

FIFO UVM Environment



Created by:

Mohamed Ahmed Mohamed Hussein

5/10/24

Verification Plan:

File	Usage			
FIFO COVERAGE COLLECTOR	The coverage collector gathers coverage information, tracking how thoroughly the DUT has been tested. It looks at functional coverage metrics like whether all possible write-read combinations, FIFO full/empty conditions, and other critical scenarios have been exercised. This component provides insight into how much of the design has been verified.			
Cover Point Name	Description	Stimulus Generation	Functional Coverage	Functionality Check
cp_wr_en	A cover point used to monitor the behavior of the wr_en signal.	Directed at the beginning of FIFO_COVERAGE		Checks if the wr_en signal transitions and how often it is asserted.
cp_rd_en	A cover point used to monitor the behavior of the rd_en signal.	Directed during FIFO_COVERAGE		Checks if the rd_en signal transitions and how often it is asserted..
cp_ack	A cover point used to observe when the wr_ack signal is generated.	Directed during FIFO_COVERAGE		Verifies that wr_ack is asserted correctly during valid write operations.
cp_overflow	A cover point used to capture occurrences of FIFO overflow.	Directed during FIFO_COVERAGE		Ensures the FIFO enters overflow when it reaches capacity and a write attempt is made.
cp_full	A cover point used to check when the FIFO is full.	Directed during FIFO_COVERAGE		Observes the correct assertion of the full signal when the FIFO reaches its maximum depth.
cp_empty	A cover point used to capture occurrences of FIFO being empty.	Directed during FIFO_COVERAGE		Validates that the FIFO empty flag is asserted when no data is present.
cp_almostfull	A cover point used to track when the FIFO is almost full.	Directed during FIFO_COVERAGE		Monitors if almostfull is asserted just before the FIFO reaches capacity.
cp_almostempty	A cover point used to track when the FIFO is almost empty.	Directed during FIFO_COVERAGE		Checks if almostempty is asserted when only one data word remains in the FIFO.
cp_underflow	A cover point used to observe underflow conditions in the FIFO	Directed during FIFO_COVERAGE		Ensures that the underflow signal is asserted when a read operation is attempted while the FIFO is empty.
Cross Coverage Name	Description	Stimulus Generation	Functional Coverage	Functionality Check
cp_wr_en_rd_en_cross	A cross coverage point used to monitor the interaction between the wr_en and rd_en signals.	Directed during FIFO_COVERAGE	Covers the simultaneous and independent assertion of wr_en and rd_en, ensuring all combinations are exercised.	Verifies how often both wr_en and rd_en are asserted simultaneously or separately, ensuring proper operation during simultaneous read and write operations.
cp_wr_en_rst_n_cross	A cross coverage point used to observe the relationship between wr_en and rst_n signals.	Directed during FIFO_COVERAGE	Covers the cases where wr_en is asserted or deasserted during both reset and non-reset states.	Ensures that no write operations (wr_en) are triggered when reset (rst_n) is asserted
cp_rd_en_rst_n_cross	A cross coverage point used to monitor how rd_en behaves when rst_n is asserted.	Directed during FIFO_COVERAGE	Covers the interaction between rd_en and rst_n, ensuring rd_en is exercised in both reset and non-reset states.	Ensures that no read operations (rd_en) occur during reset (rst_n asserted).
cp_wr_en_full_cross	A cross coverage point used to check the relationship between wr_en and full signals.	Directed during FIFO_COVERAGE	Covers scenarios where wr_en is asserted when the FIFO is full or not full.	Verifies that the wr_en signal is not asserted when the FIFO is full, ensuring no writes happen in a full FIFO.

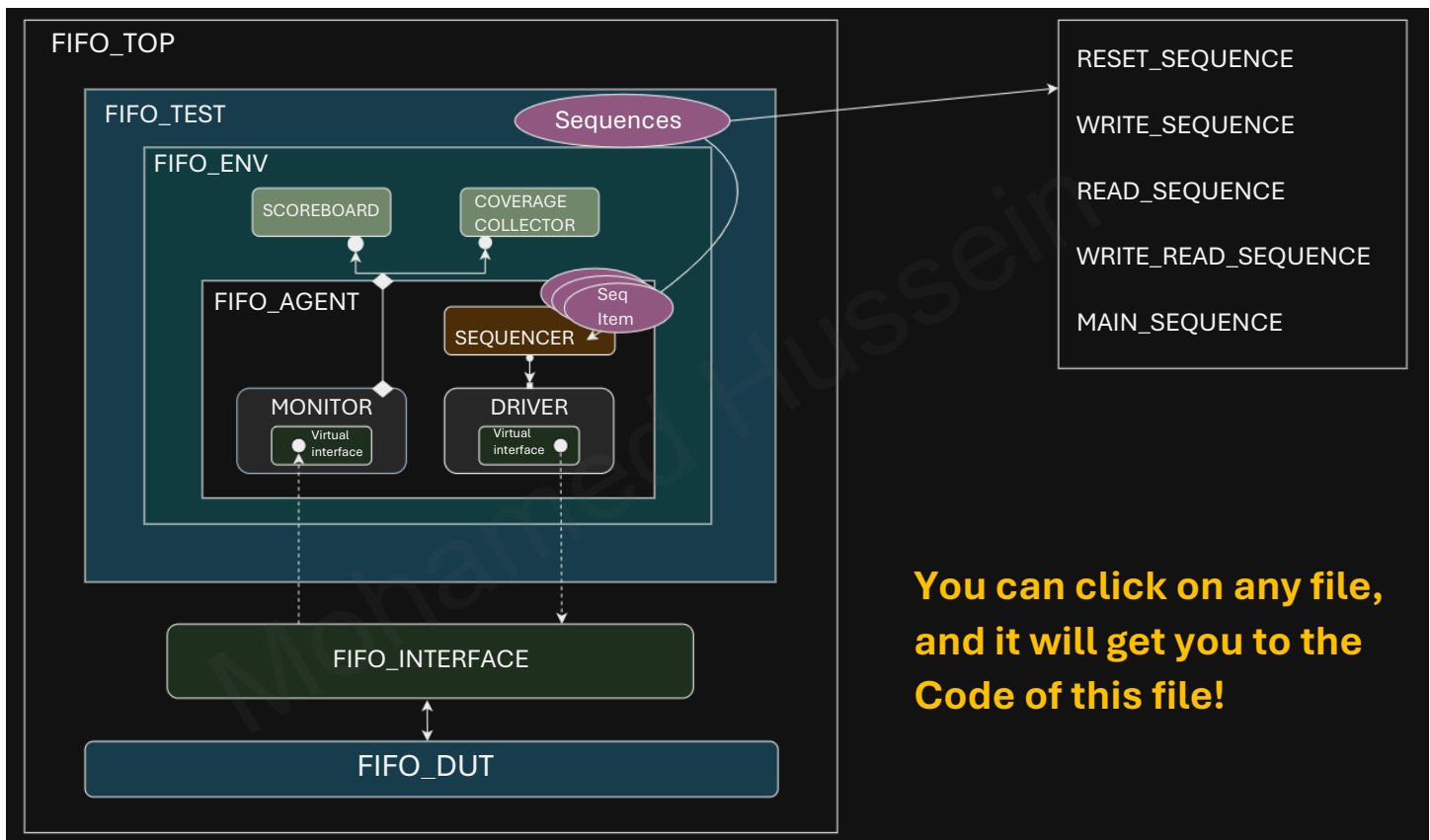
cp_rd_en_empty_cross	A cross coverage point used to track the interaction between rd_en and empty signals.	Directed during FIFO_COVERAGE	Covers the behavior of rd_en when the FIFO is empty or not empty.	Ensures that no read operations (rd_en) occur when the FIFO is empty, preventing underflow scenarios.
cp_wr_ack_wr_en_cross	A cross coverage point used to monitor the relationship between wr_ack and wr_en signals.	Directed during FIFO_COVERAGE	Covers the different combinations of wr_en and wr_ack, ensuring correct handshaking between write operations.	Verifies that wr_ack is correctly asserted when valid write operations (wr_en) are performed.
cp_rd_en_underflow_cross	A cross coverage point used to track how rd_en behaves when underflow conditions occur.	Directed during FIFO_COVERAGE	Covers scenarios where rd_en is asserted in both underflow and normal conditions.	Ensures that underflow situations are properly handled when rd_en is asserted while the FIFO is empty.
cp_full_almostfull_cross	A cross coverage point used to observe the relationship between full and almostfull signals.	Directed during FIFO_COVERAGE	Covers the transition from almostfull to full, ensuring all boundary conditions are checked.	Verifies that the almostfull signal is asserted just before full is reached, ensuring correct behavior near full capacity.
cp_full_almostfull_cross	A cross coverage point used to observe the relationship between full and almostfull signals.	Directed during FIFO_COVERAGE	Covers the transition from almostfull to full, ensuring all boundary conditions are checked.	Verifies that the almostfull signal is asserted just before full is reached, ensuring correct behavior near full capacity.
cp_empty_almostempty_cross	A cross coverage point used to monitor the transition between empty and almostempty signals.	Directed during FIFO_COVERAGE	Covers the transition between almostempty and empty, ensuring proper indication as the FIFO empties.	Ensures that almostempty is asserted when only one word remains, and empty is asserted when no data is left.
cp_overflow_wr_en_cross	A cross coverage point used to capture overflow conditions when wr_en is asserted during full conditions.	Directed at the end of FIFO_COVERAGE	Covers the scenario where wr_en is asserted when the FIFO is full, checking for correct overflow detection.	Verifies that overflow only happens when wr_en is asserted while the FIFO is full, ensuring proper handling of this scenario.

