FIFO Verification

Using UVM environment

supervisor: Eng. Kareem Wassem
Name: Ahmed Amr Ali

Table of Contents

FIFO Design

1) FIFO Overview

A **FIFO (First-In, First-Out)** is a specialized type of memory or buffer that allows data to be written into and read out in the same order it was entered. This design is frequently used in hardware systems where data must be processed in the order it arrives. It serves as a queue where the first element written is the first one to be read. FIFOs are crucial for managing data flow between two subsystems that operate at different speeds, such as when synchronizing communication between processors or between hardware modules.

Key Features of FIFO

Data Storage and Flow: Data is written into the FIFO through an input port and read out via an output port. The system ensures that data is removed in the order it was added, making it ideal for managing data streams.

Synchronous Operation: This design is synchronous, meaning that it operates with a clock signal (clk). All data transfers (write and read operations) happen on clock edges, ensuring synchronization between different parts of the system.

FIFO Width and Depth:

- **FIFO_WIDTH**: The width of the data bus, which determines how many bits are written and read at once.
- **FIFO_DEPTH**: The total number of memory locations available in the FIFO, determining its storage capacity.

Typical use cases

1. **Data Rate Matching:** FIFOs are often used to smooth out differences in data rates between components that operate at different speeds.
2. **Interfacing Different Systems:** It is common in systems where data is transmitted between processors and peripherals, such as in communication devices, networking equipment, or audio/video processing.
3. **Pipeline Buffers:** In pipelined architectures, FIFOs are used to store intermediate results or data while the system continues processing other tasks.

How it works

The FIFO works by utilizing two main operations: **write** and **read**. The flow is controlled by a series of signals:

1. **Write Operation:**
   - Data is presented on the data_in input.
   - When the wr_en (write enable) signal is asserted and the FIFO is not full, data is written into the next available memory location.
   - If the FIFO becomes full, indicated by the full flag, no further write operations are allowed until space is freed.
2. **Read Operation:**
   - Data is read from the FIFO through the data_out port.
   - When the rd_en (read enable) signal is asserted and the FIFO is not empty, the oldest data is removed from the FIFO and presented at the output.
   - If the FIFO becomes empty, indicated by the empty flag, no more data can be read until more is written.
3. **Overflow and Underflow:**
   - **Overflow:** When a write is attempted but the FIFO is full, the overflow signal is asserted, and the new data is discarded to avoid corruption.
   - **Underflow:** When a read is attempted but the FIFO is empty, the underflow signal is asserted, and no data is read.
4. **Status Signals:**
   - **Full/Empty:** Indicate when the FIFO cannot accept more data (full) or when there is no data to read (empty).
   - **Almost Full/Almost Empty:** These intermediate flags (almostfull and almostempty) provide early warnings when the FIFO is about to become full or empty, allowing for better control over data flow.
   - **Write Acknowledge (wr_ack):** Indicates a successful write operation.

In this design, assertions are added to verify correct FIFO behavior and ensure data integrity during write and read operations. The system is robust, handling edge cases like overflow and underflow gracefully.

2) Specs

Parameters

- FIFO_WIDTH: DATA in/out and memory word width (default: 16)
- FIFO_DEPTH: Memory depth (default: 8)

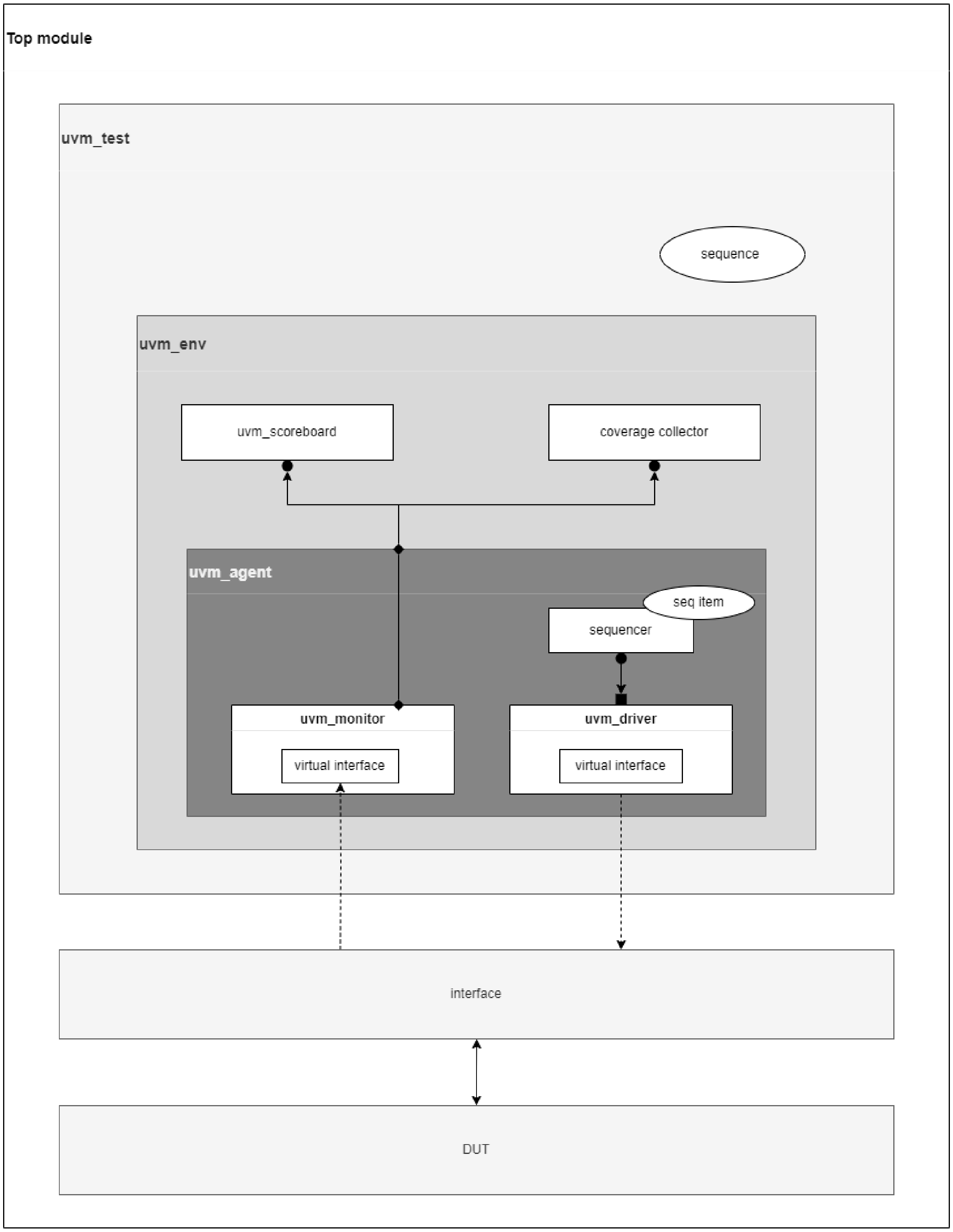
ports

Port	Direction	Function
data_in	Input	Write Data: The input data bus used when writing the FIFO.
wr_en		Write Enable: If the FIFO is not full, asserting this signal causes data (on data_in) to be written into the FIFO
rd_en		Read Enable: If the FIFO is not empty, asserting this signal causes data (on data_out) to be read from the FIFO
clk		Clock signal
rst_n		Active low asynchronous reset
data_out	Output	Read Data: The sequential output data bus used when reading from the FIFO.
full		Full Flag: When asserted, this combinational output signal indicates that the FIFO is full. Write requests are ignored when the FIFO is full, initiating a write when the FIFO is full is not destructive to the contents of the FIFO.
almostfull		Almost Full: When asserted, this combinational output signal indicates that only one more write can be performed before the FIFO is full.
empty		Empty Flag: When asserted, this combinational output signal indicates that the FIFO is empty. Read requests are ignored when the FIFO is empty, initiating a read while empty is not destructive to the FIFO.
almostempty		Almost Empty: When asserted, this output combinational signal indicates that only one more read can be performed before the FIFO goes to empty.
overflow		Overflow: This sequential output signal indicates that a write request (wr_en) was rejected because the FIFO is full. Overflowing the FIFO is not destructive to the contents of the FIFO.
underflow		Underflow: This sequential output signal Indicates that the read request (rd_en) was rejected because the FIFO is empty. Under flowing the FIFO is not destructive to the FIFO.
wr_ack		Write Acknowledge: This sequential output signal indicates that a write request (wr_en) has succeeded.

Note: If a read and write enables were high and the FIFO was empty, only writing will take place and vice versa if the FIFO was full.

3) UVM structure

Structure:



Flow:

1. Top Module (uvm_test)

The uvm_test is the starting point of the UVM testbench. It defines the stimulus generation and controls the flow of the simulation. It typically includes the instantiation of the environment (uvm_env), sequences, and virtual interface connections.

In this case, the top module would instantiate the UVM environment tailored to the FIFO. It would create a sequence to generate the necessary transactions such as write (wr_en), read (rd_en), and other control signals that interface with the DUT (FIFO).

2. Environment (uvm_env)

The UVM environment (uvm_env) encapsulates all the verification components, such as the agents, scoreboard, and coverage collectors. The environment coordinates the communication between different UVM components and drives the stimulus to the DUT (FIFO).

For the FIFO, the environment will include:

- **uvm_agent**: Contains the sequencer, driver, and monitor that will drive stimulus and observe the DUT.
- **uvm_scoreboard**: Will be used to compare the actual output of the FIFO with expected results.
- **Coverage collector**: Tracks the functional coverage of the FIFO under test, ensuring that all corner cases are covered (like full, empty, almostfull, almostempty conditions).

3. Driving the Interface (uvm_driver)

The uvm_driver component is responsible for driving signals into the DUT. It takes high-level transactions from the sequencer and translates them into pin-level signals.

For the FIFO:

- The driver will receive transaction items that include values for data_in, wr_en, and rd_en.
- If wr_en is asserted and the FIFO is not full, the driver will place the data_in onto the data bus and drive it into the FIFO.
- Similarly, for rd_en, if the FIFO is not empty, the driver will read the output from the FIFO (data_out).

The driver will also interact with control signals such as clk, rst_n, and handle flags like full, empty, almostfull, and almostempty, ensuring that reads and writes are correctly synchronized with the FIFO's internal state.

4. Sequencing (uvm_sequencer and Sequence)

The uvm_sequencer is responsible for sending a stream of sequence items (transactions) to the driver. These sequence items define the scenarios you want to test, such as:

- Writing into the FIFO when it's not full.
- Attempting to write when the FIFO is full (triggering overflow).
- Reading from the FIFO when it's not empty.
- Attempting to read when the FIFO is empty (triggering underflow).

For example, one sequence could involve writing several values into the FIFO until it's full, then attempting an additional write to check the overflow behavior.

5. Monitoring (uvm_monitor)

The uvm_monitor passively observes the transactions happening on the interface. It samples signals without driving them and records the DUT's behavior for later analysis.

For the FIFO, the monitor will:

- Observe the data being written to and read from the FIFO.
- Track the status of flags like full, almostfull, empty, and almostempty.
- Report any unexpected behavior, such as a write happening when the FIFO is full or a read occurring when the FIFO is empty.

The monitor's primary purpose is to capture both input and output signals and pass them to the scoreboard for comparison with expected results.

6. Analyzing the Output (uvm_scoreboard)

The uvm_scoreboard is used to check if the FIFO behaves as expected. It compares the actual outputs observed by the monitor to the expected outputs.

In this case:

- The expected output for data_out would be compared against the data written into the FIFO (considering the sequence of reads and writes).
- The scoreboard will also check for proper flag behavior (e.g., the full flag should assert when the FIFO is full and de-assert after a read operation).
- Additionally, it will validate overflow and underflow conditions by monitoring if the FIFO prevents further writes when full or reads when empty.

7. Coverage Collector

A coverage collector tracks which conditions of the FIFO have been tested, such as:

- Full and empty states.
- Almostfull and almostempty thresholds.
- Proper assertion and de-assertion of overflow and underflow conditions.

This helps ensure that all functional aspects of the FIFO are adequately covered during verification.

