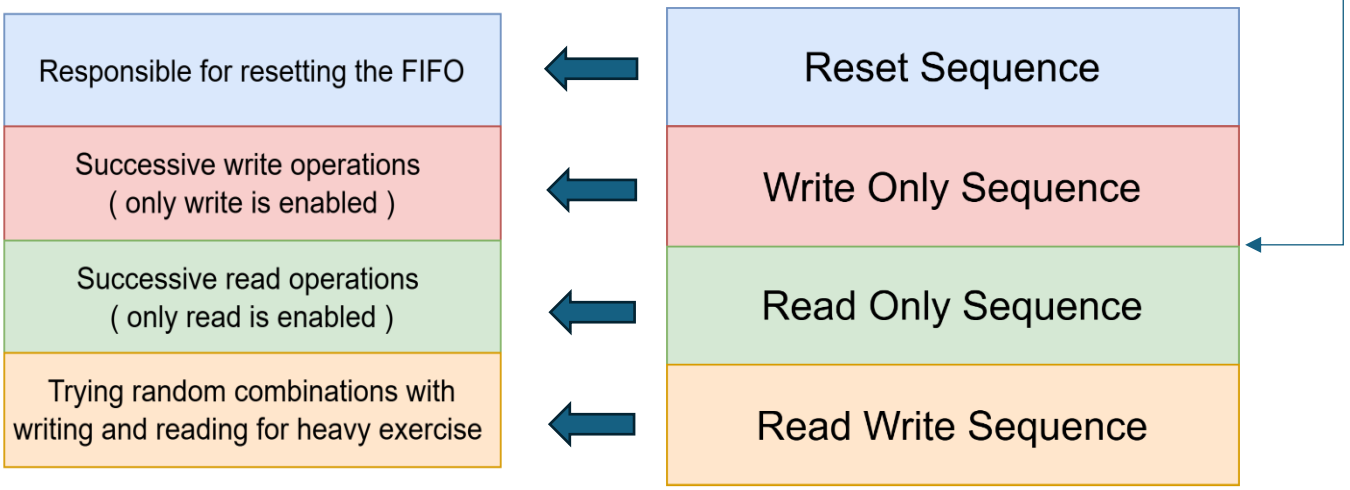
# FIFO Full UVM Environment


AHMED TAREK

# 1) UVM Full Structure



# How does it work?

## 1) Top Module

The Highest level of the verification environment where we instantiate the DUT and Interface, Generate the clock and bind our assertions module. Also it sets the virtual interface in the configuration database so that we can pass the interface to the inner components.

## 2) Interface

Defines the ports of the DUT and provide a structured mechanism to communicate with the DUT and to make the signals accessible to the driver, monitor and scoreboard.

## 3) Test

Overall test strategy defined in this component. It creates the sequences, configures the environment, runs the sequences and sets the configuration object in the database.

## 4) Sequence item and Sequences

### a) Sequence item

Here we define the transaction object that carries inputs/outputs of the FIFO. It helps abstracting data transfer operations and achieving Transaction Level Modeling (**TLM**).

### b) Sequences

- **Reset Sequence**

  Assert the reset signal to the DUT to ensure a clear start and to remove garbage values that are stored inside the internal pointers.

- **Write-Only Sequence**

  Successive ‘write’ operations to verify the write functionality on the FIFO and detect overflow, almost full and full cases.