File	Usage
FIFO SCOREBOARD	The scoreboard is responsible for checking the functional correctness of the DUT as it has the reference model. It compares the expected results (predicted by a reference model or logic) with the actual outputs observed by the monitor. In your project, the FIFO_SCOREBOARD will verify that the data written into the FIFO is correctly read out and that any underflow or overflow conditions are handled properly.

File	Usage			
FIFO SVA	The assertions module (FIFO_SVA.sv) defines SystemVerilog assertions (SVA) to check critical design properties. For example, assertions may ensure that the FIFO never overflows or underflows and that data integrity is maintained. These are runtime checks that trigger if the DUT violates any of the expected behaviors.			
Assertion Name	Description	Stimulus Generation	Functional Coverage	Functionality Check
reset assertion	An immediate assertion used to verify that signals are correctly reset when rst_n is deasserted.	Directed at the beginning of the design code (FIFO)		Ensures that all internal signals such as count, wr_ptr, rd_ptr, full, empty, etc., are correctly reset.
full_a	An assertion used to check that the full signal is asserted when the FIFO depth is reached.	Directed during the design code (FIFO)		Validates that the FIFO full flag is asserted when the FIFO reaches its maximum capacity.
empty_a	An assertion used to verify that the FIFO's empty signal is asserted when no data is present.	Directed during the design code (FIFO)		Confirms that the FIFO is indeed empty when the empty flag is high.
almostfull_a	An assertion used to check that the almostfull signal is correctly asserted when the FIFO is nearly full.	Directed during the design code (FIFO)		Verifies that the FIFO is nearing capacity when almostfull is asserted.
almostempty_a	An assertion used to check that the almostempty signal is asserted when only one data word is left in the FIFO.	Directed during the design code (FIFO)		Ensures that the FIFO is nearly empty when almostempty is correctly asserted.
ack_a	An assertion used to ensure that wr_ack is generated only when a write operation is successful.	Directed during the design code (FIFO)		Verifies that wr_ack is asserted during valid write operations.
overflow_a	An assertion used to detect when the FIFO overflows upon attempting a write operation while full.	Directed during the design code (FIFO)		Ensures that the FIFO enters overflow only when the FIFO is full and a write attempt is made.
underflow_a	An assertion used to catch underflow situations when attempting a read operation on an empty FIFO.	Directed during the design code (FIFO)		Validates that the FIFO enters underflow only when a read operation is attempted while the FIFO is empty.
wr_ptr_a	An assertion used to track the wr_ptr increment after valid write operations.	Directed during the design code (FIFO)		Ensures that the write pointer increments correctly with every valid write operation.
rd_ptr_a	An assertion used to track the rd_ptr increment after valid read operations.	Directed during the design code (FIFO)		Ensures that the read pointer increments correctly with every valid read operation.
count_write_priority_a	An assertion used to verify the counter behavior when write and read enable signals are both active with empty.	Directed during the design code (FIFO)		Confirms that the counter increments as expected in this priority case.
count_read_priority_a	An assertion used to verify the counter behavior when both write and read are enabled with the FIFO full.	Directed during the design code (FIFO)		Confirms that the counter decrements as expected in this priority case.

count_w_a	An assertion used to verify that the counter increments with valid writes and no read operations.	Directed during the design code (FIFO)		Ensures that the counter increments correctly when a valid write occurs and no read is happening.
count_r_a	An assertion used to verify that the counter decrements with valid reads and no write operations.	Directed during the design code (FIFO)		Ensures that the counter decrements correctly when a valid read occurs and no write is happening.
File	Usage			
FIFO SEQUENCE ITEM	This file has the constraints and defines the transaction object (sequence item) that carries the data and control signals for write and read operations to/from the FIFO. It abstracts the data transfer operations.			
Constraint Name	Description	Stimulus Generation	Functional Coverage	Functionality Check
reset_con	A constraint used to control the distribution of rst_n, making reset occur less frequently.	Directed at the end of FIFO_TRANSACTION		Ensures that rst_n is mostly high with rare assertions, simulating rare reset conditions during test runs.
wr_en_con	A constraint used to define the probability of wr_en being asserted based on WR_EN_ON_DIST.	Directed at the end of FIFO_TRANSACTION		Verifies that the wr_en signal follows the expected distribution, ensuring write operations occur at the correct frequency.
rd_en_con	A constraint used to define the probability of rd_en being asserted based on RD_EN_ON_DIST.	Directed at the end of FIFO_TRANSACTION		Verifies that the rd_en signal follows the expected distribution, ensuring read operations occur at the correct frequency.

FIFO UVM Structure:



How it Works?

1. Top Module (FIFO_TOP.sv):

This is the highest level of your verification environment. It instantiates the design under test (DUT) and connects the DUT to the UVM testbench. The FIFO_TOP.sv module also instantiates the FIFO_IF (interface) and links it to the DUT, ensuring signals between the testbench and the DUT are connected correctly.

2. Interface (FIFO_IF.sv):

The FIFO_IF.sv defines the signals that connect to the FIFO DUT. This includes signals like clock, reset, write enable, read enable, data inputs, and outputs. The interface provides a structured mechanism to communicate with the DUT by grouping related signals together and making them accessible to the driver, monitor, and scoreboard.

3. UVM Testbench Flow:

The UVM testbench drives the verification process by generating stimulus, driving it into the DUT, monitoring the outputs, and analyzing the results.

3.1 Configuration (FIFO_CONFIG.sv)

The configuration object holds parameters like FIFO depth, data width, and other operational settings. This configuration is passed to various components in the testbench to ensure a consistent environment for verification.

3.2 Sequences and Sequence Items

- **FIFO_SEQUENCE_ITEM.sv:**
This file defines the transaction object (sequence item) that carries the data and control signals for write and read operations to/from the FIFO. It abstracts the data transfer operations.
- **FIFO_RESET_SEQUENCE.sv, FIFO_WRITE_SEQUENCE.sv, FIFO_READ_SEQUENCE.sv, FIFO_WRITE_READ_SEQUENCE.sv, FIFO_MAIN_SEQUENCE.sv:**
These files define different sequences, which are collections of transactions. Each sequence models a specific operation: resetting the FIFO, writing data, reading data, performing combined write-read operations, and orchestrating the overall verification flow (in the main sequence). The main sequence controls the order of operations and ensures different sequences are executed as part of the verification plan.

3.3 Sequencer (FIFO_SEQUENCER.sv)

The sequencer coordinates the execution of sequences. It sends the transaction (sequence item) to the driver, which will then apply it to the DUT. In your UVM testbench, the FIFO_SEQUENCER manages the flow of read and write sequences and other operations, ensuring the correct stimulus is generated.

3.4 Driver (FIFO_DRIVER.sv)

The driver takes the transaction objects from the sequencer and translates them into pin-level activity on the DUT using the interface. It drives the stimulus into the DUT through the FIFO_IF, such as asserting write or read enable signals, and providing the data input to the FIFO.

3.5 Monitor (FIFO_MONITOR.sv)

The monitor observes the DUT's signals through the interface. It passively captures the inputs and outputs of the FIFO for analysis without modifying the DUT's behavior. The monitor collects relevant data and forwards it to other components like the coverage collector and scoreboard.

4. Analysis Components

4.1 Scoreboard (FIFO_SCOREBOARD.sv)

The scoreboard is responsible for checking the functional correctness of the DUT. It compares the expected results (predicted by a reference model or logic) with the actual outputs observed by the monitor. In your project, the FIFO_SCOREBOARD will verify that the data written into the FIFO is correctly read out and that any underflow or overflow conditions are handled properly.