4) The RTL design

Bugs:

Detected Bugs:

- The almostfull flag is raised when there are 2 empty places not 1

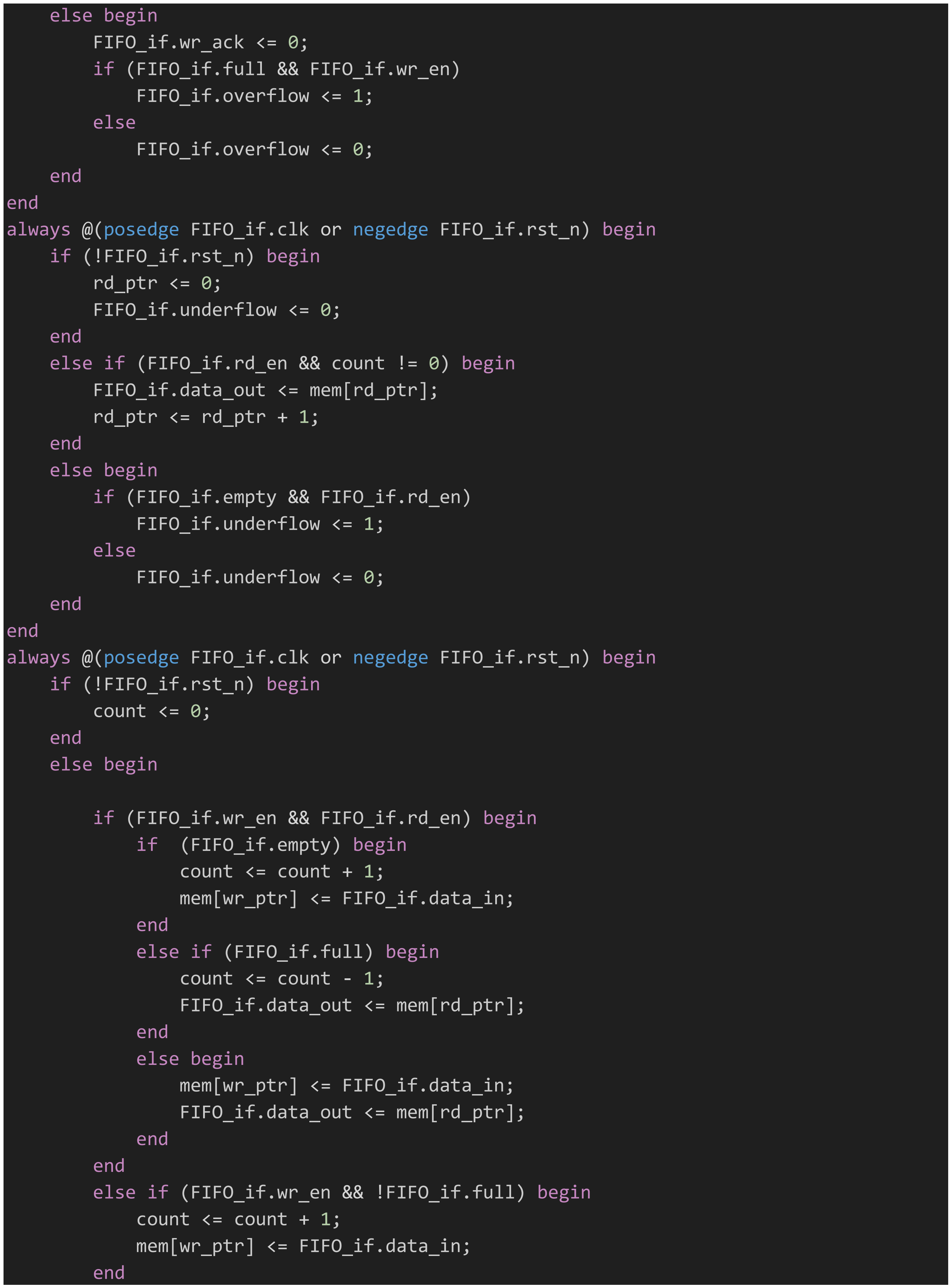
```
assign FIFO_if.almostfull = (count == FIFO_if.FIFO_DEPTH-2)? 1 : 0;
```

- When resetting the overflow, underflow doesn't reset
- When rd_en and wr_en are both asserted the FIFO doesn't do any of them sometimes, also the design didn't include when the signals are asserted when full of empty flag is raised
- Underflow signal is supposed to be sequential

```
assign FIFO_if.underflow = (FIFO_if.empty && FIFO_if.rd_en)? 1 : 0;
```

Code without bugs + assertions + interface:

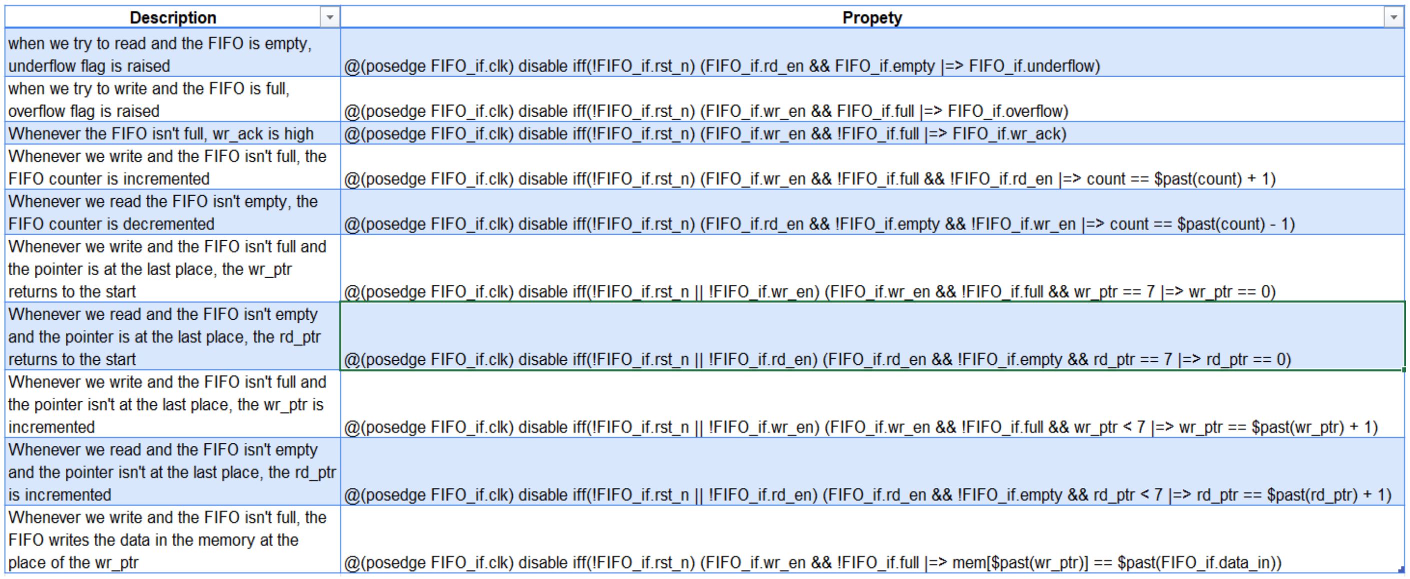
```
// Author: Kareem Waseem
// Course: Digital Verification using SV & UVM
//
// Description: FIFO Design
//
////////////////////////////////////////////////////////////////////////////////
module FIFO(FIFO_interface.DUT FIFO_if);

localparam max_fifo_addr = $clog2(FIFO_if.FIFO_DEPTH);

reg [FIFO_if.FIFO_WIDTH-1:0] mem [FIFO_if.FIFO_DEPTH-1:0];

reg [max_fifo_addr-1:0] wr_ptr, rd_ptr;
reg [max_fifo_addr:0] count;

always @(posedge FIFO_if.clk or negedge FIFO_if.rst_n) begin
    if (!FIFO_if.rst_n) begin
        wr_ptr <= 0;
        FIFO_if.overflow <= 0;
    end
    else if (FIFO_if.wr_en && count < FIFO_if.FIFO_DEPTH) begin
        mem[wr_ptr] <= FIFO_if.data_in;
        FIFO_if.wr_ack <= 1;
        wr_ptr <= wr_ptr + 1;
    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
always @(posedge FIFO_if.clk or negedge FIFO_if.rst_n) begin
    if (!FIFO_if.rst_n) begin
        rd_ptr <= 0;
        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;
    end
    else begin
        if (FIFO_if.empty && FIFO_if.rd_en)
            FIFO_if.underflow <= 1;
        else
            FIFO_if.underflow <= 0;
    end
end
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) begin
            if  (FIFO_if.empty) begin
                count <= count + 1;
                mem[wr_ptr] <= FIFO_if.data_in;
            end
            else if (FIFO_if.full) begin
                count <= count - 1;
                FIFO_if.data_out <= mem[rd_ptr];
            end
            else begin
                mem[wr_ptr] <= FIFO_if.data_in;
                FIFO_if.data_out <= mem[rd_ptr];
            end
        end
        else if (FIFO_if.wr_en && !FIFO_if.full) begin
            count <= count + 1;
            mem[wr_ptr] <= FIFO_if.data_in;
        end
        else if (FIFO_if.rd_en && !FIFO_if.empty) begin
            count <= count - 1;
            FIFO_if.data_out <= mem[rd_ptr];
        end
    end
end

assign FIFO_if.full = (count == FIFO_if.FIFO_DEPTH)? 1 : 0;
assign FIFO_if.empty = (count == 0)? 1 : 0;
assign FIFO_if.almostfull = (count == FIFO_if.FIFO_DEPTH-1)? 1 : 0;
assign FIFO_if.almostempty = (count == 1)? 1 : 0;

always_comb begin
    if(!FIFO_if.rst_n) begin
        a_reset: assert final(wr_ptr == 0 && rd_ptr == 0 && count == 0);
        c_reset: cover final(wr_ptr == 0 && rd_ptr == 0 && count == 0);
    end
    if (count == FIFO_if.FIFO_DEPTH) begin
        full_a: assert final (FIFO_if.full == 1);
        full_c: cover final (FIFO_if.full == 1);
    end
    if (count == FIFO_if.FIFO_DEPTH - 1) begin
        almostfull_a: assert final (FIFO_if.almostfull == 1);
        almostfull_c: cover final (FIFO_if.almostfull == 1);
    end
    if (count == 0) begin
        empty_a: assert final (FIFO_if.empty == 1);
        empty_c: cover final (FIFO_if.empty == 1);
    end
    if (count == 1) begin
        almostempty_a: assert final (FIFO_if.almostempty == 1);
        almostempty_c: cover final (FIFO_if.almostempty == 1);
    end
end

underflow_a: assert property (@(posedge FIFO_if.clk) FIFO_if.rd_en && FIFO_if.empty
|=> FIFO_if.underflow);
underflow_c: cover property (@(posedge FIFO_if.clk) FIFO_if.rd_en && FIFO_if.empty |=>
FIFO_if.underflow);

overflow_a: assert property (@(posedge FIFO_if.clk) FIFO_if.wr_en && FIFO_if.full |=>
FIFO_if.overflow);
overflow_c: cover property (@(posedge FIFO_if.clk) FIFO_if.wr_en && FIFO_if.full |=>
FIFO_if.overflow);
```

```
wrk_ack_a: assert property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n)
(FIFO_if.wr_en && !FIFO_if.full |=> FIFO_if.wr_ack));
wrk_ack_c: cover property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n)
(FIFO_if.wr_en && !FIFO_if.full |=> FIFO_if.wr_ack));

mem_count_up_a: assert property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n)
(FIFO_if.wr_en && !FIFO_if.full && !FIFO_if.rd_en |=> count == $past(count) + 1));
mem_count_up_c: cover property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n)
(FIFO_if.wr_en && !FIFO_if.full && !FIFO_if.rd_en |=> count == $past(count) + 1));

mem_count_down_a: assert property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n)
(FIFO_if.rd_en && !FIFO_if.empty && !FIFO_if.wr_en |=> count == $past(count) - 1));
mem_count_down_c: cover property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n)
(FIFO_if.rd_en && !FIFO_if.empty && !FIFO_if.wr_en |=> count == $past(count) - 1));

wr_ptr_zero_a: assert property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n ||
!FIFO_if.wr_en) (FIFO_if.wr_en && !FIFO_if.full && wr_ptr == 7 |=> wr_ptr == 0));
wr_ptr_zero_c: cover property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n ||
!FIFO_if.wr_en) (FIFO_if.wr_en && !FIFO_if.full && wr_ptr == 7 |=> wr_ptr == 0));

rd_ptr_zero_a: assert property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n ||
!FIFO_if.rd_en) (FIFO_if.rd_en && !FIFO_if.empty && rd_ptr == 7 |=> rd_ptr == 0));
rd_ptr_zero_c: cover property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n ||
!FIFO_if.rd_en) (FIFO_if.rd_en && !FIFO_if.empty && rd_ptr == 7 |=> rd_ptr == 0));

wr_ptr_over_zero_a: assert property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n
|| !FIFO_if.wr_en) (FIFO_if.wr_en && !FIFO_if.full && wr_ptr < 7 |=> wr_ptr ==
$past(wr_ptr) + 1));
wr_pt_over_zero_c: cover property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n
|| !FIFO_if.wr_en) (FIFO_if.wr_en && !FIFO_if.full && wr_ptr < 7 |=> wr_ptr ==
$past(wr_ptr) + 1));

rd_ptr_over_zero_a: assert property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n
|| !FIFO_if.rd_en) (FIFO_if.rd_en && !FIFO_if.empty && rd_ptr < 7 |=> rd_ptr ==
$past(rd_ptr) + 1));
rd_ptr_over_zero_c: cover property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n
|| !FIFO_if.rd_en) (FIFO_if.rd_en && !FIFO_if.empty && rd_ptr < 7 |=> rd_ptr ==
$past(rd_ptr) + 1));

write_in_mem_a: assert property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n)
(FIFO_if.wr_en && !FIFO_if.full |=> mem[$past(wr_ptr)] == $past(FIFO_if.data_in)));
write_in_mem_c: cover property (@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n)
(FIFO_if.wr_en && !FIFO_if.full |=> mem[$past(wr_ptr)] == $past(FIFO_if.data_in)));
endmodule
```

