REPORT

Project (1) FIFO

TEAM #21

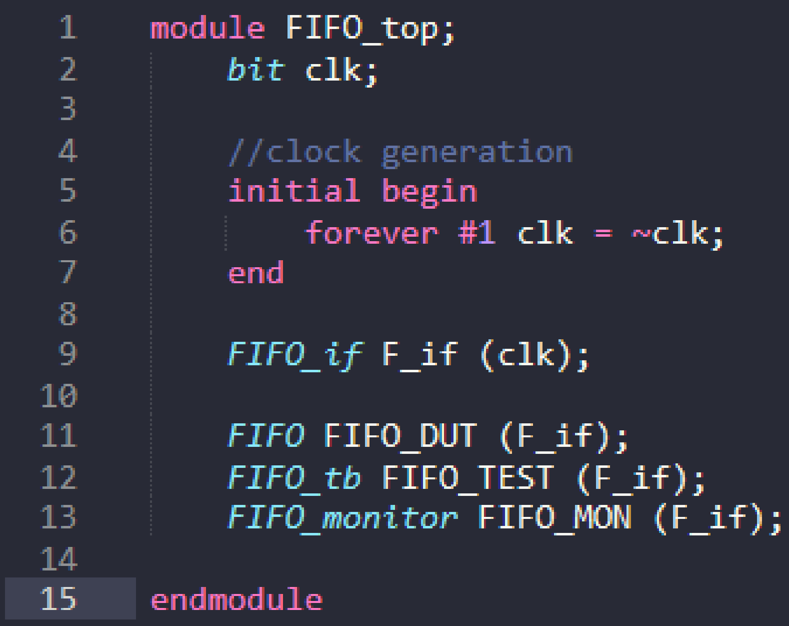
PREPARED BY
Nouran Hamdy

SUBMITTED TO
Eng Kareem Waseem

TABLE OF CONTENTS

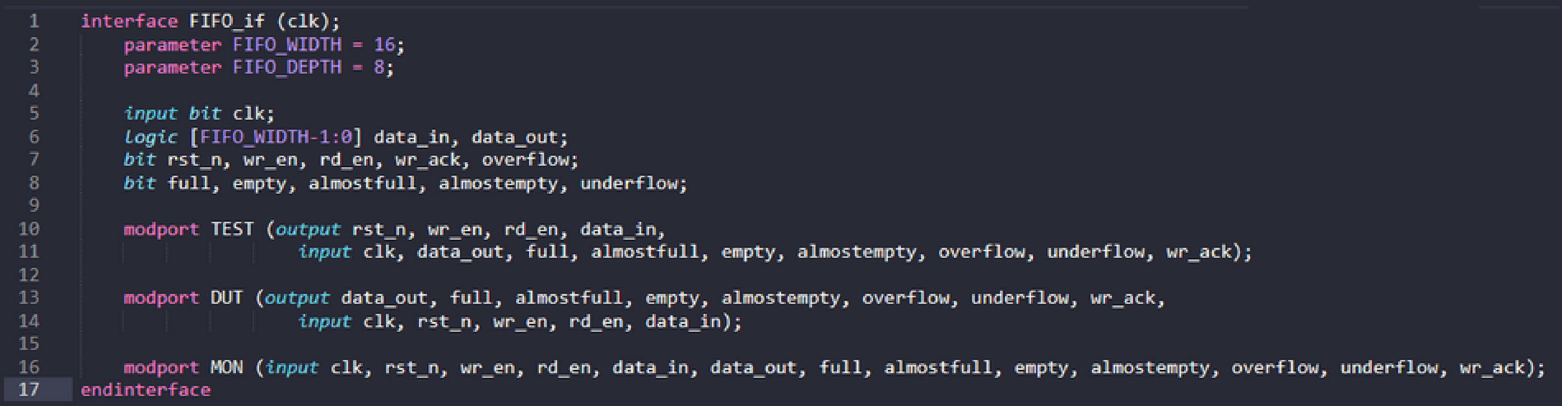
Code Snippets

TOP



```
module FIFO_top;
    bit clk;

    //clock generation
    initial begin
        forever #1 clk = ~clk;
    end

    FIFO_if F_if (clk);

    FIFO FIFO_DUT (F_if);
    FIFO_tb FIFO_TEST (F_if);
    FIFO_monitor FIFO_MON (F_if);

endmodule
```