4.2 Coverage Collector (FIFO_COVERAGE_COLLECTOR.sv)

The coverage collector gathers coverage information, tracking how thoroughly the DUT has been tested. It looks at functional coverage metrics like whether all possible write-read combinations, FIFO full/empty conditions, and other critical scenarios have been exercised. This component provides insight into how much of the design has been verified.

5. Environment (FIFO_ENV.sv)

The environment is the container for all the UVM components, such as agents, scoreboard, and coverage collector. The FIFO_ENV instantiates the agents that control the driver, monitor, and sequencer and ensures that these components interact correctly. It also connects the environment's agents to the scoreboard and coverage collector.

- **FIFO_AGENT.sv:**

The agent encapsulates the driver, monitor, and sequencer for the FIFO. It coordinates their activities to ensure the interface is driven correctly and monitored consistently. The agent simplifies managing these components by providing a unified entity to control them.

6. Assertions (FIFO_SVA.sv)

The assertions module (FIFO_SVA.sv) defines SystemVerilog assertions (SVA) to check critical design properties. For example, assertions may ensure that the FIFO never overflows or underflows and that data integrity is maintained. These are runtime checks that trigger if the DUT violates any of the expected behaviors.

7. Test (FIFO_TEST.sv)

The test file (FIFO_TEST.sv) is where the overall test strategy is defined. This file extends the UVM test class and initiates the execution of the sequences. It configures the environment, runs the main sequence, and ensures the test runs as expected. You may have multiple tests targeting different aspects of the FIFO (e.g., stress tests, boundary condition tests, etc.).

Bugs Report:

- Underflow signal bug:

Bug: underflow signal was meant to be sequential (following clock edge) in the Specs

Original design:

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

Modification: I made it following clock edge in the read always block

Modified design:

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        rd_ptr <= 0;
        underflow <= 0; // was added
    end
    else if (rd_en && count != 0) begin
        data_out <= mem[rd_ptr];
        rd_ptr <= rd_ptr + 1;
        underflow <= 0; // was added
    end
    else begin
        if(empty && rd_en) // this is sequential output not combinational
            underflow = 1;
        else
            underflow = 0;
    end
end
```

- **Missing conditions:**

Bug: the Spec was “If a read and write enables were high and the FIFO was empty, only writing will take place and vice verse if the FIFO was full” and it was not implemented in the original design

Original design:

```
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
```

Modification: I added the right conditions to meet the Specs

Modified design:

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        count <= 0;
    end
    else begin
        if (wr_en && rd_en && empty) // this condition was added
            count <= count + 1;
        else if (wr_en && rd_en && full) // this condition was added
            count <= count - 1;
        else 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
```

- **Unknown behavior while reset:**

Bug: there are some signals which their output haven't been specified when reset signal takes place

Original design:

```
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

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
```

Modification: I added the right conditions to meet the Specs

Modified design:

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

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        rd_ptr <= 0;
        underflow <= 0; // was added
    end
    else if (rd_en && count != 0) begin
        data_out <= mem[rd_ptr];
        rd_ptr <= rd_ptr + 1;
        underflow <= 0; // was added
    end
    else begin
        if(empty && rd_en) // this is sequential output not combinational
            underflow = 1;
        else
            underflow = 0;
    end
end
```

Code Coverage:

- Branch:

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

FIFO.sv
47 if(!rst_n) begin
59 if((count == FIFO_DEPTH))
61 if((count == 0))
65 if((count == FIFO_DEPTH - 1))
67 if((count == 1))
131 if (!rst_n) begin
136 else if (wr_en && count < FIFO_DEPTH) begin
142 else begin
144 if (full & wr_en)
146 else
152 if (!rst_n) begin
156 else if (rd_en && count != 0) begin
161 else begin
162 if(empty && rd_en) // this is sequential output no combinational
164 else
170 if (!rst_n) begin
173 else begin
174 if (wr_en && rd_en && empty) // this condition was added
176 else if (wr_en && rd_en && full) // this condition was added
178 else if ( ({wr_en, rd_en} == 2'b10) && !full)
180 else if ( ({wr_en, rd_en} == 2'b01) && !empty)
185 assign full = (count == FIFO_DEPTH)? 1 : 0;
186 assign empty = (count == 0)? 1 : 0;
188 assign almostfull = (count == FIFO_DEPTH-1)? 1 : 0;
189 assign almostempty = (count == 1)? 1 : 0;
```

- Toggle:

```
Toggles - by instance (/fifo_top/tb)

sim:/fifo_top/tb
almostempty
almostfull
clk
data_in
data_out
empty
full
overflow
rd_en
rst_n
underflow
wr_ack
wr_en
```

- Statement:

FIFO.sv





































```
✓ 20 assign clk = fifoif.clk;
✓ 21 assign data_in = fifoif.data_in;
✓ 22 assign rst_n = fifoif.rst_n;
✓ 23 assign wr_en = fifoif.wr_en;
✓ 24 assign rd_en = fifoif.rd_en;
✓ 46 always_comb begin // for asynchronous reset
✓ 130 always @(posedge clk or negedge rst_n) begin
✓ 132 wr_ptr <= 0;
✓ 133 wr_ack <= 0; // was added
✓ 134 overflow <= 0; // was added
✓ 137 mem[wr_ptr] <= data_in;
✓ 138 wr_ack <= 1;
✓ 139 wr_ptr <= wr_ptr + 1;
✓ 140 overflow <= 0; // was added
✓ 143 wr_ack <= 0;
✓ 145 overflow <= 1;
✓ 147 overflow <= 0;
✓ 151 always @(posedge clk or negedge rst_n) begin
✓ 153 rd_ptr <= 0;
✓ 154 underflow <= 0; // was added
✓ 157 data_out <= mem[rd_ptr];
✓ 158 rd_ptr <= rd_ptr + 1;
✓ 159 underflow <= 0; // was added
✓ 163 underflow = 1;
✓ 165 underflow = 0;
✓ 169 always @(posedge clk or negedge rst_n) begin
✓ 171 count <= 0;
✓ 171 count <= 0;
✓ 175 count <= count + 1;
✓ 177 count <= count - 1;
✓ 179 count <= count + 1;
✓ 181 count <= count - 1;
✓ 185 assign full = (count == FIFO_DEPTH)? 1 : 0;
✓ 186 assign empty = (count == 0)? 1 : 0;
✓ 188 assign almostfull = (count == FIFO_DEPTH-1)? 1 : 0;
✓ 189 assign almostempty = (count == 1)? 1 : 0;
```


Functional Coverage (including cross coverage):

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
/fifo_coverage_pkg/FIFO_coverage		100.0%				
TYPE FIFO_Cross_Group	FIFO_cover...	100.0%	100	100.0%		✓
CVP FIFO_Cross_Group::cp_wr_en	FIFO_cover...	100.0%	100	100.0%		✓
bin auto[0]		286	1	100.0%		✓
bin auto[1]		715	1	100.0%		✓
CVP FIFO_Cross_Group::cp_rd_en	FIFO_cover...	100.0%	100	100.0%		✓
bin auto[0]		721	1	100.0%		✓
bin auto[1]		280	1	100.0%		✓
CVP FIFO_Cross_Group::cp_ack	FIFO_cover...	100.0%	100	100.0%		✓
bin auto[0]		649	1	100.0%		✓
bin auto[1]		352	1	100.0%		✓
CVP FIFO_Cross_Group::cp_overflow	FIFO_cover...	100.0%	100	100.0%		✓
bin auto[0]		647	1	100.0%		✓
bin auto[1]		354	1	100.0%		✓
CVP FIFO_Cross_Group::cp_full	FIFO_cover...	100.0%	100	100.0%		✓
bin auto[0]		503	1	100.0%		✓
bin auto[1]		498	1	100.0%		✓
CVP FIFO_Cross_Group::cp_empty	FIFO_cover...	100.0%	100	100.0%		✓
bin auto[0]		979	1	100.0%		✓
bin auto[1]		22	1	100.0%		✓
CVP FIFO_Cross_Group::cp_almostfull	FIFO_cover...	100.0%	100	100.0%		✓
bin auto[0]		732	1	100.0%		✓
bin auto[1]		269	1	100.0%		✓
CVP FIFO_Cross_Group::cp_almostempty	FIFO_cover...	100.0%	100	100.0%		✓
bin auto[0]		965	1	100.0%		✓
bin auto[1]		36	1	100.0%		✓
CVP FIFO_Cross_Group::cp_underflow	FIFO_cover...	100.0%	100	100.0%		✓
bin auto[0]		993	1	100.0%		✓
bin auto[1]		8	1	100.0%		✓