Assertions table



Description	Propety
when we try to read and the FIFO is empty, underflow flag is raised	@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n) (FIFO_if.rd_en && FIFO_if.empty |=> FIFO_if.underflow)
when we try to write and the FIFO is full, overflow flag is raised	@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n) (FIFO_if.wr_en && FIFO_if.full |=> FIFO_if.overflow)
Whenever the FIFO isn't full, wr_ack is high	@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n) (FIFO_if.wr_en && !FIFO_if.full |=> FIFO_if.wr_ack)
Whenever we write and the FIFO isn't full, the FIFO counter is incremented	@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n) (FIFO_if.wr_en && !FIFO_if.full && !FIFO_if.rd_en |=> count == $past(count) + 1)
Whenever we read the FIFO isn't empty, the FIFO counter is decremented	@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n) (FIFO_if.rd_en && !FIFO_if.empty && !FIFO_if.wr_en |=> count == $past(count) - 1)
Whenever we write and the FIFO isn't full and the pointer is at the last place, the wr_ptr returns to the start	@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n || !FIFO_if.wr_en) (FIFO_if.wr_en && !FIFO_if.full && wr_ptr == 7 |=> wr_ptr == 0)
Whenever we read and the FIFO isn't empty and the pointer is at the last place, the rd_ptr returns to the start	@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n || !FIFO_if.rd_en) (FIFO_if.rd_en && !FIFO_if.empty && rd_ptr == 7 |=> rd_ptr == 0)
Whenever we write and the FIFO isn't full and the pointer isn't at the last place, the wr_ptr is incremented	@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n || !FIFO_if.wr_en) (FIFO_if.wr_en && !FIFO_if.full && wr_ptr < 7 |=> wr_ptr == $past(wr_ptr) + 1)
Whenever we read and the FIFO isn't empty and the pointer isn't at the last place, the rd_ptr is incremented	@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n || !FIFO_if.rd_en) (FIFO_if.rd_en && !FIFO_if.empty && rd_ptr < 7 |=> rd_ptr == $past(rd_ptr) + 1)
Whenever we write and the FIFO isn't full, the FIFO writes the data in the memory at the place of the wr_ptr	@(posedge FIFO_if.clk) disable iff(!FIFO_if.rst_n) (FIFO_if.wr_en && !FIFO_if.full |=> mem[$past(wr_ptr)] == $past(FIFO_if.data_in))

5) Verification plan

Label	Description	Stimulus Generation	Functional Coverage (Later)	Functionality Check
FIFO_1	When the reset is asserted, the output data_out, the pointers, and the counter value should be low	Directed at the start of the simulation then randomized with constraint that drive the reset to be off most of the time	-	immediate assertions to check for the functionality of the rst_n (a_reset)
FIFO_2	when wr_en is high the FIFO will write the data_in in the memory sequentialy untill the memory is full then the FIFO won't write any other data_in. when there is one empty place in the memory the almostfull flag will be high and when there is no place the full flag will be high	the data_in will be randomized while the wr_en & rst_n is high and the rd_en is low	cross coverage between the wr_en signal, rd_en signal and each control signal (full, almostfull, empty, almostempty, overflow, underflow, wr_ack) resulting in a total number of 7 cross cover points	refrence model function to check for the data_out and read functionality. immediate assertion to check for the full flag functionality (full_a) immediate assertion to check for the empty flag functionality (empty_a) immediate assertion to check for the almostfull flag functionality (almost_full_a) immediate assertion to check for the almostempty flag functionality (almost_empty_a) concurrent assertion to check for the underflow flag functionallity (underflow_a) concurrent assertion to check for the overflow flag functionallity (overflow_a) concurrent assertion to check for the wr_ack flag functionallity (wr_ack_a) concurrent assertion to check for the counter increment and decrement functionallity (mem_count_up_a)(mem_count_down_a) concurrent assertion to check for the rd_ptr and wr_ptr increment functionallity (wr_ptr_a)(rd_ptr_a) concurrent assertion to check for the write in mem and data_in functionallity (write_in_mem_a)
FIFO_3	when rd_en is high the FIFO will read from the memory on the data_out sequentialy untill the memory is empty then the FIFO won't read anymore. when there is one occupied place in the memory the almostempty flag will be high and when there is no occupied the empty flag will be high	the data_in won't be needed in this part since the wr_en low while rst_n & rd_en is high.	cross coverage between the wr_en signal, rd_en signal and each control signal (full, almostfull, empty, almostempty, overflow, underflow, wr_ack) resulting in a total number of 7 cross cover points	refrence model function to check for the data_out and read functionality. immediate assertion to check for the full flag functionality (full_a) immediate assertion to check for the empty flag functionality (empty_a) immediate assertion to check for the almostfull flag functionality (almost_full_a) immediate assertion to check for the almostempty flag functionality (almost_empty_a) concurrent assertion to check for the underflow flag functionallity (underflow_a) concurrent assertion to check for the overflow flag functionallity (overflow_a) concurrent assertion to check for the wr_ack flag functionallity (wr_ack_a) concurrent assertion to check for the counter increment and decrement functionallity (mem_count_up_a)(mem_count_down_a) concurrent assertion to check for the rd_ptr and wr_ptr increment functionallity (wr_ptr_a)(rd_ptr_a) concurrent assertion to check for the write in mem and data_in functionallity (write_in_mem_a)
FIFO_4	when wr_en & rd_en are high the FIFO will read & write at the same time and the count of the data in the memory should be fixed.	the data_in will be randomized while the wr_en & rst_n & rd_en are high	cross coverage between the wr_en signal, rd_en signal and each control signal (full, almostfull, empty, almostempty, overflow, underflow, wr_ack) resulting in a total number of 7 cross cover points	refrence model function to check for the data_out and read functionality. immediate assertion to check for the full flag functionality (full_a) immediate assertion to check for the empty flag functionality (empty_a) immediate assertion to check for the almostfull flag functionality (almost_full_a) immediate assertion to check for the almostempty flag functionality (almost_empty_a) concurrent assertion to check for the underflow flag functionallity (underflow_a) concurrent assertion to check for the overflow flag functionallity (overflow_a) concurrent assertion to check for the wr_ack flag functionallity (wr_ack_a) concurrent assertion to check for the counter increment and decrement functionallity (mem_count_up_a)(mem_count_down_a) concurrent assertion to check for the rd_ptr and wr_ptr increment functionallity (wr_ptr_a)(rd_ptr_a) concurrent assertion to check for the write in mem and data_in functionallity (write_in_mem_a)
FIFO_5	when wr_en is high the FIFO will write the data_in in the memory sequentialy untill the memory is full then the FIFO won't write any other data_in. when rd_en is high the FIFO will read from the memory on the data_out sequentialy untill the memory is empty then the FIFO won't read anymore. when there is one empty place in the memory the almostfull flag will be high and when there is no place the full flag will be high. when there is one occupied place in the memory the almostempty flag will be high and when there is no occupied the empty flag will be high.	randomized under the constraint that rst_n should be disabled 95% of the time, the wr_en enabled 70% of the time and the rd_en enabled 30% of the time.	cross coverage between the wr_en signal, rd_en signal and each control signal (full, almostfull, empty, almostempty, overflow, underflow, wr_ack) resulting in a total number of 7 cross cover points	refrence model function to check for the data_out and read functionality. immediate assertion to check for the full flag functionality (full_a) immediate assertion to check for the empty flag functionality (empty_a) immediate assertion to check for the almostfull flag functionality (almost_full_a) immediate assertion to check for the almostempty flag functionality (almost_empty_a) concurrent assertion to check for the underflow flag functionallity (underflow_a) concurrent assertion to check for the overflow flag functionallity (overflow_a) concurrent assertion to check for the wr_ack flag functionallity (wr_ack_a) concurrent assertion to check for the counter increment and decrement functionallity (mem_count_up_a)(mem_count_down_a) concurrent assertion to check for the rd_ptr and wr_ptr increment functionallity (wr_ptr_a)(rd_ptr_a) concurrent assertion to check for the write in mem and data_in functionallity (write_in_mem_a)

6) Testbench

FIFO_shared_pkg.sv

```systemverilog
package shared_pkg;
    parameter FIFO_WIDTH = 16;
    parameter FIFO_DEPTH = 8;

endpackage
```

FIFO_top.sv

```systemverilog
import uvm_pkg::*;
import FIFO_test::*;
`include "uvm_macros.svh"

module FIFO_top ();
    bit clk;

    initial begin
        clk = 0;
        forever
            #1 clk = ~clk;
    end

    FIFO_interface FIFO_if (clk);
    FIFO dut (FIFO_if);

    initial begin
        uvm_config_db#(virtual FIFO_interface)::set(null, "uvm_test_top",
"FIFO_interface", FIFO_if);
        run_test("FIFO_test");
    end
endmodule
```

FIFO_interface.sv

```systemverilog
import shared_pkg::*;

interface FIFO_interface (input bit clk);

    parameter FIFO_WIDTH = 16;
    parameter FIFO_DEPTH = 8;

    logic [FIFO_WIDTH-1:0] data_in;
    logic rst_n, wr_en, rd_en;

    logic [FIFO_WIDTH-1:0] data_out;
    logic wr_ack, overflow;
    logic full, empty, almostfull, almostempty, underflow;

    modport DUT (input clk, rst_n, data_in, wr_en, rd_en, output data_out, wr_ack,
overflow, full, empty, almostfull, almostempty, underflow);
    modport MONITOR (input clk, rst_n, data_in, wr_en, rd_en, data_out, wr_ack, overflow,
full, empty, almostfull, almostempty, underflow);
    modport TEST (output rst_n, data_in, wr_en, rd_en, input clk, data_out, wr_ack,
overflow, full, empty, almostfull, almostempty, underflow);
endinterface
```

FIFO_test.sv

```
package FIFO_test;
    import uvm_pkg::*;
    import FIFO_config::*;
    import FIFO_main_sequence::*;
    import FIFO_reset_sequence::*;
    import FIFO_env::*;

    `include "uvm_macros.svh"

    class FIFO_test extends  uvm_test;

        `uvm_component_utils(FIFO_test)
        FIFO_config FIFO_cfg;
        FIFO_env env;
        FIFO_main_sequence main_seq;
        FIFO_reset_sequence reset_seq;

        function new(string name = "FIFO_test", uvm_component parent=null);
            super.new(name, parent);
        endfunction

        function void build_phase(uvm_phase phase);
            super.build_phase(phase);

            FIFO_cfg = FIFO_config::type_id::create("FIFO_cfg");
            main_seq = FIFO_main_sequence::type_id::create("main_seq");
            reset_seq = FIFO_reset_sequence::type_id::create("reset_seq");
            env = FIFO_env::type_id::create("env", this);

            if (!(uvm_config_db#(virtual FIFO_interface)::get(this, "", "FIFO_interface",
FIFO_cfg.FIFO_vif)))
                `uvm_fatal("build_phase", "unable to get vitual interface from top
module");

            uvm_config_db#(FIFO_config)::set(this, "*", "CFG", FIFO_cfg);
        endfunction
        task run_phase(uvm_phase phase);
            super.run_phase(phase);
            phase.raise_objection(this);

            `uvm_info("run phase", "Reset Asserted", UVM_LOW);
            reset_seq.start(env.agt.sqr);
            `uvm_info("run phase", "Reset Asserted", UVM_LOW);

            `uvm_info("run phase", "Stimulus Generation Started", UVM_LOW);
            main_seq.start(env.agt.sqr);
            `uvm_info("run phase", "Stimulus Generation Ended", UVM_LOW);

            phase.drop_objection(this);
        endtask
    endclass
endpackage
```