- **Read-Only Sequence**

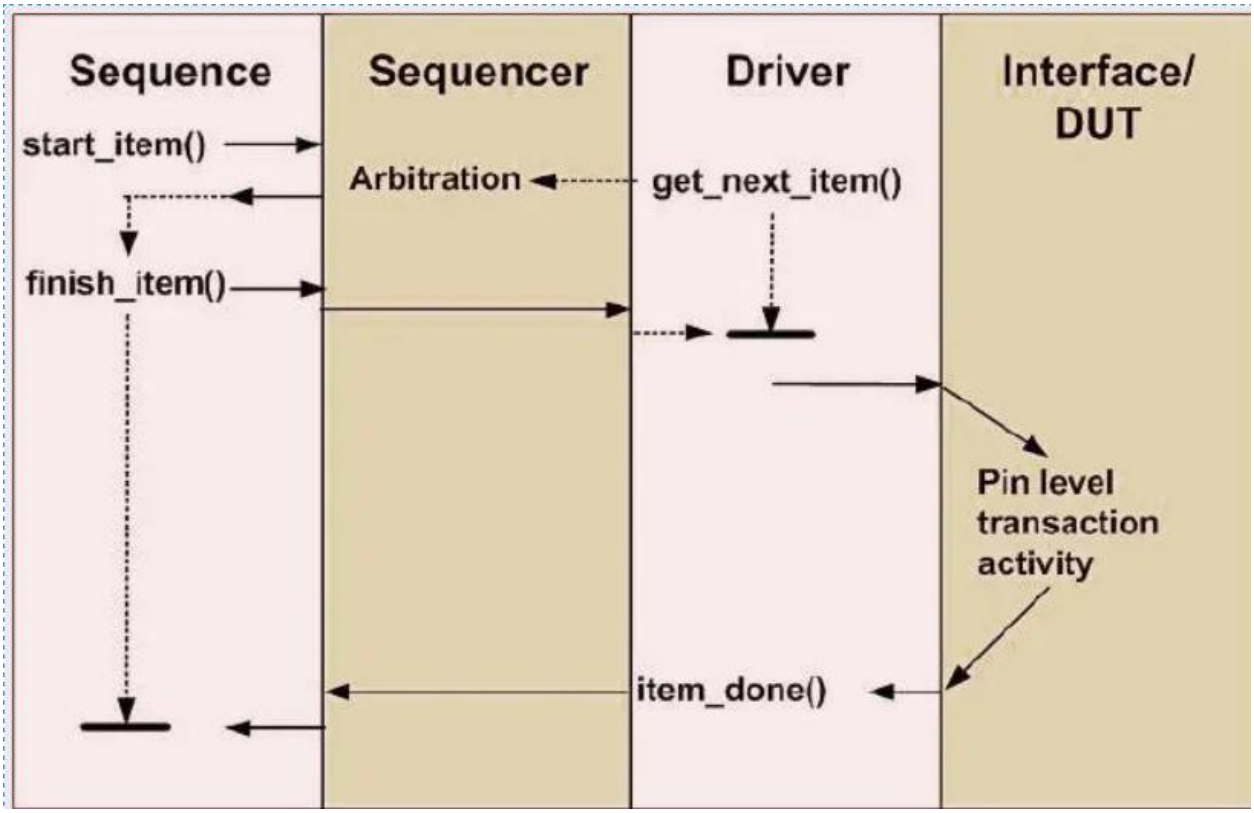
  Successive ‘read’ operations to verify the read functionality from the FIFO and detect underflow, almost empty and empty cases.

- **Read-Write Sequence**

  Heavy exercise for random combinations of simultaneous read and write operations

## 5) Sequencer

Coordinates the execution of the sequences. It sends the sequence item to the driver, which will be later applied to the DUT. It behaves as the mediator/arbiter to synchronize the stimulus generation and driving.




## 6) Driver

Translates the transaction objects received from the sequencer into pin-level activity using the Interface, it also tells the sequencer that this transaction is done let's get the next one.

## 7) Scoreboard

The scoreboard is responsible for checking the correctness of the FIFO. It compares the actual output signals observed by the monitor with the reference/expected results calculated by the reference models and reports how many correct and error cases.

## 8) Coverage Collector

Coverage Collector gathers coverage information, tracking how we exercised the DUT. We define in it some interesting scenarios to make sure that we covered the critical cases. This component gives insight of how much of the design is tested.

## 9) Environment

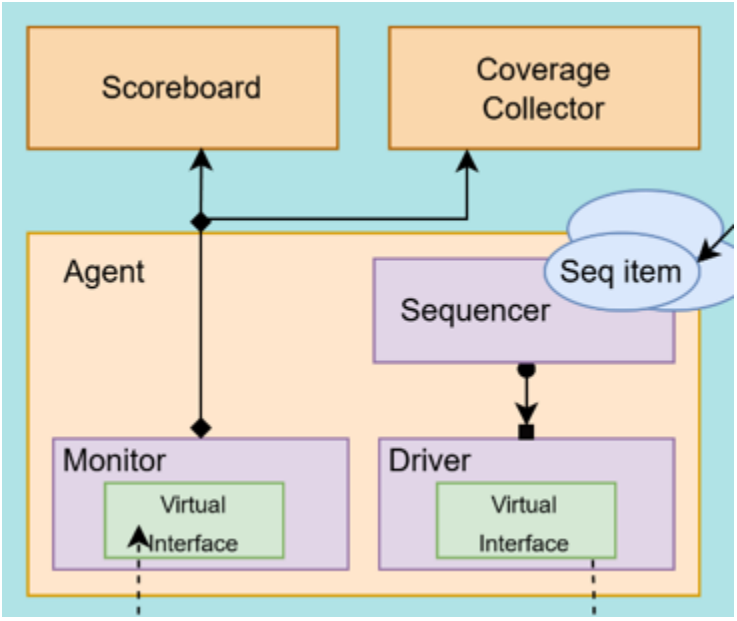
Container of all UVM components such as agents, scoreboard and coverage collector.

## 10) Agent

Encapsulates the driver, monitor and sequencer. It simplifies managing these components by providing a single entity to control them. Also it gets the configuration object from the database and passes the virtual interface to the monitor and driver so that they can access the DUT signals

## 11) Monitor

The monitor captures the signals using the interface and assign them to a transaction object (sequence item) and passes that object to its subscribers such as scoreboard and coverage collector.