INTERFACE

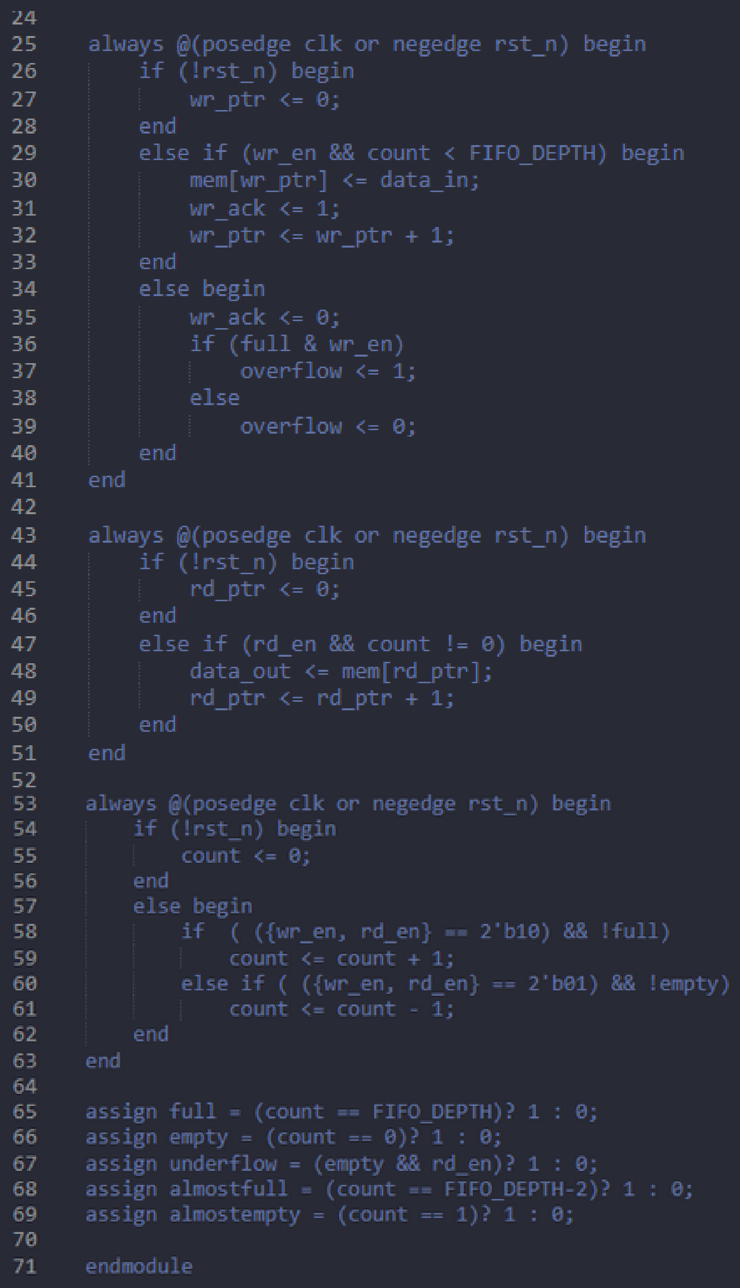


```
interface FIFO_if (clk);
    parameter FIFO_WIDTH = 16;
    parameter FIFO_DEPTH = 8;

    input bit clk;
    logic [FIFO_WIDTH-1:0] data_in, data_out;
    bit rst_n, wr_en, rd_en, wr_ack, overflow;
    bit full, empty, almostfull, almostempty, underflow;