FIFO_env.sv

```
package FIFO_env;
    import uvm_pkg::*;
    import FIFO_agent::*;
    import FIFO_coverage::*;
    import FIFO_scoreboard::*;

    `include "uvm_macros.svh"

    class FIFO_env extends  uvm_env;

        `uvm_component_utils(FIFO_env)
        FIFO_agent agt;
        FIFO_coverage cov;
        FIFO_scoreboard sb;

        function new(string name = "FIFO_env", uvm_component parent=null);
            super.new(name, parent);
        endfunction

        function void build_phase(uvm_phase phase);
            super.build_phase(phase);
                agt = FIFO_agent::type_id::create("agt", this);
                cov = FIFO_coverage::type_id::create("cov", this);
                sb = FIFO_scoreboard::type_id::create("sb", this);
        endfunction

        function void connect_phase(uvm_phase phase);
            super.connect_phase(phase);
            agt.agt_ap.connect(cov.cov_export);
            agt.agt_ap.connect(sb.sb_export);
        endfunction
    endclass
endpackage
```

FIFO_agent.sv

```systemverilog
package FIFO_agent;
    import uvm_pkg::*;
    import FIFO_sequence_item::*;
    import FIFO_sequencer::*;
    import FIFO_config::*;
    import FIFO_driver::*;
    import FIFO_monitor::*;

    `include "uvm_macros.svh"

    class FIFO_agent extends  uvm_agent;

        `uvm_component_utils(FIFO_agent)
        uvm_analysis_port#(FIFO_sequence_item) agt_ap;
        FIFO_config FIFO_cfg;
        FIFO_monitor mon;
        FIFO_driver drv;
        FIFO_sequencer sqr;

        function new(string name = "FIFO_agent", uvm_component parent=null);
            super.new(name, parent);
        endfunction

        function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            if (!(uvm_config_db#(FIFO_config)::get(this, "", "CFG", FIFO_cfg)))
                `uvm_fatal("build phase", "Agent - unable to get configuration object")
            sqr = FIFO_sequencer::type_id::create("sqr", this);
            drv = FIFO_driver::type_id::create("drv", this);
            mon = FIFO_monitor::type_id::create("mon", this);
            agt_ap = new("agt_ap", this);
        endfunction

        function void connect_phase(uvm_phase phase);
            super.connect_phase(phase);
            drv.FIFO_vif = FIFO_cfg.FIFO_vif;
            mon.FIFO_vif = FIFO_cfg.FIFO_vif;
            drv.seq_item_port.connect(sqr.seq_item_export);
            mon.mon_ap.connect(agt_ap);
        endfunction
    endclass
endpackage
```

FIFO_driver.sv

```
package FIFO_driver;
    import uvm_pkg::*;
    import FIFO_sequence_item::*;

    `include "uvm_macros.svh"

    class FIFO_driver extends  uvm_driver#(FIFO_sequence_item);

        `uvm_component_utils(FIFO_driver)
        FIFO_sequence_item stim_seq_item;
        virtual FIFO_interface FIFO_vif;
        function new(string name = "FIFO_driver", uvm_component parent=null);
            super.new(name, parent);
        endfunction

        task run_phase(uvm_phase phase);
            super.run_phase(phase);
            forever begin
                stim_seq_item = FIFO_sequence_item::type_id::create("stim_seq_item");
                seq_item_port.get_next_item(stim_seq_item);
                FIFO_vif.data_in = stim_seq_item.data_in;
                FIFO_vif.rst_n = stim_seq_item.rst_n;
                FIFO_vif.wr_en = stim_seq_item.wr_en;
                FIFO_vif.rd_en = stim_seq_item.rd_en;
                @(negedge FIFO_vif.clk);
                seq_item_port.item_done();
                `uvm_info("run phase", stim_seq_item.convert2string(), UVM_HIGH)
            end
        endtask
    endclass
endpackage
```

FIFO_coverage.sv

```
package FIFO_coverage;
    import uvm_pkg::*;
    import FIFO_sequence_item::*;

    `include "uvm_macros.svh"

    class FIFO_coverage extends  uvm_component;

        `uvm_component_utils(FIFO_coverage)
        FIFO_sequence_item cov_item;
        uvm_analysis_export#(FIFO_sequence_item) cov_export;
        uvm_tlm_analysis_fifo#(FIFO_sequence_item) cov_fifo;

        covergroup FIFO_cvg;
            wr_full: cross cov_item.wr_en, cov_item.full;
            wr_rd_almostfull: cross cov_item.wr_en, cov_item.rd_en, cov_item.almostfull;
            wr_rd_empty: cross cov_item.wr_en, cov_item.rd_en, cov_item.empty;
            wr_rd_almostempty: cross cov_item.wr_en, cov_item.rd_en,
cov_item.almostempty;
            wr_rd_overflow: cross cov_item.wr_en, cov_item.rd_en, cov_item.overflow;
            wr_rd_underflow: cross cov_item.wr_en, cov_item.rd_en, cov_item.underflow;
            wr_rd_wr_ack: cross cov_item.wr_en, cov_item.rd_en, cov_item.wr_ack;
        endgroup

        function new(string name = "FIFO_coverage", uvm_component parent=null);
            super.new(name, parent);
            FIFO_cvg = new();
        endfunction

        function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            cov_export = new("cov_export", this);
            cov_fifo = new("cov_fifo", this);
        endfunction

        function void connect_phase(uvm_phase phase);
            super.connect_phase(phase);
            cov_export.connect(cov_fifo.analysis_export);
        endfunction
        task run_phase(uvm_phase phase);
            super.run_phase(phase);
            forever begin
                cov_fifo.get(cov_item);
                FIFO_cvg.sample();
            end
        endtask
    endclass
endpackage
```

FIFO_config.sv

```
package FIFO_config;
    import uvm_pkg::*;
    `include "uvm_macros.svh"

    class FIFO_config extends  uvm_object;

        `uvm_object_utils(FIFO_config)
        virtual FIFO_interface FIFO_vif;

        function new(string name = "FIFO_config");
            super.new(name);
        endfunction

    endclass
endpackage
```

FIFO_monitor.sv

```
package FIFO_monitor;
    import uvm_pkg::*;
    import FIFO_sequence_item::*;
    `include "uvm_macros.svh"
    class FIFO_monitor extends  uvm_monitor;
        `uvm_component_utils(FIFO_monitor)
        virtual FIFO_interface FIFO_vif;
        FIFO_sequence_item rsp_seq_item;
        uvm_analysis_port#(FIFO_sequence_item) mon_ap;

        function new(string name = "FIFO_monitor", uvm_component parent=null);
            super.new(name, parent);
        endfunction

        function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            mon_ap = new("mon_ap", this);
        endfunction

        task run_phase(uvm_phase phase);
            super.run_phase(phase);
            forever begin
                rsp_seq_item = FIFO_sequence_item::type_id::create("rsp_seq_item");
                @(negedge FIFO_vif.clk);
                rsp_seq_item.data_in = FIFO_vif.data_in;
                rsp_seq_item.rst_n = FIFO_vif.rst_n;
                rsp_seq_item.wr_en = FIFO_vif.wr_en;
                rsp_seq_item.rd_en = FIFO_vif.rd_en;
                rsp_seq_item.data_out = FIFO_vif.data_out;
                rsp_seq_item.wr_ack = FIFO_vif.wr_ack;
                rsp_seq_item.overflow = FIFO_vif.overflow;
                rsp_seq_item.full = FIFO_vif.full;
                rsp_seq_item.empty = FIFO_vif.empty;
                rsp_seq_item.almostfull = FIFO_vif.almostfull;
                rsp_seq_item.almostempty = FIFO_vif.almostempty;
                rsp_seq_item.underflow = FIFO_vif.underflow;
                mon_ap.write(rsp_seq_item);
                `uvm_info("run phase", rsp_seq_item.convert2string(), UVM_HIGH)
            end
        endtask
    endclass
endpackage
```

FIFO_scoreboard.sv

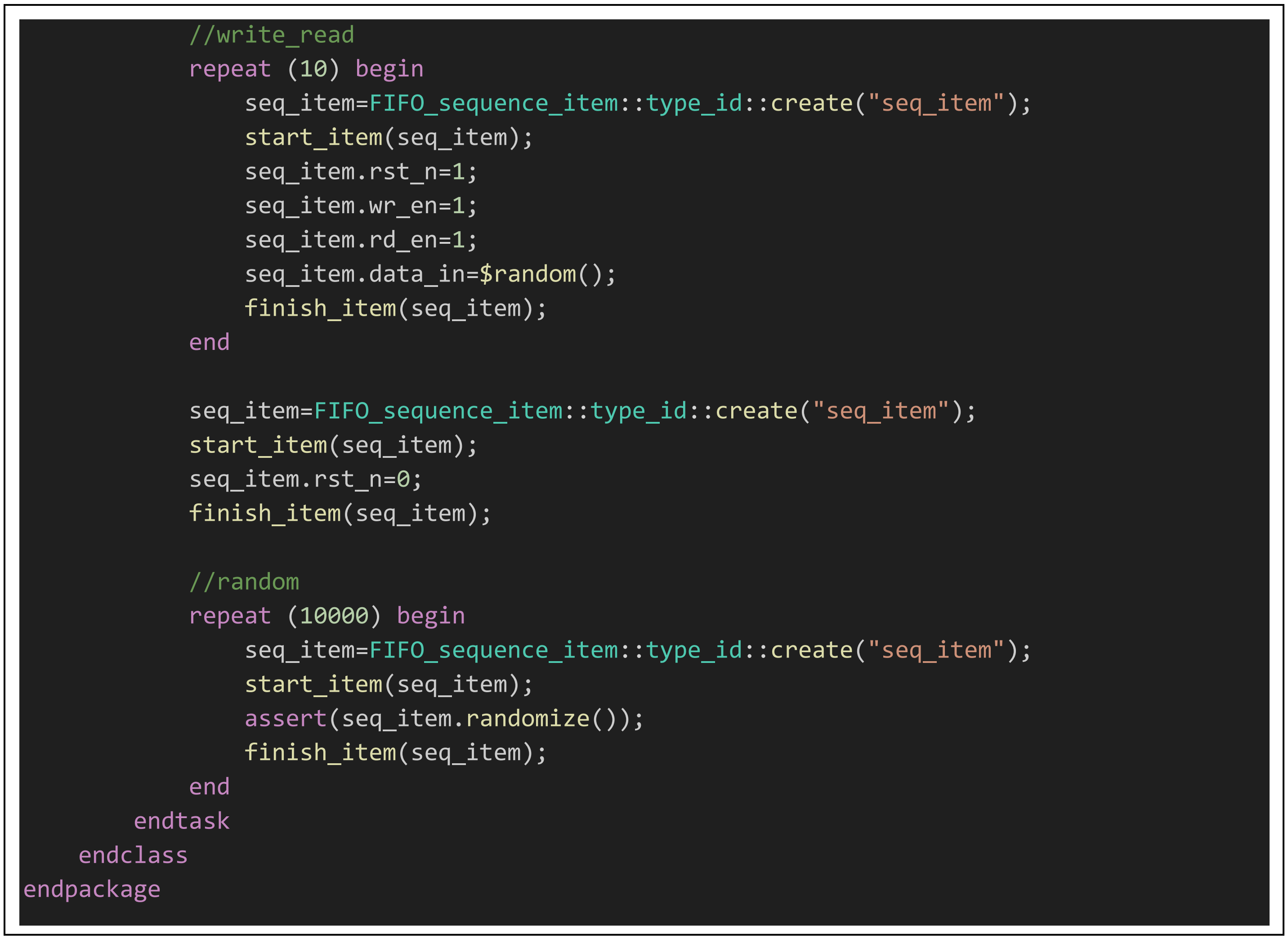
```
package FIFO_scoreboard;
    import uvm_pkg::*;
    import FIFO_sequence_item::*;
    import shared_pkg::*;

    `include "uvm_macros.svh"

    logic [FIFO_WIDTH-1:0] data_out_ref;

    logic [FIFO_WIDTH - 1 : 0] mem_ref_queue [FIFO_DEPTH];

    bit [2:0] wr_ptr = 0;
    bit [2:0] rd_ptr = 0;
    integer count = 0;

    class FIFO_scoreboard extends  uvm_scoreboard;

        `uvm_component_utils(FIFO_scoreboard)
        uvm_analysis_export#(FIFO_sequence_item) sb_export;
        uvm_tlm_analysis_fifo#(FIFO_sequence_item) sb_fifo;
        FIFO_sequence_item sb_item;
        integer correct_count = 0;
        integer error_count = 0;

        function new(string name = "FIFO_scoreboard", uvm_component parent=null);
            super.new(name, parent);
        endfunction

        function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            sb_export = new("sb_export", this);
            sb_fifo = new("sb_fifo", this);
        endfunction

        function void connect_phase(uvm_phase phase);
            super.connect_phase(phase);
            sb_export.connect(sb_fifo.analysis_export);
        endfunction
        task run_phase(uvm_phase phase);
            super.run_phase(phase);
            forever begin
                sb_fifo.get(sb_item);
                ref_model(sb_item);
                if (sb_item.data_out != data_out_ref) begin
                    `uvm_error("run phase", $sformatf("Comparison failed Transaction
Received by the DUT: %s, While the reference out: 0x%0h",
                        sb_item.convert2string(), data_out_ref))
                    error_count ++;
                end
                else begin
                    `uvm_info("run_phase", $sformatf("Correct FIFO out: %s",
sb_item.convert2string()), UVM_HIGH)
                    correct_count++;
                end
            end
        endtask

        task ref_model(FIFO_sequence_item seq_item_chk);
            if (!seq_item_chk.rst_n) begin
                wr_ptr = 0;
                rd_ptr = 0;
                count = 0;
            end
            else begin
                if (seq_item_chk.wr_en && seq_item_chk.rd_en) begin
                    if (count == 0) begin
                        mem_ref_queue[wr_ptr] = seq_item_chk.data_in;
                        wr_ptr++;
                        count++;
                    end
                    else if (count == FIFO_DEPTH) begin
                        data_out_ref = mem_ref_queue[rd_ptr];
                        rd_ptr++;
                        count--;
                    end
                    else begin
                        data_out_ref = mem_ref_queue[rd_ptr];
                        rd_ptr++;
                        mem_ref_queue[wr_ptr] = seq_item_chk.data_in;
                        wr_ptr++;
                    end
                end
                else if (seq_item_chk.wr_en && !(count == FIFO_DEPTH)) begin
                    mem_ref_queue[wr_ptr] = seq_item_chk.data_in;
                        wr_ptr++;
                        count++;
                end
                else if (seq_item_chk.rd_en && !(count == 0)) begin
                        data_out_ref = mem_ref_queue[rd_ptr];
                        rd_ptr++;
                        count--;
                end
            end
        endtask

        function void report_phase(uvm_phase phase);
            super.report_phase(phase);
            `uvm_info("report_phase", $sformatf("Total successful transactions: %0d", 
correct_count), UVM_MEDIUM)
            `uvm_info("report_phase", $sformatf("Total failed transactions: %0d", 
error_count), UVM_MEDIUM)
        endfunction
    endclass
endpackage
```