# 2) Verification Plan

| File | Usage | | | |
|---|---|---|---|---|
| **FIFO Coverage Collector** | Coverage Collector gathers coverage information, tracking how we exercised the DUT. We define in it some interesting scenarios to make sure that we covered the critical cases. This component gives insight of how much of the design is tested | | | |
| **Label** | **Design Requirement Description** | **Stimulus Generation** | **Functional Coverage** | **Functionality Check** |
| **overflow_cross** | Verify that the overflow signal is asserted when writing to the FIFO and it is full | Write only sequence make sure that we overflow when writing and read write sequence covers the remaining combinations | Cross coverage between overflow signal and read and write enables combinations | Concurrent assertion in the Assertions Module |
| **underflow_cross** | Verify that the underflow signal is asserted when reading from the FIFO and it is empty | Read only sequence make sure that we overflow when writing and read write sequence covers the remaining combinations | Cross coverage between overflow signal and read and write enables combinations | Concurrent assertion in the Assertions Module |
| **empty_cross** | Verify that the empty signal is asserted when reading the last data available in the FIFO | Testing write only then read only till emptying the FIFO and then try read write with all combinations | Cross coverage between empty signal and read and write enables combinations | Concurrent assertion in the Assertions Module |
| **full_cross** | Verify that the full signal is asserted when writing on all available entries in the FIFO | Testing write only then read only till emptying the FIFO and then try read write with all combinations | Cross coverage between full signal and read and write enables combinations | Concurrent assertion in the Assertions Module |
| **Almostfull_cross** | Verify the functionality of the almost full signal to be asserted when the is only one available entry in the FIFO | Testing write only then read only till emptying the FIFO and then try read write with all combinations | Cross coverage between almostfull signal and read and write enables combinations | Concurrent assertion in the Assertions Module |
| **Almostempty_cross** | Verify the functionality of the almost full signal to be asserted when the is only one remaining entry in the FIFO | Testing write only then read only till emptying the FIFO and then try read write with all combinations | Cross coverage between almostempty signal and read and write enables combinations | Concurrent assertion in the Assertions Module |
| **wr_ack_cross** | Verify the functionality of write acknowledge signal so that the wr_ack signal is asserted after a successful write operation | Testing write only then read only till emptying the FIFO and then try read write with all combinations | Cross coverage between wr_ack signal and read and write enables combinations | Concurrent assertion in the Assertions Module |

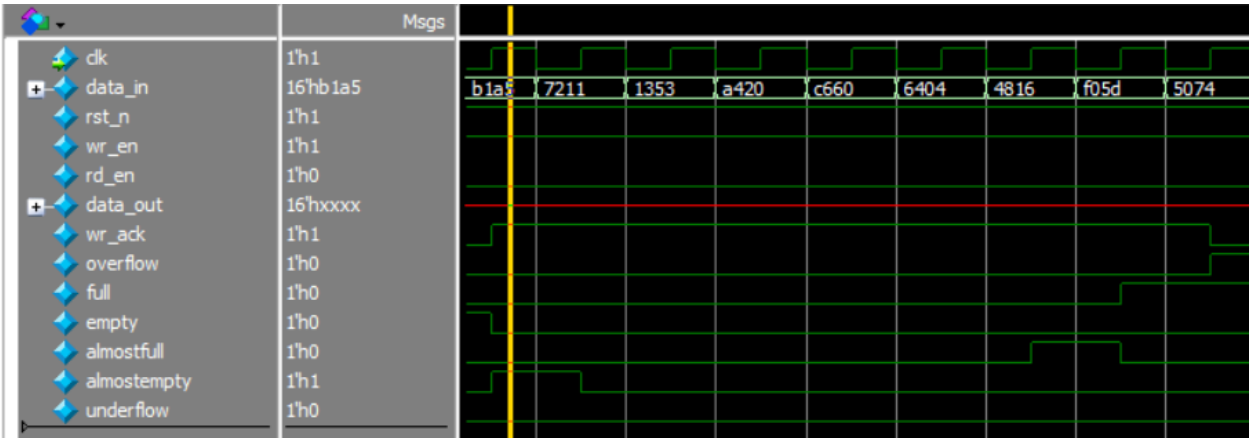
| File | Usage | | | |
|---|---|---|---|---|
| **FIFO Test** | Overall test strategy defined in this component. It creates the sequences, configures the environment, runs the sequences and sets the configuration object in the database. 4 Types of sequences are created to test different scenarios | | | |
| **Label** | **Design Requirement Description** | **Stimulus Generation** | **Functional Coverage** | **Functionality Check** |
| **Reset Sequence** | Verify the reset functionality and clearing the FIFO | Directed at the start of the simulation | Covers reseting internal pointers | Immediate assertion in the assertions module |
| **Write-Only Sequence** | Verify the write functionality to the FIFO | Randomized data in and forcing wr_en to be high and rd_en to be low using 'write_only_c' constraint | Covers full, almost full and overflow coverpoints | Concurrent assertions in the assertions module |
| **Read-Only Sequence** | Verify the read functionality from the FIFO | forcing wr_en to be low and rd_en to be high using 'read_only_c' constraint | Covers empty, almost empty and underflow coverpoints | Concurrent assertions in the assertions module |
| **Read-Write Sequence** | Heavy exercise for different combinations of read and write operations | Randomization under constraint that rd_en is high 30% of the time and wr_en to be high 70% of the time using 'read_write_c' constraint | Covers the cases of {1 , 1} and {0 , 0} with all output signals | Concurrent assertions in the assertions module and a checker in the scoreboard to verify the data out |