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
CVP FIFO_Cross_Group::cp_underflow	FIFO_cover...	100.0%	100	100.0%		✓
CROSS FIFO_Cross_Group::wr_ack_C	FIFO_cover...	100.0%	100	100.0%		✓
bin <auto[0],auto[0],auto[0]>		192	1	100.0%		✓
bin <auto[1],auto[0],auto[0]>		287	1	100.0%		✓
bin <auto[0],auto[1],auto[0]>		94	1	100.0%		✓
bin <auto[1],auto[1],auto[0]>		76	1	100.0%		✓
bin <auto[1],auto[0],auto[1]>		242	1	100.0%		✓
bin <auto[1],auto[1],auto[1]>		110	1	100.0%		✓
illegal_bin zero_zero_one		0	-	-		✓
CROSS FIFO_Cross_Group::overflow_C	FIFO_cover...	100.0%	100	100.0%		✓
bin <auto[0],auto[0],auto[0]>		192	1	100.0%		✓
bin <auto[1],auto[0],auto[0]>		249	1	100.0%		✓
bin <auto[0],auto[1],auto[0]>		94	1	100.0%		✓
bin <auto[1],auto[1],auto[0]>		112	1	100.0%		✓
bin <auto[1],auto[0],auto[1]>		280	1	100.0%		✓
bin <auto[1],auto[1],auto[1]>		74	1	100.0%		✓
illegal_bin zero_w_one		0	-	-		✓
CROSS FIFO_Cross_Group::full_C	FIFO_cover...	100.0%	100	100.0%		✓
bin <auto[0],auto[0],auto[0]>		100	1	100.0%		✓
bin <auto[0],auto[1],auto[0]>		94	1	100.0%		✓
bin <auto[1],auto[0],auto[0]>		123	1	100.0%		✓
bin <auto[1],auto[1],auto[0]>		186	1	100.0%		✓
bin <auto[0],auto[0],auto[1]>		92	1	100.0%		✓
bin <auto[1],auto[0],auto[1]>		406	1	100.0%		✓
illegal_bin one_r_one		0	-	-		✓
CROSS FIFO_Cross_Group::empty_C	FIFO_cover...	100.0%	100	100.0%		✓
bin <auto[0],auto[0],auto[0]>		186	1	100.0%		✓
bin <auto[1],auto[0],auto[0]>		522	1	100.0%		✓
bin <auto[0],auto[1],auto[0]>		87	1	100.0%		✓
bin <auto[1],auto[1],auto[0]>		184	1	100.0%		✓
bin <auto[0],auto[0],auto[1]>		6	1	100.0%		✓
bin <auto[1],auto[0],auto[1]>		7	1	100.0%		✓
bin <auto[0],auto[1],auto[1]>		7	1	100.0%		✓
bin <auto[1],auto[1],auto[1]>		2	1	100.0%		✓
CROSS FIFO_Cross_Group::almostfull_C	FIFO_cover...	100.0%	100	100.0%		✓
bin <auto[0],auto[0],auto[0]>		129	1	100.0%		✓
bin <auto[1],auto[0],auto[0]>		499	1	100.0%		✓
bin <auto[0],auto[1],auto[0]>		48	1	100.0%		✓
bin <auto[1],auto[1],auto[0]>		56	1	100.0%		✓
bin <auto[0],auto[0],auto[1]>		63	1	100.0%		✓
bin <auto[1],auto[0],auto[1]>		30	1	100.0%		✓
bin <auto[0],auto[1],auto[1]>		46	1	100.0%		✓
bin <auto[1],auto[1],auto[1]>		130	1	100.0%		✓
CROSS FIFO_Cross_Group::almostempty_C	FIFO_cover...	100.0%	100	100.0%		✓
bin <auto[0],auto[0],auto[0]>		184	1	100.0%		✓
bin <auto[1],auto[0],auto[0]>		519	1	100.0%		✓
bin <auto[0],auto[1],auto[0]>		90	1	100.0%		✓
bin <auto[1],auto[1],auto[0]>		172	1	100.0%		✓
bin <auto[0],auto[0],auto[1]>		8	1	100.0%		✓
bin <auto[1],auto[0],auto[1]>		10	1	100.0%		✓
bin <auto[0],auto[1],auto[1]>		4	1	100.0%		✓
bin <auto[1],auto[1],auto[1]>		14	1	100.0%		✓
CROSS FIFO_Cross_Group::underflow_C	FIFO_cover...	100.0%	100	100.0%		✓
bin <auto[0],auto[0],auto[0]>		192	1	100.0%		✓
bin <auto[0],auto[1],auto[0]>		91	1	100.0%		✓
bin <auto[1],auto[0],auto[0]>		529	1	100.0%		✓
bin <auto[1],auto[1],auto[0]>		181	1	100.0%		✓
bin <auto[0],auto[1],auto[1]>		3	1	100.0%		✓
bin <auto[1],auto[1],auto[1]>		5	1	100.0%		✓
illegal_bin zero_r_one		0	-	-		✓

Assertions Coverage:

Name	Language	Enabled	Log	Count	AtLeast	Limit	Weight	Cmplt %	Cmplt graph	Included
 /fifo_top/dut/ack_c	SVA		Off	349	1	Unlimited	1	100%		
 /fifo_top/dut/overflow_c	SVA		Off	349	1	Unlimited	1	100%		
 /fifo_top/dut/underflow_c	SVA		Off	8	1	Unlimited	1	100%		
 /fifo_top/dut/wr_ptr_c	SVA		Off	349	1	Unlimited	1	100%		
 /fifo_top/dut/rd_ptr_c	SVA		Off	267	1	Unlimited	1	100%		
 /fifo_top/dut/count_write_priority_c	SVA		Off	5	1	Unlimited	1	100%		
 /fifo_top/dut/count_read_priority_c	SVA		Off	73	1	Unlimited	1	100%		
 /fifo_top/dut/count_w_c	SVA		Off	240	1	Unlimited	1	100%		
 /fifo_top/dut/count_r_c	SVA		Off	90	1	Unlimited	1	100%		

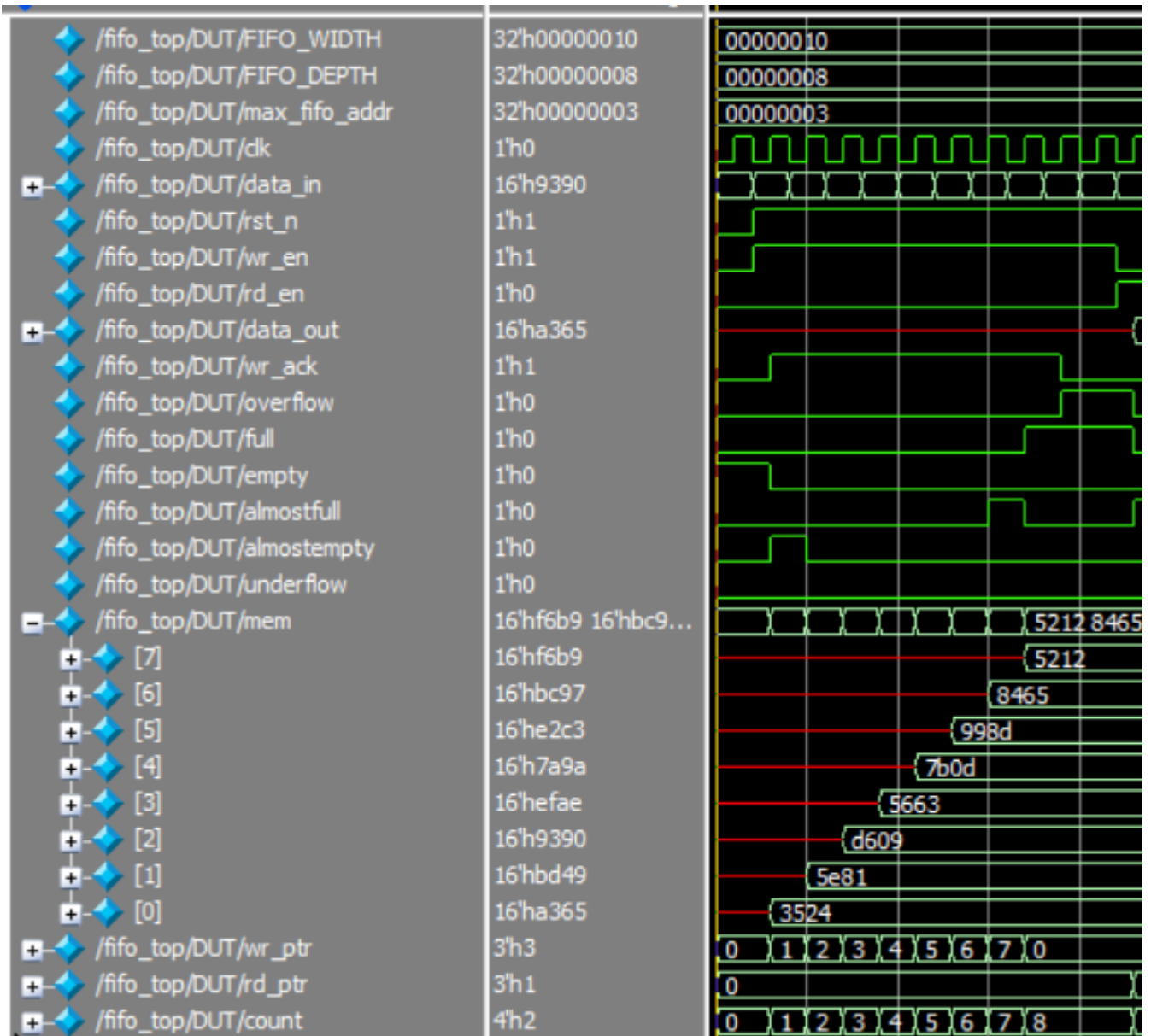
Assertions Passed:

Name	Assertion Type	Language	Enable	Failure Count	Pass Count	Active Count	Memory	Peak	CPU	ATV	Assertion Expression	Ind
+/fifo_top/dut/count_aa	Immediate	SVA	on	0	1	-	-	-	-	off	assert (count==0)	✓
+/fifo_top/dut/wr_ptr_aa	Immediate	SVA	on	0	1	-	-	-	-	off	assert (wr_ptr==0)	✓
+/fifo_top/dut/rd_ptr_aa	Immediate	SVA	on	0	1	-	-	-	-	off	assert (rd_ptr==0)	✓
+/fifo_top/dut/full_aa	Immediate	SVA	on	0	1	-	-	-	-	off	assert (full==0)	✓
+/fifo_top/dut/empty_aa	Immediate	SVA	on	0	1	-	-	-	-	off	assert (empty==1)	✓
+/fifo_top/dut/underflow_aa	Immediate	SVA	on	0	1	-	-	-	-	off	assert (underflow==0)	✓
+/fifo_top/dut/almostfull_aa	Immediate	SVA	on	0	1	-	-	-	-	off	assert (almostfull==0)	✓
+/fifo_top/dut/almostempty_aa	Immediate	SVA	on	0	1	-	-	-	-	off	assert (almostempty==0)	✓
+/fifo_top/dut/wr_ack_aa	Immediate	SVA	on	0	1	-	-	-	-	off	assert (wr_ack==0)	✓
+/fifo_top/dut/overflow_aa	Immediate	SVA	on	0	1	-	-	-	-	off	assert (overflow==0)	✓
+/fifo_top/dut/full_a	Immediate	SVA	on	0	1	-	-	-	-	off	assert (full==1)	✓
+/fifo_top/dut/empty_a	Immediate	SVA	on	0	1	-	-	-	-	off	assert (empty==1)	✓
+/fifo_top/dut/almostfull_a	Immediate	SVA	on	0	1	-	-	-	-	off	assert (almostfull==1)	✓
+/fifo_top/dut/almostempty_a	Immediate	SVA	on	0	1	-	-	-	-	off	assert (almostempty==1)	✓
+/fifo_top/dut/ack_a	Concurrent	SVA	on	0	1	-	0B...	off	assert (@posedge ck) disable iff (rst_n==0) (((wr_en&&count<8)) >=wr_ack)	✓
+/fifo_top/dut/overflow_a	Concurrent	SVA	on	0	1	-	0B...	off	assert (@posedge ck) disable iff (rst_n==0) (((count==8&&wr_en)) >=overflow==1)	✓
+/fifo_top/dut/underflow_a	Concurrent	SVA	on	0	1	-	0B...	off	assert (@posedge ck) disable iff (rst_n==0) (((empty&&rd_en)) >=underflow)	✓
+/fifo_top/dut/wr_ptr_a	Concurrent	SVA	on	0	1	-	0B...	off	assert (@posedge ck) disable iff (rst_n==0) (((rd_en&&count<8)) >=rd_ptr==\$past(wr_ptr)+1)	✓
+/fifo_top/dut/rd_ptr_a	Concurrent	SVA	on	0	1	-	0B...	off	assert (@posedge ck) disable iff (rst_n==0) (((wr_en&&count<8)) >=rd_ptr==\$past(rd_ptr)+1)	✓
+/fifo_top/dut/count_write_priority_a	Concurrent	SVA	on	0	1	-	0B...	off	assert (@posedge ck) disable iff (rst_n==0) (((wr_en&&rd_en)&&empty)) >=count==\$past(count)+1)	✓
+/fifo_top/dut/count_read_priority_a	Concurrent	SVA	on	0	1	-	0B...	off	assert (@posedge ck) disable iff (rst_n==0) (((wr_en&&rd_en)&&full)) >=count==\$past(count)-1)	✓
+/fifo_top/dut/count_w_a	Concurrent	SVA	on	0	1	-	0B...	off	assert (@posedge ck) disable iff (rst_n==0) (((wr_en&&rd_en)&&full)) >=count==\$past(count)+1)	✓
+/fifo_top/dut/count_r_a	Concurrent	SVA	on	0	1	-	0B...	off	assert (@posedge ck) disable iff (rst_n==0) (((~wr_en&&rd_en)&&empty)) >=count==\$past(count)-1)	✓
+/fifo_top/tb/#publ:217929410/#38/Immed_39	Immediate	SVA	on	0	1	-	-	-	-	off	assert (randomize(...))	✓