FIFO_reset_sequence.sv

```
package FIFO_reset_sequence;
    import uvm_pkg::*;
    import FIFO_sequence_item::*;

    `include "uvm_macros.svh"

    class FIFO_reset_sequence extends  uvm_sequence#(FIFO_sequence_item);

        `uvm_object_utils(FIFO_reset_sequence)
        FIFO_sequence_item seq_item;

        function new(string name = "FIFO_reset_sequence");
            super.new(name);
        endfunction

        task body();
            seq_item = FIFO_sequence_item::type_id::create("seq_item");
            start_item(seq_item);
            seq_item.data_in = 16'hFFFF;
            seq_item.rst_n = 0;
            seq_item.wr_en = 1;
            seq_item.rd_en = 1;
            finish_item(seq_item);
        endtask
    endclass
endpackage
```

FIFO_main_sequence.sv

```
package FIFO_main_sequence;
    import uvm_pkg::*;
    import FIFO_sequence_item::*;

    `include "uvm_macros.svh"

    class FIFO_main_sequence extends  uvm_sequence#(FIFO_sequence_item);

        `uvm_object_utils(FIFO_main_sequence)
        FIFO_sequence_item seq_item;

        function new(string name = "FIFO_main_sequence");
            super.new(name);
        endfunction

        task body();
            //write
            repeat (10) begin
                seq_item=FIFO_sequence_item::type_id::create("seq_item");
                start_item(seq_item);
                seq_item.rst_n=1;
                seq_item.wr_en=1;
                seq_item.rd_en=0;
                seq_item.data_in=$random();
                finish_item(seq_item);
            end

            //read
            repeat (10) begin
                seq_item=FIFO_sequence_item::type_id::create("seq_item");
                start_item(seq_item);
                seq_item.rst_n=1;
                seq_item.wr_en=0;
                seq_item.rd_en=1;
                seq_item.data_in=$random();
                finish_item(seq_item);
            end
```



```
            //write_read
            repeat (10) begin
                seq_item=FIFO_sequence_item::type_id::create("seq_item");
                start_item(seq_item);
                seq_item.rst_n=1;
                seq_item.wr_en=1;
                seq_item.rd_en=1;
                seq_item.data_in=$random();
                finish_item(seq_item);
            end

            seq_item=FIFO_sequence_item::type_id::create("seq_item");
            start_item(seq_item);
            seq_item.rst_n=0;
            finish_item(seq_item);

            //random
            repeat (10000) begin
                seq_item=FIFO_sequence_item::type_id::create("seq_item");
                start_item(seq_item);
                assert(seq_item.randomize());
                finish_item(seq_item);
            end
        endtask
    endclass
endpackage
```

FIFO_sequencer.sv

```
package FIFO_sequencer;
    import uvm_pkg::*;
    import FIFO_sequence_item::*;

    `include "uvm_macros.svh"

class FIFO_sequencer extends  uvm_sequencer#(FIFO_sequence_item);

    `uvm_component_utils(FIFO_sequencer)

    function new(string name = "FIFO_sequencer", uvm_component parent=null);
        super.new(name, parent);
    endfunction

endclass
endpackage
```

FIFO_sequence_item.sv

```
package FIFO_sequence_item;
    import uvm_pkg::*;
    import shared_pkg::*;

    `include "uvm_macros.svh"

    class FIFO_sequence_item extends  uvm_sequence_item;

        `uvm_object_utils(FIFO_sequence_item)

        rand logic [FIFO_WIDTH-1:0] data_in;
        rand logic rst_n, wr_en, rd_en;

        logic [FIFO_WIDTH-1:0] data_out;
        logic wr_ack, overflow;
        logic full, empty, almostfull, almostempty, underflow;

        integer RD_EN_ON_DIST;
        integer WR_EN_ON_DIST;
        function new(string name = "FIFO_sequence_item", integer rd_en_on_dist = 30,
wr_en_on_dist = 70);
            super.new(name);
            RD_EN_ON_DIST = rd_en_on_dist;
            WR_EN_ON_DIST = wr_en_on_dist;
        endfunction

        function string convert2string();
            return $sformatf("%s data_in = 0x%0h, rst_n = 0b%0b, wr_en = 0b%0b, rd_en =
0b%0b, data_out = 0x%0h, wr_ack = 0b%0b, overflow = 0b%0b, full = 0b%0b, empty = 0b%0b,
almostfull = 0b%0b, almostempty = 0b%0b, underflow = 0b%0b",
                super.convert2string(), data_in, rst_n, wr_en, rd_en, data_out, wr_ack,
overflow, full, empty, almostfull, almostempty, underflow);
        endfunction

        constraint rst_n_dist {rst_n dist{1:=95, 0:=5};}
        constraint wr_en_dist {wr_en dist{1:=WR_EN_ON_DIST, 0:=100 - WR_EN_ON_DIST};}
        constraint rd_en_dist {rd_en dist{1:=RD_EN_ON_DIST, 0:=100 - RD_EN_ON_DIST};}
    endclass
endpackage
```

7) Do file

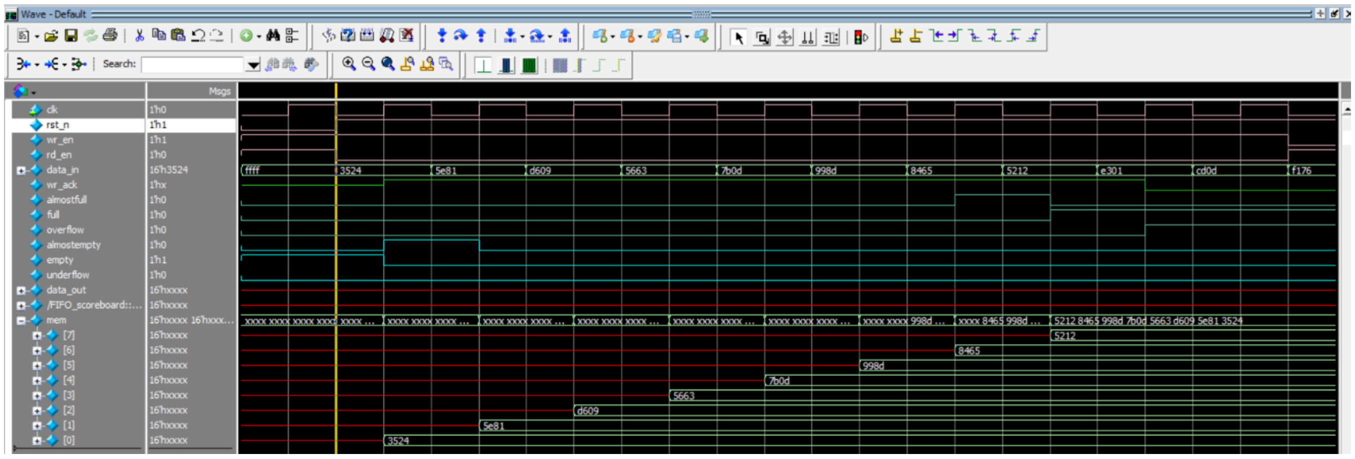
```
vlib work
vlog -f src_file.list +cover -covercells
vsim -voptargs=+acc work.FIFO_top -classdebug  -uvmcontrol=all -cover
add wave /FIFO_top/FIFO_if/*
add wave -position insertpoint  \
sim:/FIFO_scoreboard::data_out_ref
add wave -position insertpoint  \
sim:/FIFO_top/dut/mem
coverage save FIFO_tb.ucdb -onexit
run -all
```

The src_file.list:

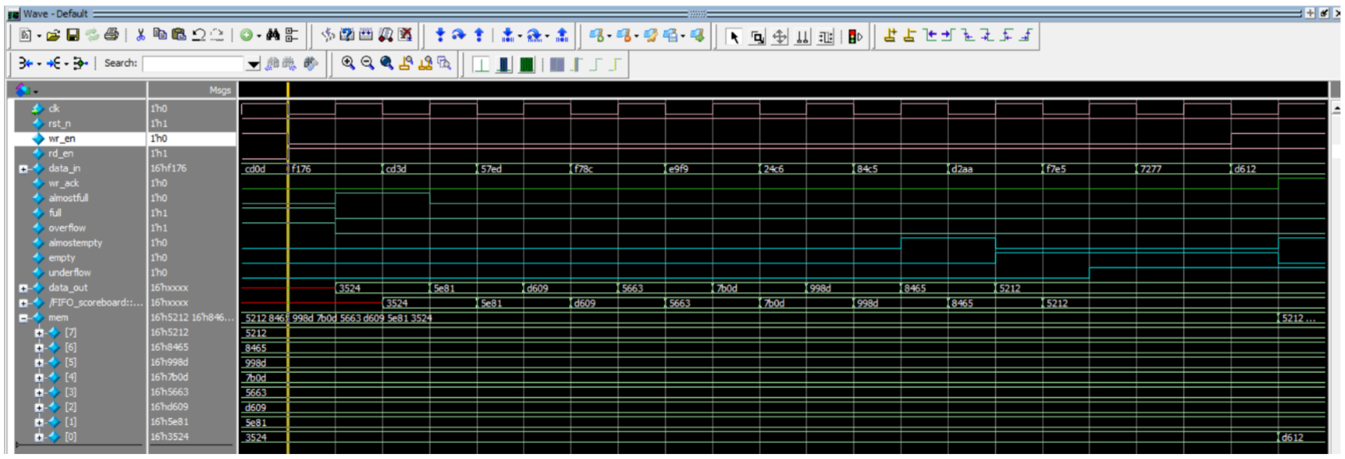
```
FIFO.sv
FIFO_sva.sv
FIFO_shared_pkg.sv
FIFO_interface.sv
FIFO_sequence_item.sv
FIFO_sequencer.sv
FIFO_main_sequence.sv
FIFO_reset_sequence.sv
FIFO_scoreboard.sv
FIFO_monitor.sv
FIFO_config.sv
FIFO_coverage.sv
FIFO_driver.sv
FIFO_agent.sv
FIFO_env.sv
FIFO_test.sv
FIFO_top.sv
```