# Assertions module

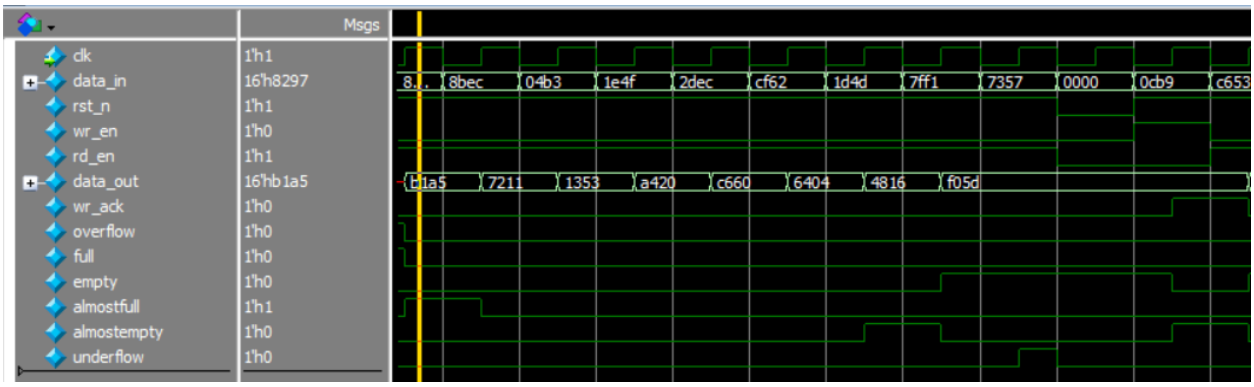
| File | Usage | | |
|---|---|---|---|
| **FIFO SVA** | Checkers to verify the correctness of the signals and internal pointers | | |
| | **Feature** | **Label** | **Assertion** |
| **1]** | After reset is asserted (rst_n = 0), internal pointers and counters must reset to 0 | a_reset | assert final (!DUT.rd_ptr && !DUT.wr_ptr && DUT.count == 0) |
| **2]** | When a write enable signal (wr_en) is active and the FIFO is not full, wr_ack should be asserted to confirm the write operation. | wr_ack_check | "@(posedge clk) disable iff(!rst_n) wr_en && !full \|=> (wr_ack == 1);" |
| **3]** | If a write is attempted when the FIFO is full, overflow should be asserted. | overflow_check | " @(posedge clk) disable iff(!rst_n) wr_en && full \|=> (overflow == 1);" |
| **4]** | If a read is attempted when the FIFO is empty, underflow should be asserted. | underflow_check | " @(posedge clk) disable iff(!rst_n) rd_en && empty \|=> (underflow == 1);" |
| **5]** | When the internal count is zero, the empty flag should be asserted. | empty_check | " @(posedge clk) DUT.count == 0 \|-> (empty == 1);" |
| **6]** | When the internal count equals the FIFO depth, the full flag should be asserted. | full_check | " @(posedge clk) disable iff(!rst_n) (DUT.count == DUT.FIFO_DEPTH) \|-> (full == 1);" |
| **7]** | When the count reaches FIFO depth - 1, almostfull should be asserted. | almostfull_check | " @(posedge clk) disable iff(!rst_n) (DUT.count == DUT.FIFO_DEPTH-1) \|-> (almostfull == 1);" |
| **8]** | When the count equals 1, the almostempty signal should be asserted. | almostempty_check | " @(posedge clk) disable iff(!rst_n) (DUT.count == 1) \|-> (almostempty == 1);" |
| **9]** | internal read pointer wraps around after reading all entries from 0 to 7 | rd_ptr_wrap_check | " @(posedge clk) disable iff(!rst_n) (DUT.rd_ptr == DUT.FIFO_DEPTH-1 && rd_en && !empty) \|=> (DUT.rd_ptr == 0);" |
| **10]** | internal write pointer wraps around after writing all entries from 0 to 7 | wr_ptr_wrap_check | " @(posedge clk) disable iff(!rst_n) (DUT.wr_ptr == DUT.FIFO_DEPTH-1 && wr_en && !full) \|=> (DUT.wr_ptr == 0);" |
| **11]** | Internal count is incremented on successful write operation | counter_check_up | " @(posedge clk) disable iff(!rst_n) (wr_en && !full && !rd_en) \|=> (DUT.count == $past(DUT.count) + 1);" |
| **12]** | Internal count is decremented on successful read operation | counter_check_down | " @(posedge clk) disable iff(!rst_n) (rd_en && !empty && !wr_en) \|=> (DUT.count == $past(DUT.count) - 1); " |
| **13]** | Transition of the count register from 8 to 0 ( wrap around ) | cnt_wrap_check | " @(posedge clk) ($past(DUT.count , 2) == DUT.FIFO_DEPTH && !$past(rst_n)) \|-> (DUT.count == 0);" |
| **14]** | Internal pointers cannot exceed the FIFO_DEPTH entries in any given time. | ptr_threshold | " @(posedge clk) DUT.rd_ptr < DUT.FIFO_DEPTH && DUT.wr_ptr < DUT.FIFO_DEPTH;" |