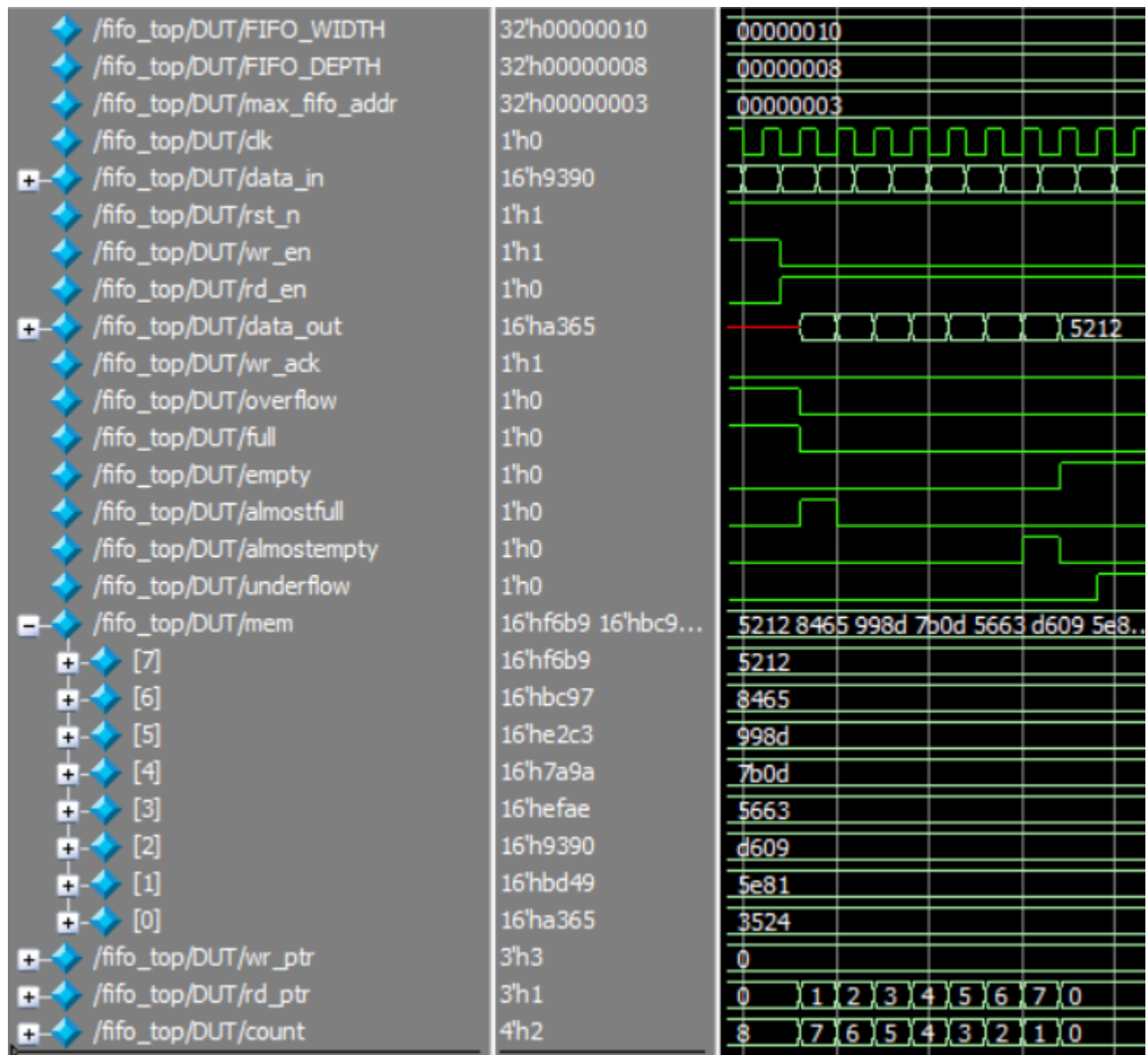
	/fifo_top/dut/count_aa	INACTIVE
	/fifo_top/dut/wr_ptr_aa	INACTIVE
	/fifo_top/dut/rd_ptr_aa	INACTIVE
	/fifo_top/dut/full_aa	INACTIVE
	/fifo_top/dut/empty_aa	INACTIVE
	/fifo_top/dut/underflow_aa	INACTIVE
	/fifo_top/dut/almostfull_aa	INACTIVE
	/fifo_top/dut/almostempty_aa	INACTIVE
	/fifo_top/dut/wr_ack_aa	INACTIVE
	/fifo_top/dut/overflow_aa	INACTIVE
	/fifo_top/dut/full_a	INACTIVE
	/fifo_top/dut/empty_a	INACTIVE
	/fifo_top/dut/almostfull_a	INACTIVE
	/fifo_top/dut/almostempty_a	INACTIVE
	/fifo_top/dut/ack_a	ACTIVE
	/fifo_top/dut/overflow_a	INACTIVE
	/fifo_top/dut/underflow_a	INACTIVE
	/fifo_top/dut/wr_ptr_a	ACTIVE
	/fifo_top/dut/rd_ptr_a	ACTIVE
	/fifo_top/dut/count_write_priority_a	INACTIVE
	/fifo_top/dut/count_read_priority_a	INACTIVE
	/fifo_top/dut/count_wv_a	INACTIVE
	/fifo_top/dut/count_rj_a	INACTIVE

Snippets:

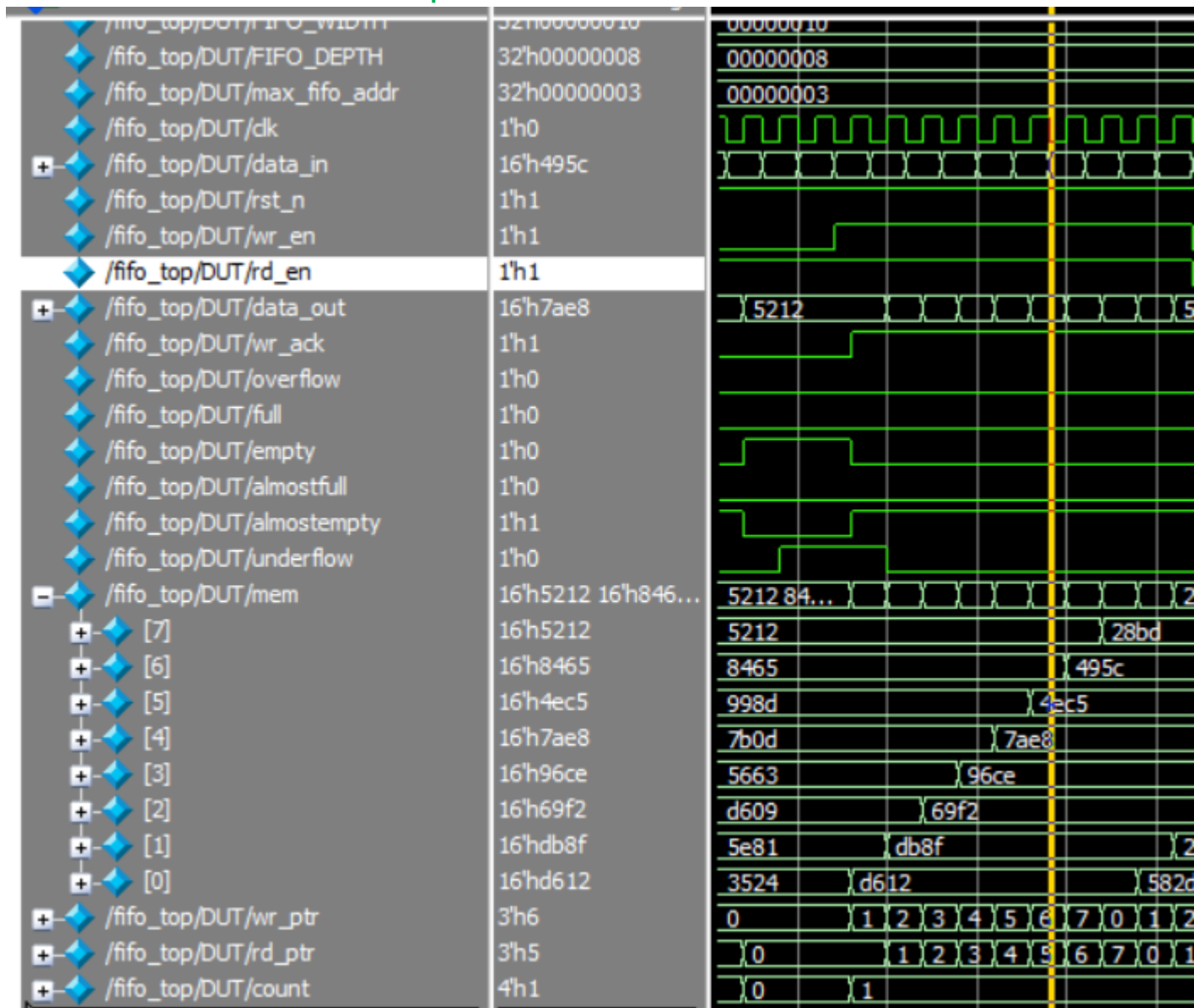
- Write only sequence:



- Read only sequence:

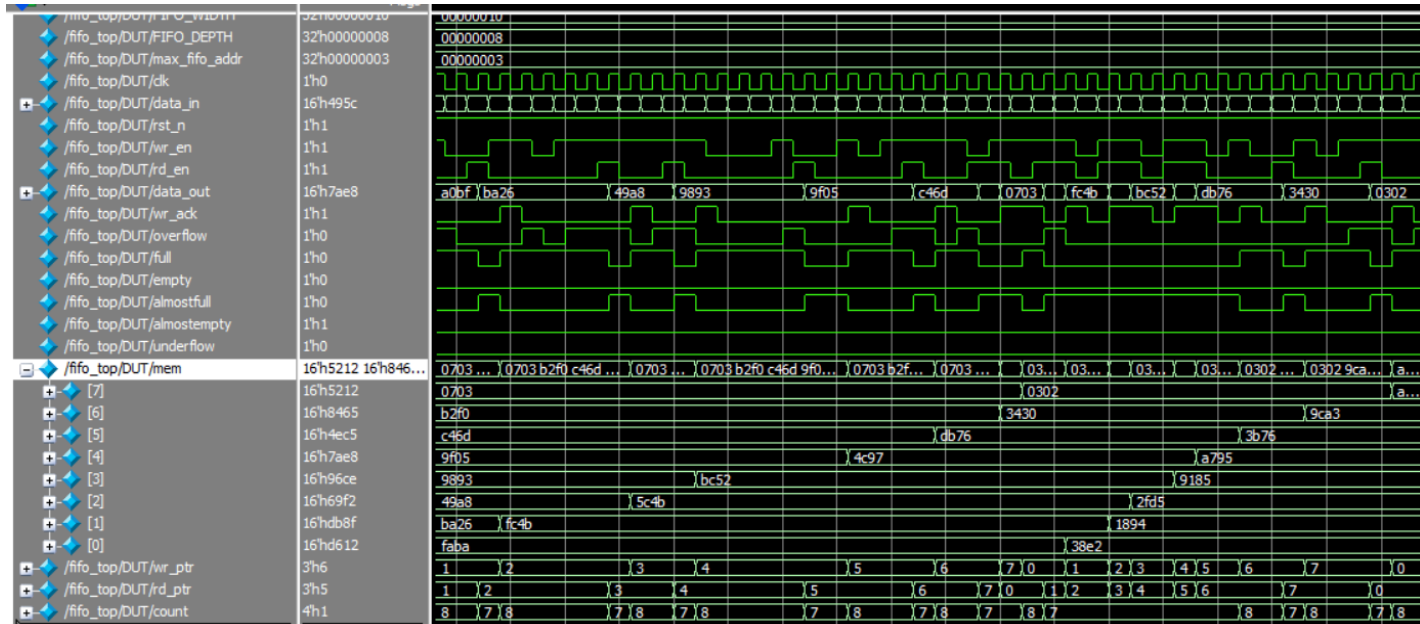


- Both Write and Read Sequence:

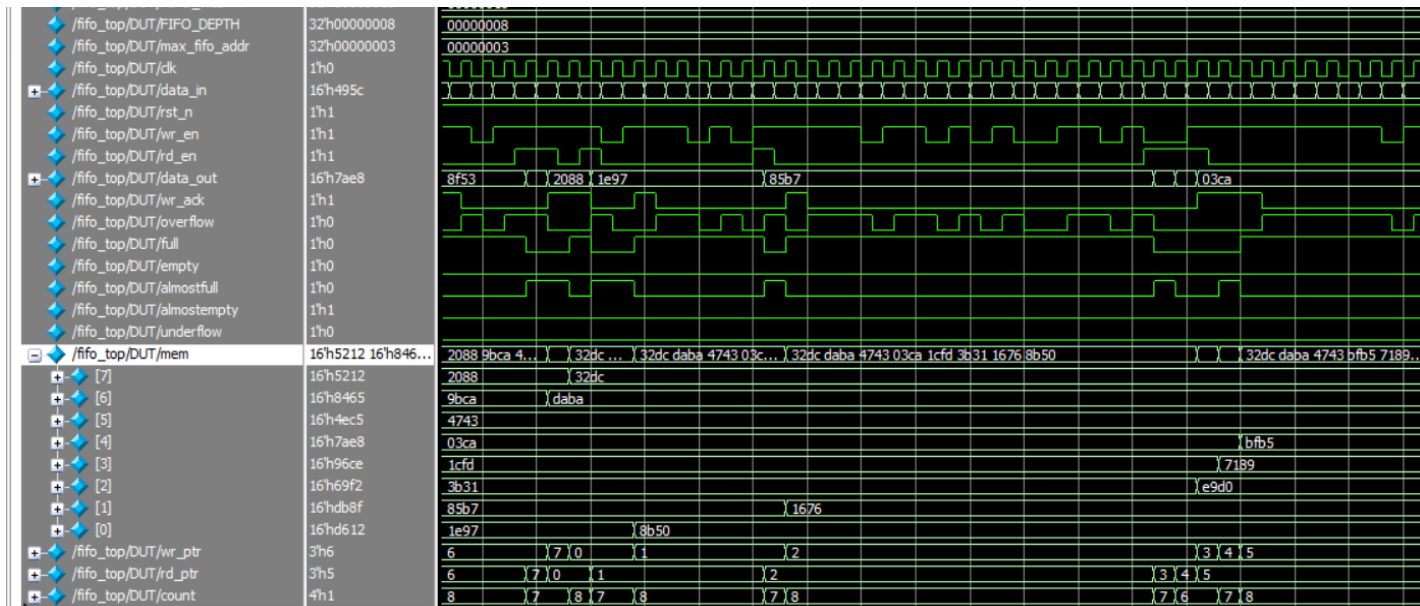


- Main (Randomization) Sequence:

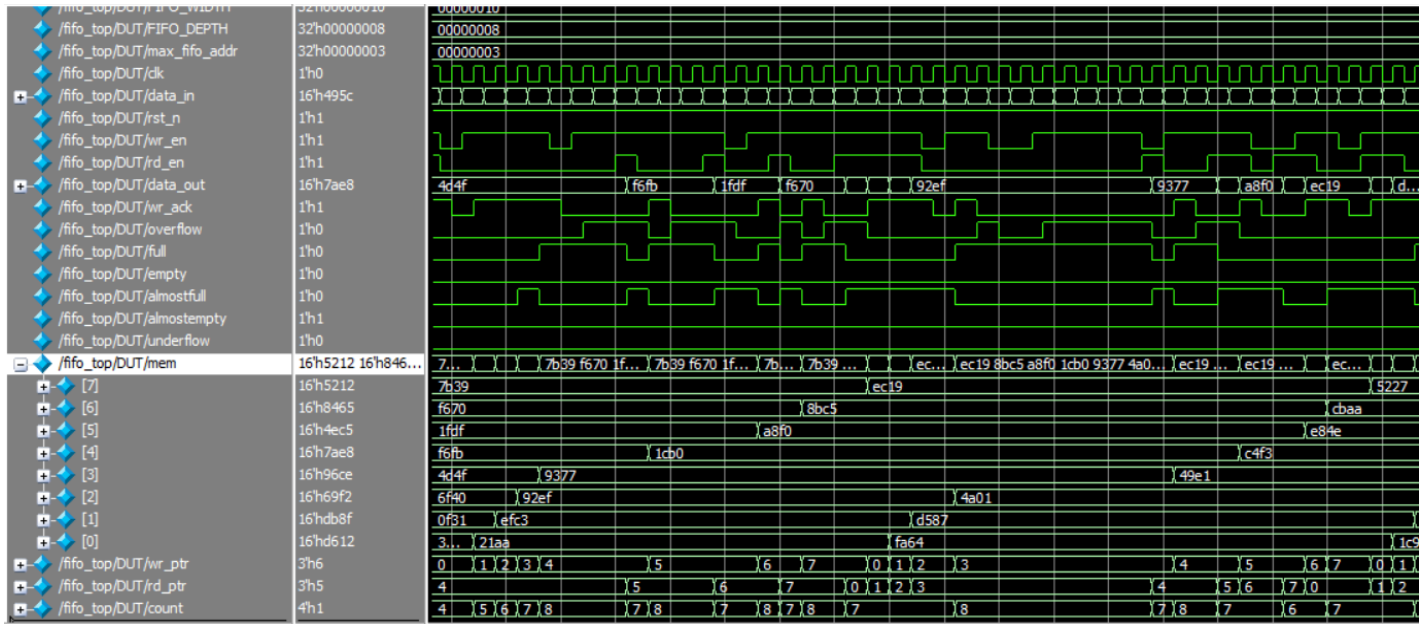
- Random Snippet:



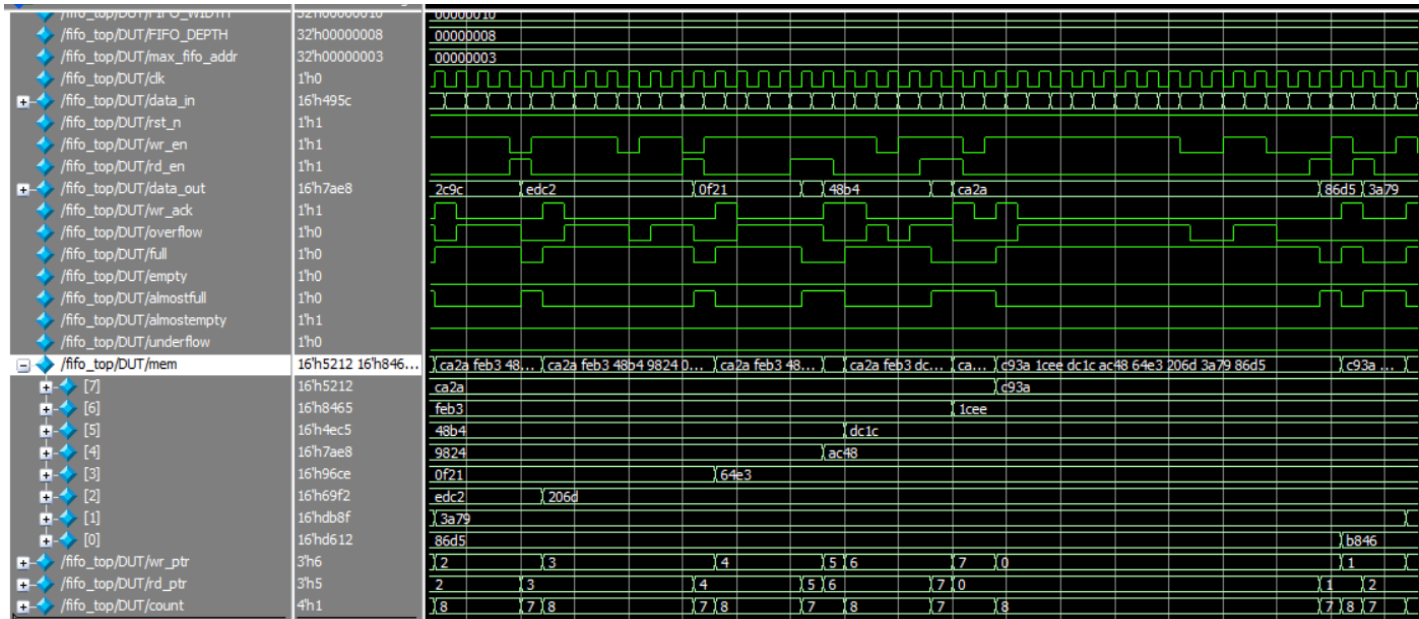
- Random Snippet:



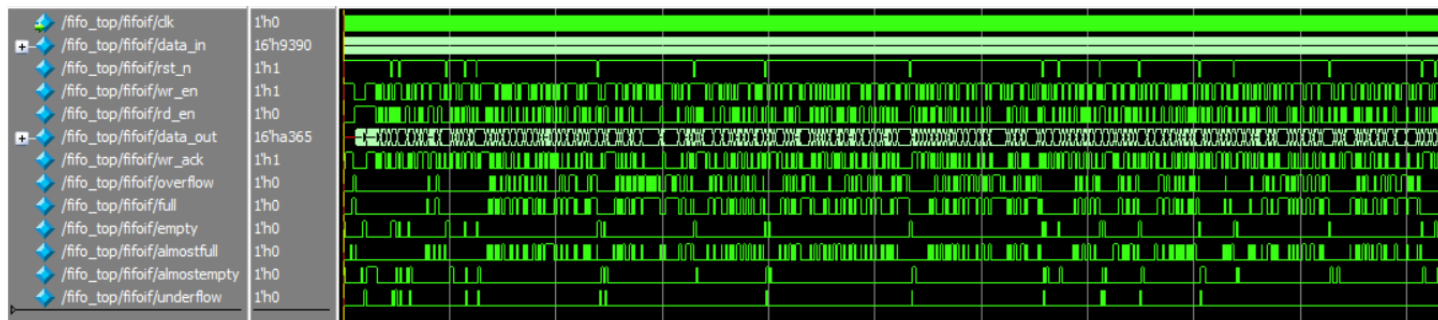
- Random Snippet:



- Random Snippet:



- Full wave:



- UVM Report:

```
#
# You are using a version of the UVM library that has been compiled
# with `UVM_NO_DEPRECATED` undefined.
# See http://www.eda.org/svdb/view.php?id=3313 for more details.
#
# You are using a version of the UVM library that has been compiled
# with `UVM_OBJECT_MUST_HAVE_CONSTRUCTOR` undefined.
# See http://www.eda.org/svdb/view.php?id=3770 for more details.
#
# (Specify +UVM_NO_RELNOTES to turn off this notice)
#
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.3
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(217) @ 0: reporter [Questa UVM] questa_uvm::init(+struct)
# UVM_INFO @ 0: reporter [RNTST] Running test fifo_test...
# UVM_INFO FIFO_TEST.sv(55) @ 0: uvm_test_top [run_phase] reset asserted
# UVM_INFO FIFO_TEST.sv(57) @ 20: uvm_test_top [run_phase] reset dasserted
# UVM_INFO FIFO_TEST.sv(60) @ 20: uvm_test_top [run_phase] write asserted
# UVM_INFO FIFO_TEST.sv(62) @ 220: uvm_test_top [run_phase] write dasserted
# UVM_INFO FIFO_TEST.sv(65) @ 220: uvm_test_top [run_phase] read asserted
# UVM_INFO FIFO_TEST.sv(67) @ 420: uvm_test_top [run_phase] read dasserted
# UVM_INFO FIFO_TEST.sv(70) @ 420: uvm_test_top [run_phase] write_read asserted
# UVM_INFO FIFO_TEST.sv(72) @ 620: uvm_test_top [run_phase] write_read dasserted
# UVM_INFO FIFO_TEST.sv(75) @ 620: uvm_test_top [run_phase] main asserted
# UVM_INFO FIFO_TEST.sv(77) @ 20620: uvm_test_top [run_phase] main dasserted
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @ 20620: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO FIFO_SCOREBOARD.sv(121) @ 20620: uvm_test_top.env.sb [report_phase] total successful transactions: 1031
# UVM_INFO FIFO_SCOREBOARD.sv(122) @ 20620: uvm_test_top.env.sb [report_phase] total failed transactions: 0
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 16
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
#
# ** Report counts by id
# [Questa UVM] 2
# [RNTST] 1
# [TEST_DONE] 1
# [report_phase] 2
# [run_phase] 10
#
# ** Note: $finish : C:/modeltech64_10.6d/win64/./verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
# Time: 20620 ns Iteration: 60 Instance: //fifo_top
```

Assertions Table:

	Feature	Assertion
1		
2	After reset, DUT.count should be 0	assert final(DUT.count == 0);
3	After reset, DUT.wr_ptr should be 0	assert final(DUT.wr_ptr == 0);
4	After reset, DUT.rd_ptr should be 0	assert final(DUT.rd_ptr == 0);
5	After reset, FIFO full flag should be 0	assert final(fifoif.full == 0);
6	After reset, FIFO empty flag should be 1	assert final(fifoif.empty == 1);
7	After reset, FIFO underflow flag should be 0	assert final(fifoif.underflow == 0);
8	After reset, FIFO almost full flag should be 0	assert final(fifoif.almostfull == 0);
9	After reset, FIFO almost empty flag should be 0	assert final(fifoif.almostempty == 0);
10	After reset, FIFO write acknowledge (wr_ack) should be 0	assert final(fifoif.wr_ack == 0);
11	After reset, FIFO overflow flag should be 0	assert final(fifoif.overflow == 0);
12	FIFO is full when DUT.count equals FIFO_DEPTH	assert final(fifoif.full == 1);
13	FIFO is empty when DUT.count equals 0	assert final(fifoif.empty == 1);
14	FIFO is almost full when DUT.count equals FIFO_DEPTH - 1	assert final(fifoif.almostfull == 1);
15	FIFO is almost empty when DUT.count equals 1	assert final(fifoif.almostempty == 1);
16	Write acknowledge should be asserted when write is enabled and FIFO is not full	`@(posedge fifoif.clk) disable iff (fifoif.rst_n == 0) fifoif.wr_en && (DUT.count < FIFO_DEPTH)
17	Overflow flag should be set when FIFO is full and write enable is asserted	`@(posedge fifoif.clk) disable iff (fifoif.rst_n == 0) ((DUT.count == FIFO_DEPTH) && fifoif.wr_en)
18	Underflow flag should be set when FIFO is empty and read enable is asserted	`@(posedge fifoif.clk) disable iff (fifoif.rst_n == 0) (fifoif.empty) && (fifoif.rd_en)
19	Write pointer should increment on write when FIFO is not full	`@(posedge fifoif.clk) disable iff (fifoif.rst_n == 0) (fifoif.wr_en) && (DUT.count < FIFO_DEPTH)
20	Read pointer should increment on read when FIFO is not empty	`@(posedge fifoif.clk) disable iff (fifoif.rst_n == 0) (fifoif.rd_en) && (DUT.count != 0)
21	Priority for write over read when FIFO is empty	`@(posedge fifoif.clk) disable iff (fifoif.rst_n == 0) (fifoif.wr_en) && (fifoif.rd_en) && (fifoif.empty)
22	Priority for read over write when FIFO is full	`@(posedge fifoif.clk) disable iff (fifoif.rst_n == 0) (fifoif.wr_en) && (fifoif.rd_en) && (fifoif.full)
23	Count should increment on write when no read is happening	`@(posedge fifoif.clk) disable iff (fifoif.rst_n == 0) (fifoif.wr_en) && (!fifoif.rd_en) && (!fifoif.full)
24	Count should decrement on read when no write is happening	`@(posedge fifoif.clk) disable iff (fifoif.rst_n == 0) (!fifoif.wr_en) && (fifoif.rd_en) && (!fifoif.empty)