8) Waveform

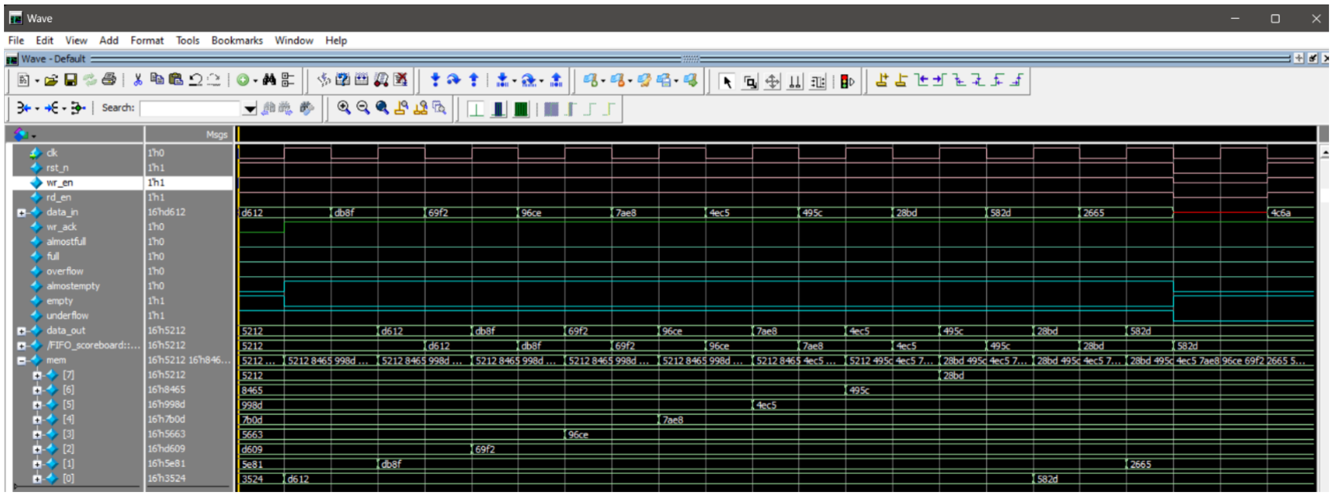
FIFO_1 + FIFO_2



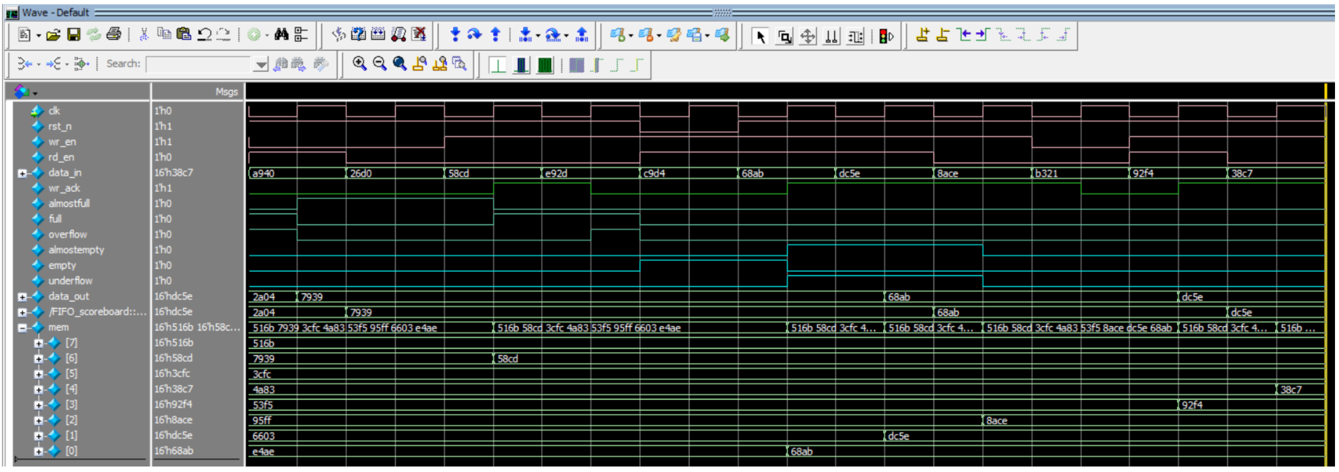
FIFO_3



FIFO_4



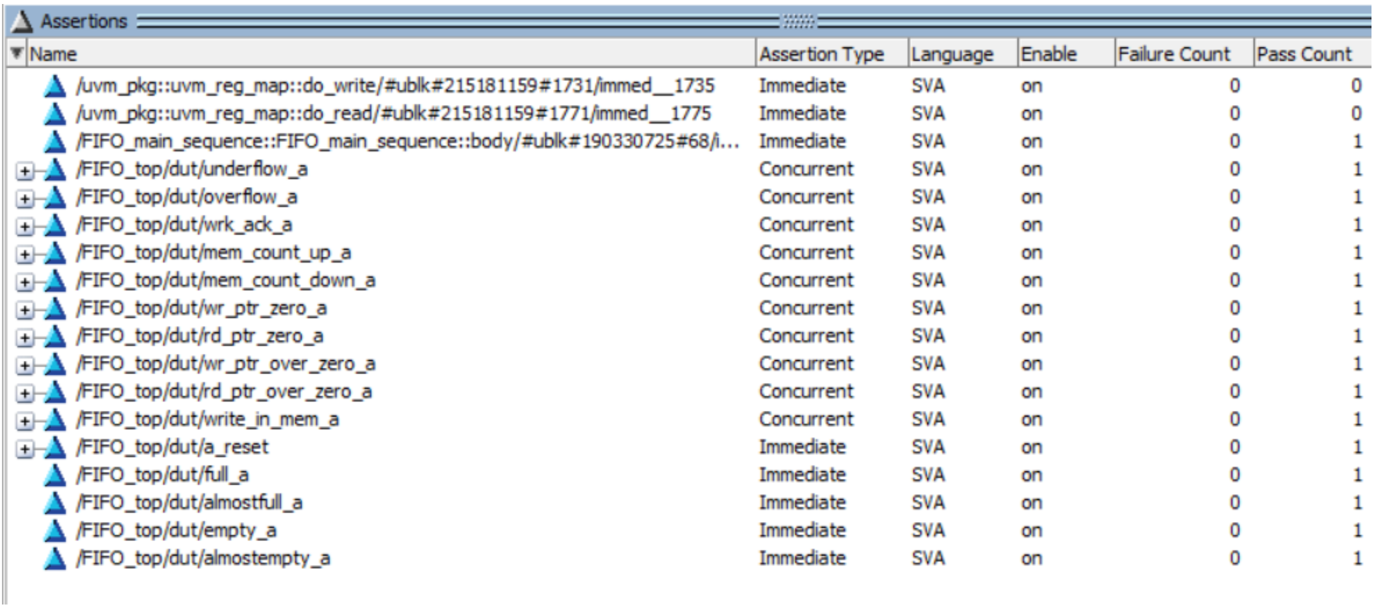
FIFO_5 + End of simulation



9) Coverage report

Assertions coverage

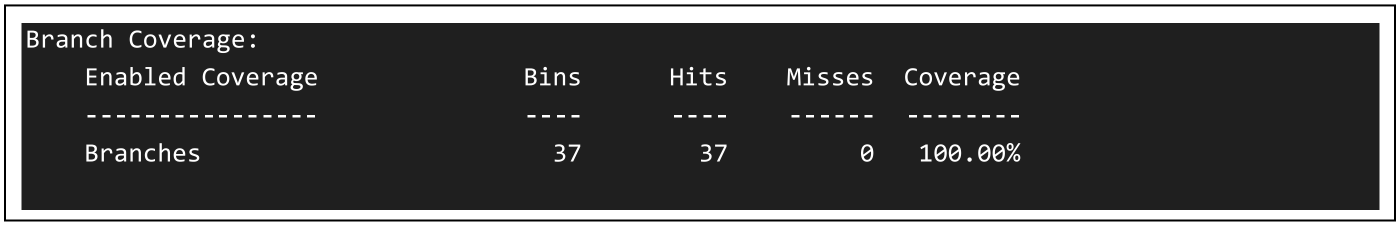
```
================================================================================
=== Instance: /FIFO_top/dut
=== Design Unit: work.FIFO
================================================================================

Assertion Coverage:
    Assertions                    15         15           0    100.00%
```



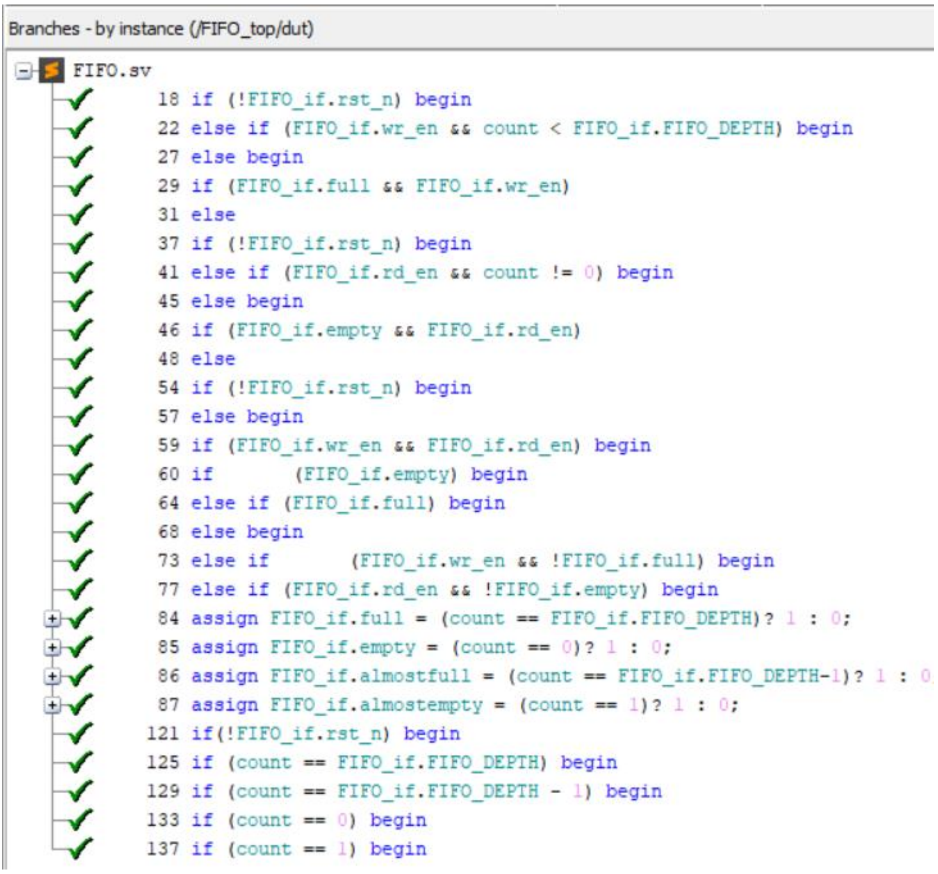
Assertions

Name	Assertion Type	Language	Enable	Failure Count	Pass Count
/uvm_pkg::uvm_reg_map::do_write/#ublk#215181159#1731/immed__1735	Immediate	SVA	on	0	0
/uvm_pkg::uvm_reg_map::do_read/#ublk#215181159#1771/immed__1775	Immediate	SVA	on	0	0
/FIFO_main_sequence::FIFO_main_sequence::body/#ublk#190330725#68/i...	Immediate	SVA	on	0	1
/FIFO_top/dut/underflow_a	Concurrent	SVA	on	0	1
/FIFO_top/dut/overflow_a	Concurrent	SVA	on	0	1
/FIFO_top/dut/wrk_ack_a	Concurrent	SVA	on	0	1
/FIFO_top/dut/mem_count_up_a	Concurrent	SVA	on	0	1
/FIFO_top/dut/mem_count_down_a	Concurrent	SVA	on	0	1
/FIFO_top/dut/wr_ptr_zero_a	Concurrent	SVA	on	0	1
/FIFO_top/dut/rd_ptr_zero_a	Concurrent	SVA	on	0	1
/FIFO_top/dut/wr_ptr_over_zero_a	Concurrent	SVA	on	0	1
/FIFO_top/dut/rd_ptr_over_zero_a	Concurrent	SVA	on	0	1
/FIFO_top/dut/write_in_mem_a	Concurrent	SVA	on	0	1
/FIFO_top/dut/a_reset	Immediate	SVA	on	0	1
/FIFO_top/dut/full_a	Immediate	SVA	on	0	1
/FIFO_top/dut/almostfull_a	Immediate	SVA	on	0	1
/FIFO_top/dut/empty_a	Immediate	SVA	on	0	1
/FIFO_top/dut/almostempty_a	Immediate	SVA	on	0	1