# 3) Questasim snapshots

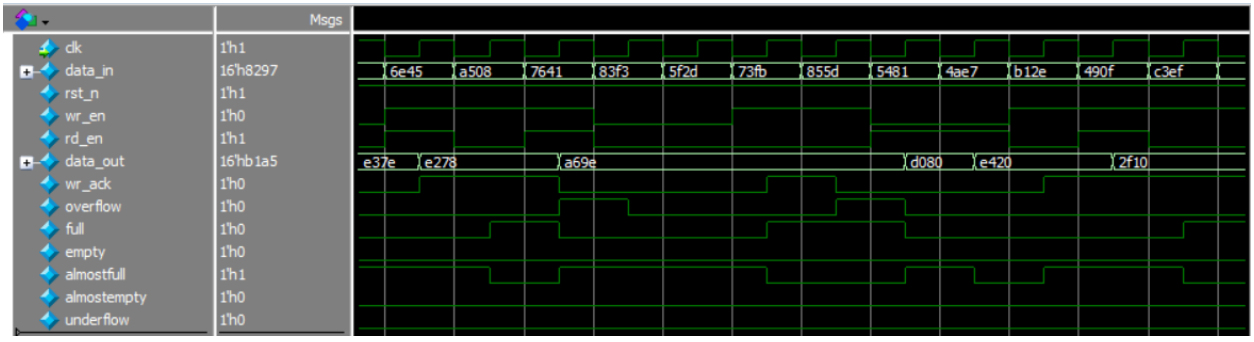
❖ Write-Only Sequence




❖ Read-Only Sequence




❖ Read-Write Sequence



# 4) Bug Report

a) Fixed the reset behavior to reset wr_ack, overflow and underflow

b) De-assert overflow on successful write operation

c) De-assert underflow on successful read operation

d) Changed underflow from combinational to sequential logic

e) Missing cases in the part that handles simultaneous read and write cases to calculate the internal ‘count’

f) Change in the calculation of almost full. Changed from ( depth – 2 ) to ( depth – 1 )

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        wr_ptr <= 0;
    end
    else if (wr_en && count < FIFO_DEPTH) begin
        mem[wr_ptr] <= data_in;
        wr_ack <= 1;
        wr_ptr <= wr_ptr + 1;
    end
    else begin
        wr_ack <= 0;
        if (full & wr_en)
            overflow <= 1;
        else
            overflow <= 0;
    end
end
```

**Before**

```
always @(posedge FIFO_if.clk or negedge FIFO_if.rst_n) begin
    if (!FIFO_if.rst_n) begin
        // fix 1 : reset wr_ack on reset
        FIFO_if.wr_ack <= 0;
        // fix 2 : reset overflow on reset
        FIFO_if.overflow <= 0;
        wr_ptr <= 0;
    end
    else if (FIFO_if.wr_en && count < FIFO_DEPTH) begin
        mem[wr_ptr] <= FIFO_if.data_in;
        FIFO_if.wr_ack <= 1;
        wr_ptr <= wr_ptr + 1;
        FIFO_if.overflow <= 0; // fix 3 : reset overflow on successfu
    end
    else begin
        FIFO_if.wr_ack <= 0;
        if (FIFO_if.full & FIFO_if.wr_en)
            FIFO_if.overflow <= 1;
        else
            FIFO_if.overflow <= 0;
    end
end
```

**After**

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        rd_ptr <= 0;
    end
    else if (rd_en && count != 0) begin
        data_out <= mem[rd_ptr];
        rd_ptr <= rd_ptr + 1;
    end
end
```

**Before**

```
always @(posedge FIFO_if.clk or negedge FIFO_if.rst_n) begin
    if (!FIFO_if.rst_n) begin
        rd_ptr <= 0;
        // fix 4 : reset underflow on reset
        FIFO_if.underflow <= 0;
    end
    else if (FIFO_if.rd_en && count != 0) begin
        FIFO_if.data_out <= mem[rd_ptr];
        rd_ptr <= rd_ptr + 1;
        FIFO_if.underflow <= 0; // fix 5 : reset underflow on successful read
    end
    else begin
        // fix 6 add underflow condition to sequential logic
        if (FIFO_if.empty && FIFO_if.rd_en) begin
            FIFO_if.underflow <= 1;
        end else begin
            FIFO_if.underflow <= 0;
        end
    end
end
```