Branch coverage

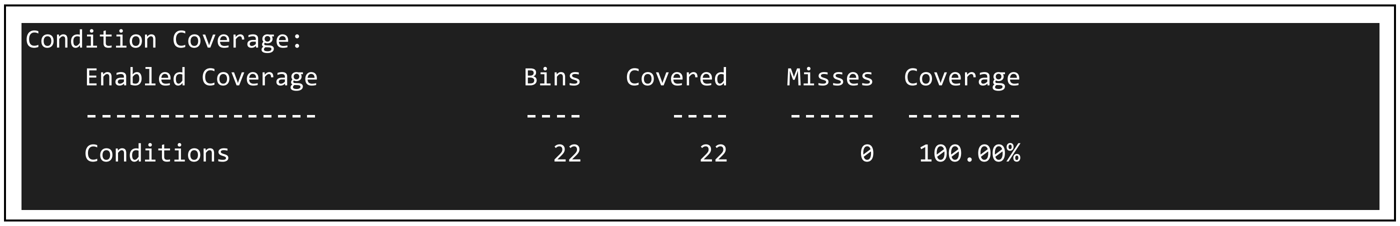


```
Branch Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                        37        37         0   100.00%
```

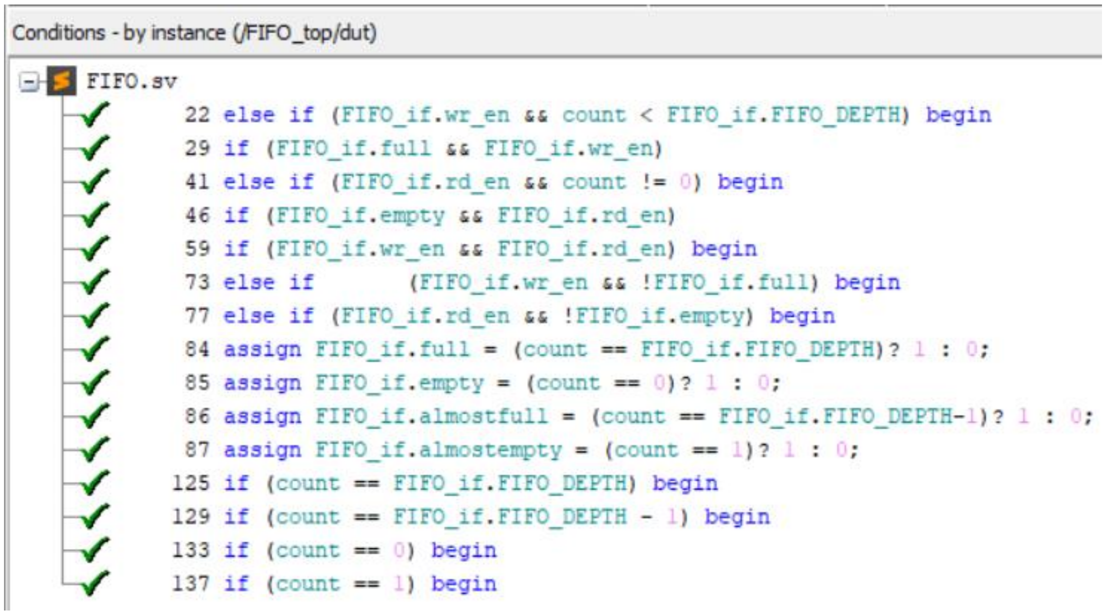


```
Branches - by instance (/FIFO_top/dut)

FIFO.sv
  ✓  18 if (!FIFO_if.rst_n) begin
  ✓  22 else if (FIFO_if.wr_en && count < FIFO_if.FIFO_DEPTH) begin
  ✓  27 else begin
  ✓  29 if (FIFO_if.full && FIFO_if.wr_en)
  ✓  31 else
  ✓  37 if (!FIFO_if.rst_n) begin
  ✓  41 else if (FIFO_if.rd_en && count != 0) begin
  ✓  45 else begin
  ✓  46 if (FIFO_if.empty && FIFO_if.rd_en)
  ✓  48 else
  ✓  54 if (!FIFO_if.rst_n) begin
  ✓  57 else begin
  ✓  59 if (FIFO_if.wr_en && FIFO_if.rd_en) begin
  ✓  60 if       (FIFO_if.empty) begin
  ✓  64 else if (FIFO_if.full) begin
  ✓  68 else begin
  ✓  73 else if       (FIFO_if.wr_en && !FIFO_if.full) begin
  ✓  77 else if (FIFO_if.rd_en && !FIFO_if.empty) begin
+ ✓  84 assign FIFO_if.full = (count == FIFO_if.FIFO_DEPTH)? 1 : 0;
+ ✓  85 assign FIFO_if.empty = (count == 0)? 1 : 0;
+ ✓  86 assign FIFO_if.almostfull = (count == FIFO_if.FIFO_DEPTH-1)? 1 : 0
+ ✓  87 assign FIFO_if.almostempty = (count == 1)? 1 : 0;
  ✓ 121 if(!FIFO_if.rst_n) begin
  ✓ 125 if (count == FIFO_if.FIFO_DEPTH) begin
  ✓ 129 if (count == FIFO_if.FIFO_DEPTH - 1) begin
  ✓ 133 if (count == 0) begin
  ✓ 137 if (count == 1) begin
```

Condition coverage



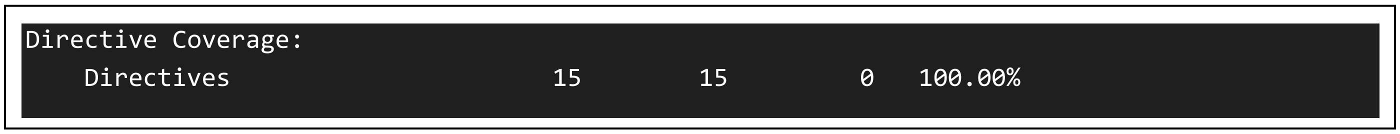
```
Condition Coverage:
    Enabled Coverage              Bins    Covered     Misses  Coverage
    ----------------              ----       ----     ------  --------
    Conditions                      22         22          0   100.00%
```



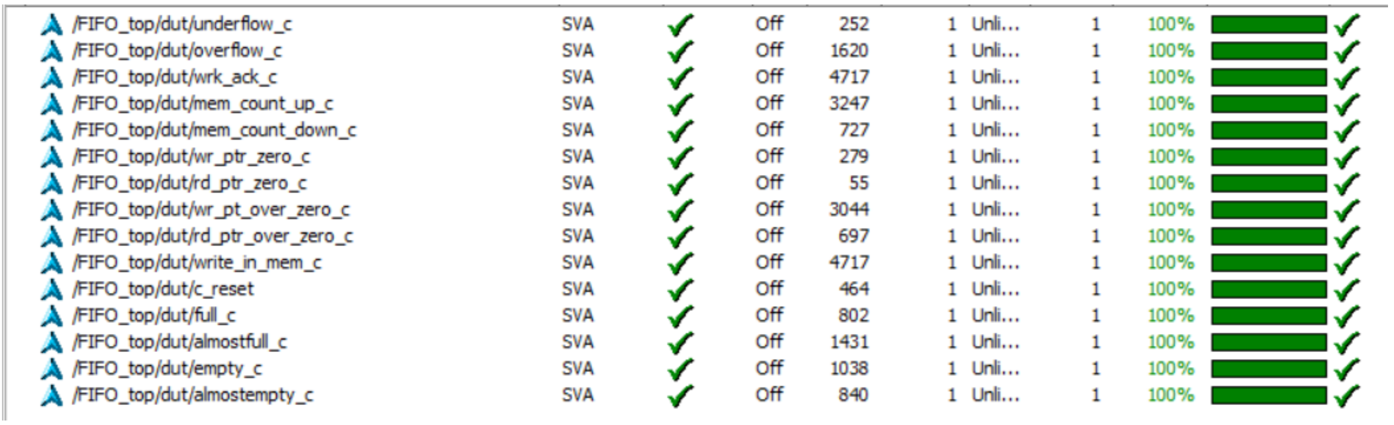
Conditions - by instance (/FIFO_top/dut)

```
FIFO.sv
✓  22 else if (FIFO_if.wr_en && count < FIFO_if.FIFO_DEPTH) begin
✓  29 if (FIFO_if.full && FIFO_if.wr_en)
✓  41 else if (FIFO_if.rd_en && count != 0) begin
✓  46 if (FIFO_if.empty && FIFO_if.rd_en)
✓  59 if (FIFO_if.wr_en && FIFO_if.rd_en) begin
✓  73 else if       (FIFO_if.wr_en && !FIFO_if.full) begin
✓  77 else if (FIFO_if.rd_en && !FIFO_if.empty) begin
✓  84 assign FIFO_if.full = (count == FIFO_if.FIFO_DEPTH)? 1 : 0;
✓  85 assign FIFO_if.empty = (count == 0)? 1 : 0;
✓  86 assign FIFO_if.almostfull = (count == FIFO_if.FIFO_DEPTH-1)? 1 : 0;
✓  87 assign FIFO_if.almostempty = (count == 1)? 1 : 0;
✓ 125 if (count == FIFO_if.FIFO_DEPTH) begin
✓ 129 if (count == FIFO_if.FIFO_DEPTH - 1) begin
✓ 133 if (count == 0) begin
✓ 137 if (count == 1) begin
```

Directive coverage

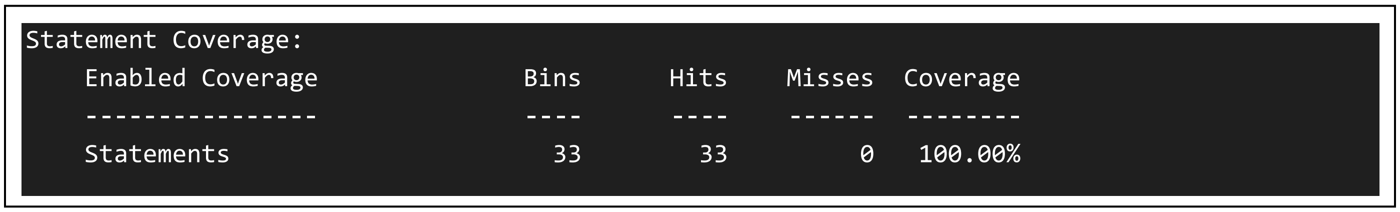


```
Directive Coverage:
    Directives                      15         15          0   100.00%
```