**After**

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        count <= 0;
    end
    else begin
        if  ( ({wr_en, rd_en} == 2'b10) && !full)
            count <= count + 1;
        else if ( ({wr_en, rd_en} == 2'b01) && !empty)
            count <= count - 1;
    end
end

assign full = (count == FIFO_DEPTH)? 1 : 0;
assign empty = (count == 0)? 1 : 0;
assign underflow = (empty && rd_en)? 1 : 0;
assign almostfull = (count == FIFO_DEPTH-2)? 1 : 0;
assign almostempty = (count == 1)? 1 : 0;
```

**Before**

```
always @(posedge FIFO_if.clk or negedge FIFO_if.rst_n) begin
    if (!FIFO_if.rst_n) begin
        count <= 0;
    end
    else begin
        if (FIFO_if.wr_en && FIFO_if.rd_en && !FIFO_if.full && !FIFO_if.empty)
            count <= count; // simultaneous write and read
        else if (FIFO_if.wr_en && !FIFO_if.full)
            count = count + 1;
        else if (FIFO_if.rd_en && !FIFO_if.empty)
            count = count - 1;
        else
            count = count; // no change in count
    end
end

assign FIFO_if.full = (count == FIFO_DEPTH)? 1 : 0;
assign FIFO_if.empty = (count == 0)? 1 : 0;
// assign FIFO_if.underflow = (FIFO_if.empty && FIFO_if.rd_en)? 1 : 0; // change to sequential logic
assign FIFO_if.almostfull = (count == FIFO_DEPTH-1)? 1 : 0;
assign FIFO_if.almostempty = (count == 1)? 1 : 0;
```

**After**

# 5) Functional Coverage

| Name | Class Type | Coverage | Goal | % of Goal | Status | Included |
|---|---|---|---|---|---|---|
| /fifo_coverage_pkg/fifo_coverage | | 100.0% | | | | |
| TYPE FIFO_cvg_group | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CVP FIFO_cvg_group::wr_en_cp | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CVP FIFO_cvg_group::rd_en_cp | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CVP FIFO_cvg_group::overflow_cp | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CVP FIFO_cvg_group::underflow_cp | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CVP FIFO_cvg_group::full_cp | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CVP FIFO_cvg_group::empty_cp | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CVP FIFO_cvg_group::almostfull_cp | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CVP FIFO_cvg_group::almostempty_cp | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CVP FIFO_cvg_group::wr_ack_cp | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CROSS FIFO_cvg_group::overflow_cross | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CROSS FIFO_cvg_group::underflow_cross | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CROSS FIFO_cvg_group::full_cross | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CROSS FIFO_cvg_group::empty_cross | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CROSS FIFO_cvg_group::almostfull_cross | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CROSS FIFO_cvg_group::almostempty_cross | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |
| CROSS FIFO_cvg_group::wr_ack_cross | fifo_coverage | 100.0% | 100 | 100.0% | | ✓ |

| Name | Language | Enabled | Log | Count | AtLeast | Limit | Weight | Cmplt % | Cmplt graph | Included |
|---|---|---|---|---|---|---|---|---|---|---|
| /fifo_top/DUT/fifo_sva_inst/cover__ptrs_threshold | SVA | ✓ | Off | 20020 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__count_wrap_check | SVA | ✓ | Off | 234 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__counter_check_down | SVA | ✓ | Off | 1531 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__counter_check_up | SVA | ✓ | Off | 6618 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__wr_ptr_wrap_check | SVA | ✓ | Off | 817 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__rd_ptr_wrap_check | SVA | ✓ | Off | 333 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__almostempty_check | SVA | ✓ | Off | 1917 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__almostfull_check | SVA | ✓ | Off | 3445 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__full_check | SVA | ✓ | Off | 4953 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__empty_check | SVA | ✓ | Off | 2477 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__underflow_check | SVA | ✓ | Off | 426 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__overflow_check | SVA | ✓ | Off | 3303 | 1 | Unli... | 1 | 100% | | ✓ |
| /fifo_top/DUT/fifo_sva_inst/cover__wr_ack_check | SVA | ✓ | Off | 9384 | 1 | Unli... | 1 | 100% | | ✓ |

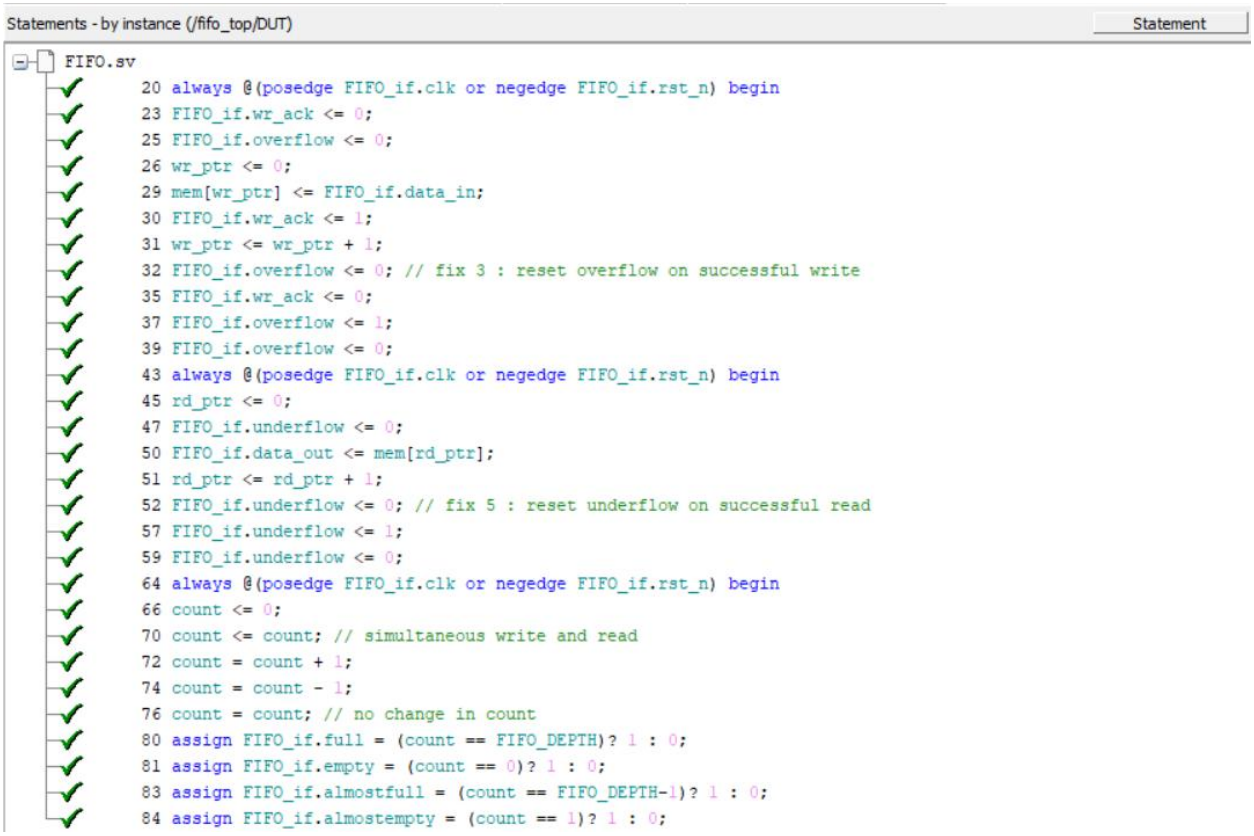
## 6) Sequential Domain Coverage