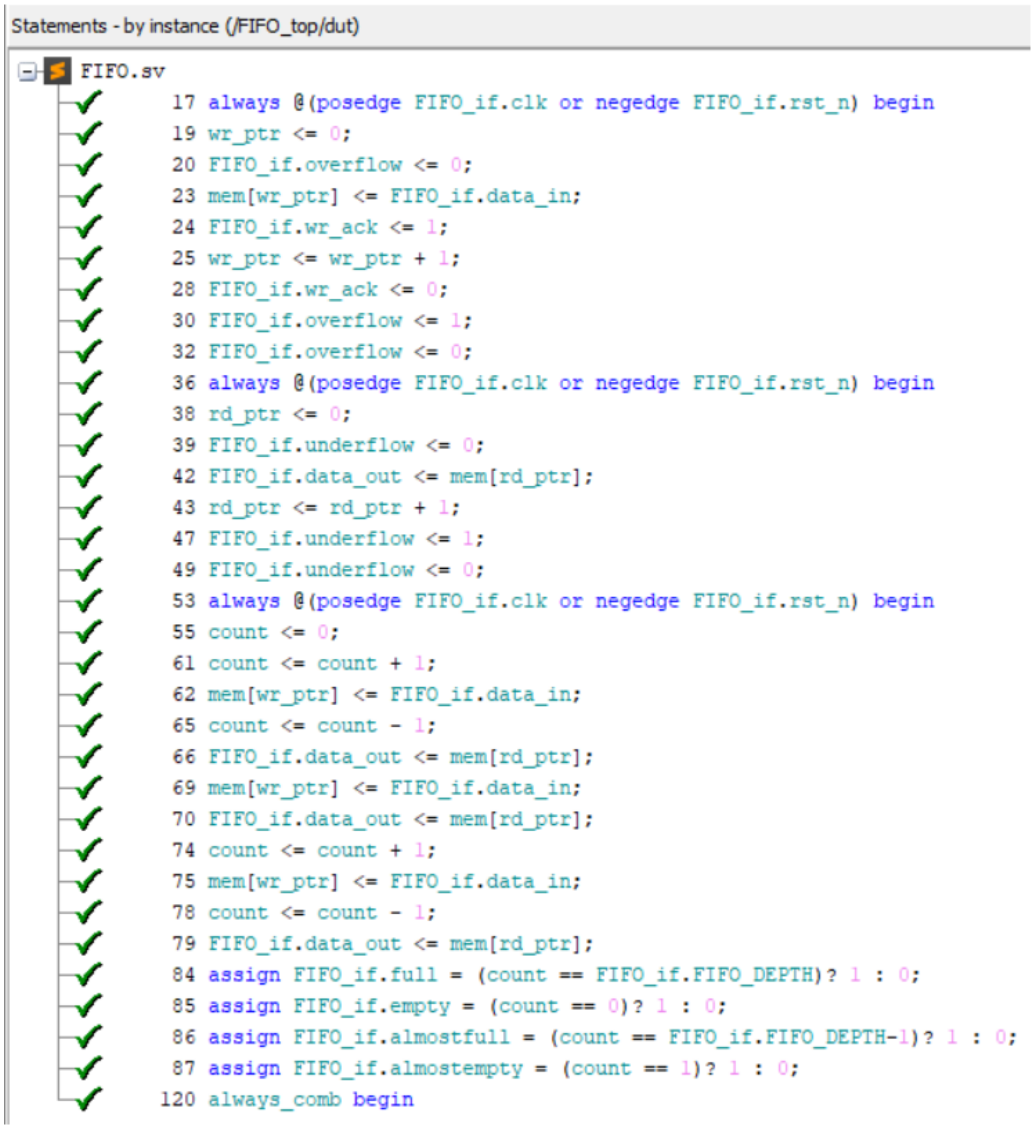


/FIFO_top/dut/underflow_c	SVA	✓	Off	252	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/overflow_c	SVA	✓	Off	1620	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/wrk_ack_c	SVA	✓	Off	4717	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/mem_count_up_c	SVA	✓	Off	3247	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/mem_count_down_c	SVA	✓	Off	727	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/wr_ptr_zero_c	SVA	✓	Off	279	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/rd_ptr_zero_c	SVA	✓	Off	55	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/wr_pt_over_zero_c	SVA	✓	Off	3044	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/rd_ptr_over_zero_c	SVA	✓	Off	697	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/write_in_mem_c	SVA	✓	Off	4717	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/c_reset	SVA	✓	Off	464	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/full_c	SVA	✓	Off	802	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/almostfull_c	SVA	✓	Off	1431	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/empty_c	SVA	✓	Off	1038	1	Unli...	1	100%	🟩	✓
/FIFO_top/dut/almostempty_c	SVA	✓	Off	840	1	Unli...	1	100%	🟩	✓

Statement coverage



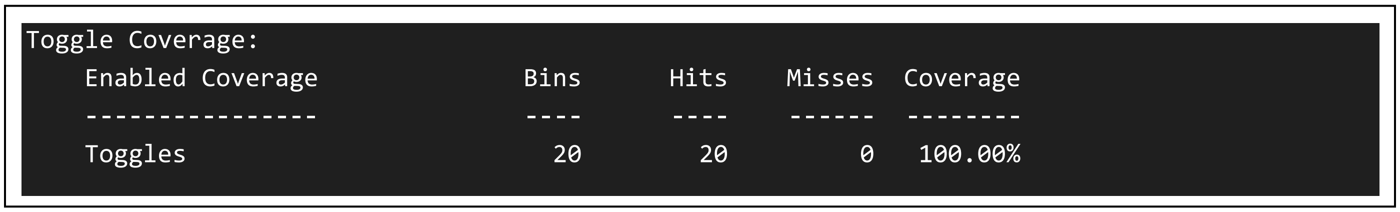
```
Statement Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Statements                      33        33         0   100.00%
```



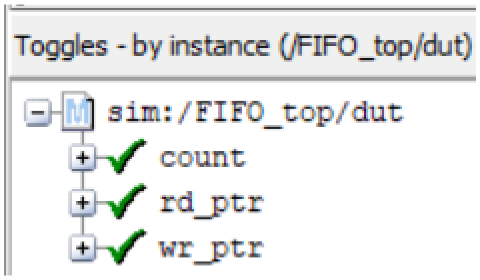
Statements - by instance (/FIFO_top/dut)

```
FIFO.sv
✓   17 always @(posedge FIFO_if.clk or negedge FIFO_if.rst_n) begin
✓   19 wr_ptr <= 0;
✓   20 FIFO_if.overflow <= 0;
✓   23 mem[wr_ptr] <= FIFO_if.data_in;
✓   24 FIFO_if.wr_ack <= 1;
✓   25 wr_ptr <= wr_ptr + 1;
✓   28 FIFO_if.wr_ack <= 0;
✓   30 FIFO_if.overflow <= 1;
✓   32 FIFO_if.overflow <= 0;
✓   36 always @(posedge FIFO_if.clk or negedge FIFO_if.rst_n) begin
✓   38 rd_ptr <= 0;
✓   39 FIFO_if.underflow <= 0;
✓   42 FIFO_if.data_out <= mem[rd_ptr];
✓   43 rd_ptr <= rd_ptr + 1;
✓   47 FIFO_if.underflow <= 1;
✓   49 FIFO_if.underflow <= 0;
✓   53 always @(posedge FIFO_if.clk or negedge FIFO_if.rst_n) begin
✓   55 count <= 0;
✓   61 count <= count + 1;
✓   62 mem[wr_ptr] <= FIFO_if.data_in;
✓   65 count <= count - 1;
✓   66 FIFO_if.data_out <= mem[rd_ptr];
✓   69 mem[wr_ptr] <= FIFO_if.data_in;
✓   70 FIFO_if.data_out <= mem[rd_ptr];
✓   74 count <= count + 1;
✓   75 mem[wr_ptr] <= FIFO_if.data_in;
✓   78 count <= count - 1;
✓   79 FIFO_if.data_out <= mem[rd_ptr];
✓   84 assign FIFO_if.full = (count == FIFO_if.FIFO_DEPTH)? 1 : 0;
✓   85 assign FIFO_if.empty = (count == 0)? 1 : 0;
✓   86 assign FIFO_if.almostfull = (count == FIFO_if.FIFO_DEPTH-1)? 1 : 0;
✓   87 assign FIFO_if.almostempty = (count == 1)? 1 : 0;
✓  120 always_comb begin
```

Toggle coverage



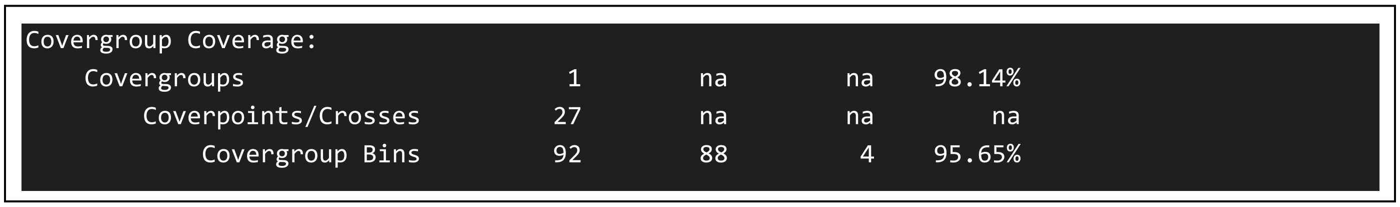
```
Toggle Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                         20        20         0   100.00%
```



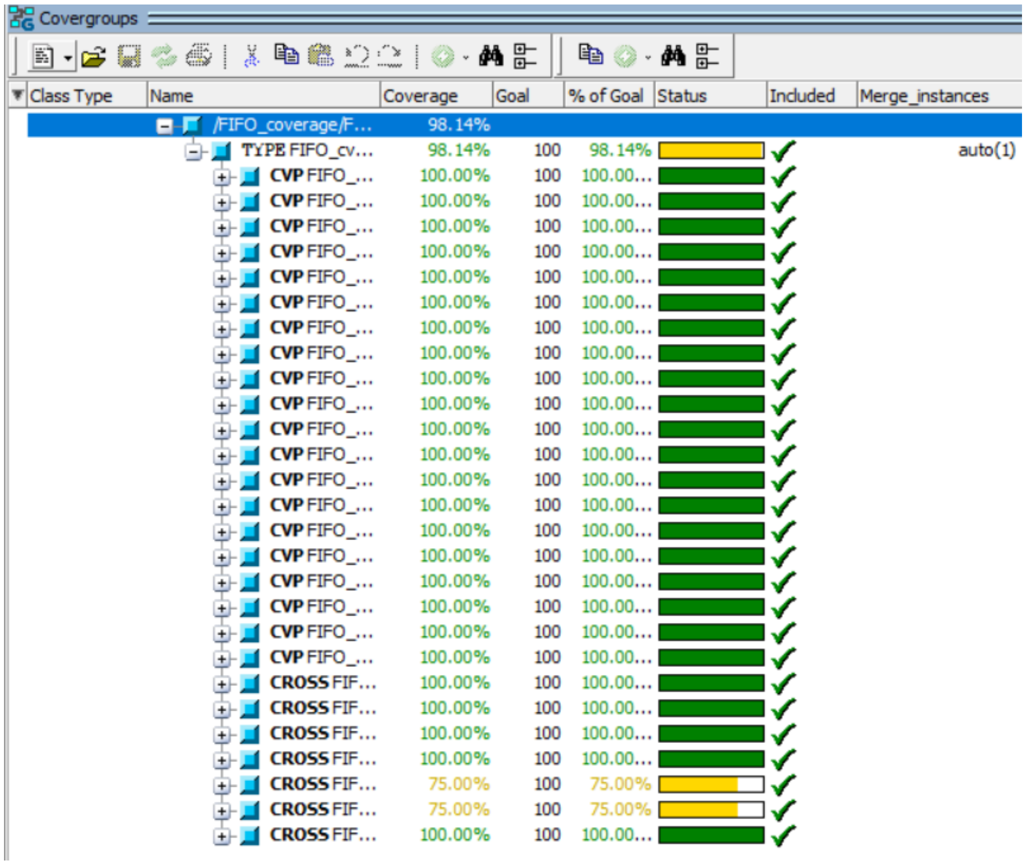
Toggles - by instance (/FIFO_top/dut)

```
sim:/FIFO_top/dut
    count
    rd_ptr
    wr_ptr
```

Covergroup coverage



```
Covergroup Coverage:
    Covergroups                      1        na        na    98.14%
        Coverpoints/Crosses         27        na        na        na
            Covergroup Bins         92        88         4    95.65%
```



Class Type	Name	Coverage	Goal	% of Goal	Status	Included	Merge_instances
	/FIFO_coverage/F...	98.14%					
	TYPE FIFO_cv...	98.14%	100	98.14%	🟨	✓	auto(1)
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CVP FIFO_...	100.00%	100	100.00...	🟩	✓	
	CROSS FIF...	100.00%	100	100.00...	🟩	✓	
	CROSS FIF...	100.00%	100	100.00...	🟩	✓	
	CROSS FIF...	100.00%	100	100.00...	🟩	✓	
	CROSS FIF...	100.00%	100	100.00...	🟩	✓	
	CROSS FIF...	75.00%	100	75.00%	🟨	✓	
	CROSS FIF...	75.00%	100	75.00%	🟨	✓	
	CROSS FIF...	100.00%	100	100.00...	🟩	✓	