```
ASSERTION RESULTS:
---------------------------------------------------
Name                 File(Line)          Failure Pass
                                         Count   Count
---------------------------------------------------
/fifo_top/DUT/fifo_sva_inst/assert__counter_check_down
                     fifo_sva.sv(102)          0     1
/fifo_top/DUT/fifo_sva_inst/assert__counter_check_up
                     fifo_sva.sv(101)          0     1
/fifo_top/DUT/fifo_sva_inst/assert__almostempty_check
                     fifo_sva.sv(98)           0     1
/fifo_top/DUT/fifo_sva_inst/assert__almostfull_check
                     fifo_sva.sv(97)           0     1
/fifo_top/DUT/fifo_sva_inst/assert__full_check
                     fifo_sva.sv(96)           0     1
/fifo_top/DUT/fifo_sva_inst/assert__empty_check
                     fifo_sva.sv(95)           0     1
/fifo_top/DUT/fifo_sva_inst/assert__underflow_check
                     fifo_sva.sv(94)           0     1
/fifo_top/DUT/fifo_sva_inst/assert__overflow_check
                     fifo_sva.sv(93)           0     1
/fifo_top/DUT/fifo_sva_inst/assert__wr_ack_check
                     fifo_sva.sv(92)           0     1
/fifo_top/DUT/fifo_sva_inst/a_reset
                     fifo_sva.sv(15)           0     1
/fifo_top/DUT/fifo_sva_inst/ptr_wrap_1
                     fifo_sva.sv(99)           0     1
/fifo_top/DUT/fifo_sva_inst/ptr_wrap_2
                     fifo_sva.sv(100)          0     1
/fifo_top/DUT/fifo_sva_inst/cnt_wrap_check
                     fifo_sva.sv(103)          0     1
/fifo_top/DUT/fifo_sva_inst/ptr_threshold
                     fifo_sva.sv(104)          0     1
```

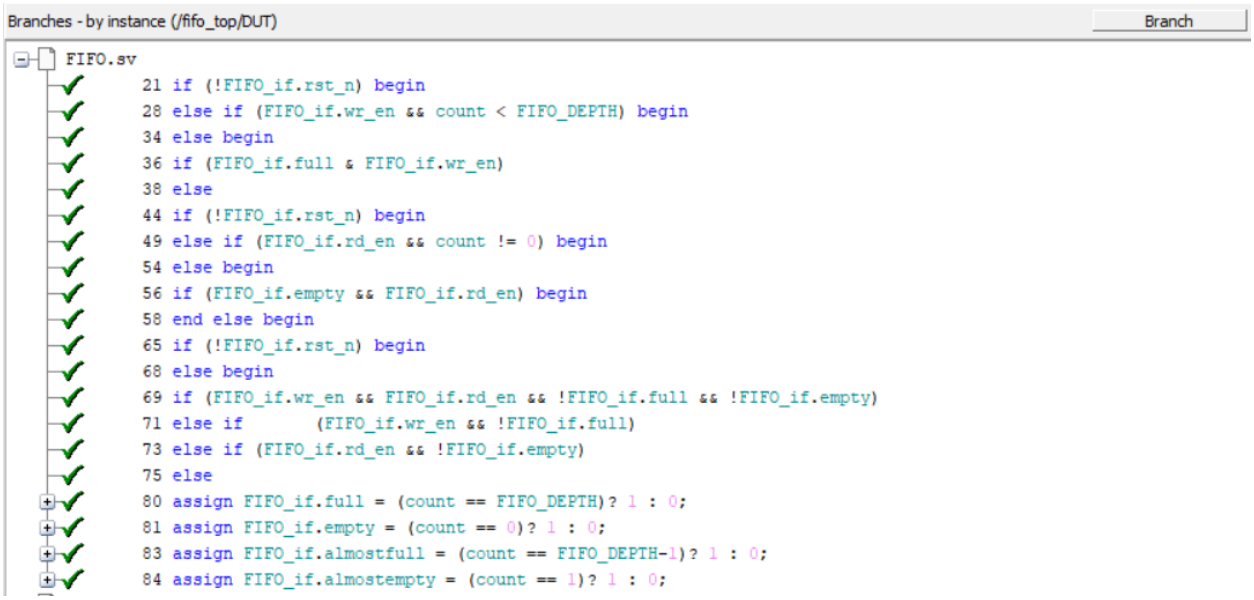
```
DIRECTIVE COVERAGE:
--------------------------------------------------------------------------------------------
Name                                    Design Design   Lang File(Line)      Count Status
                                        Unit   UnitType
--------------------------------------------------------------------------------------------
/fifo_top/DUT/fifo_sva_inst/cover__ptrs_threshold
                                        fifo_sva Verilog  SVA  fifo_sva.sv(118)20020 Covered
/fifo_top/DUT/fifo_sva_inst/cover__count_wrap_check
                                        fifo_sva Verilog  SVA  fifo_sva.sv(117)  234 Covered
/fifo_top/DUT/fifo_sva_inst/cover__counter_check_down
                                        fifo_sva Verilog  SVA  fifo_sva.sv(116) 1531 Covered
/fifo_top/DUT/fifo_sva_inst/cover__counter_check_up
                                        fifo_sva Verilog  SVA  fifo_sva.sv(115) 6618 Covered
/fifo_top/DUT/fifo_sva_inst/cover__wr_ptr_wrap_check
                                        fifo_sva Verilog  SVA  fifo_sva.sv(114)  817 Covered
/fifo_top/DUT/fifo_sva_inst/cover__rd_ptr_wrap_check
                                        fifo_sva Verilog  SVA  fifo_sva.sv(113)  333 Covered
/fifo_top/DUT/fifo_sva_inst/cover__almostempty_check
                                        fifo_sva Verilog  SVA  fifo_sva.sv(112) 1917 Covered
/fifo_top/DUT/fifo_sva_inst/cover__almostfull_check
                                        fifo_sva Verilog  SVA  fifo_sva.sv(111) 3445 Covered
/fifo_top/DUT/fifo_sva_inst/cover__full_check
                                        fifo_sva Verilog  SVA  fifo_sva.sv(110) 4953 Covered
/fifo_top/DUT/fifo_sva_inst/cover__empty_check
                                        fifo_sva Verilog  SVA  fifo_sva.sv(109) 2477 Covered
/fifo_top/DUT/fifo_sva_inst/cover__underflow_check
                                        fifo_sva Verilog  SVA  fifo_sva.sv(108)  426 Covered
/fifo_top/DUT/fifo_sva_inst/cover__overflow_check
                                        fifo_sva Verilog  SVA  fifo_sva.sv(107) 3303 Covered
/fifo_top/DUT/fifo_sva_inst/cover__wr_ack_check
                                        fifo_sva Verilog  SVA  fifo_sva.sv(106) 9384 Covered

TOTAL DIRECTIVE COVERAGE: 100.0%  COVERS: 13
```

# 7) Code Coverage

## a) Statement Coverage




## b) Branch Coverage



c) Toggle Coverage