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

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

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

Code Snippets

DESIGN

Before (Original)

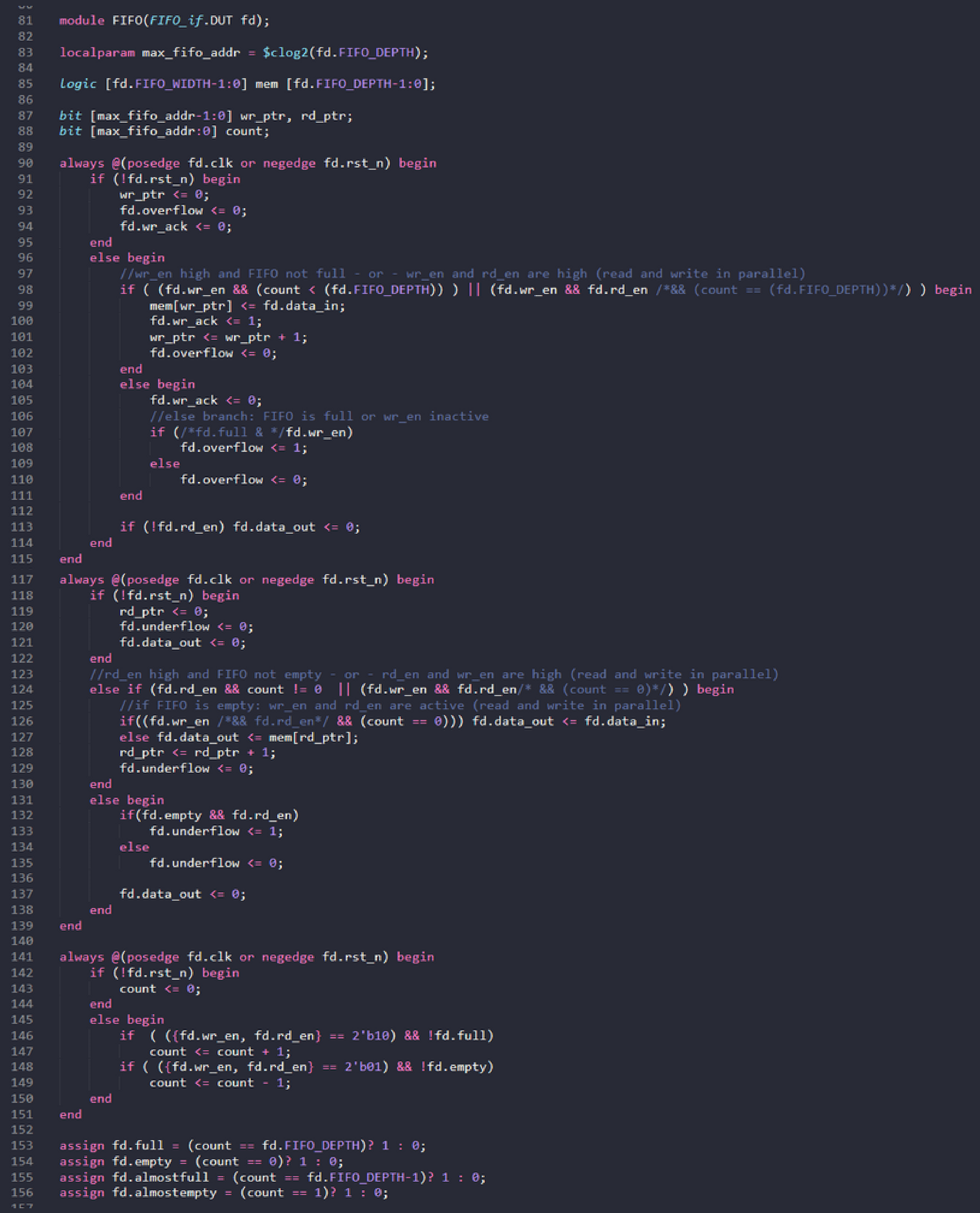


```verilog

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

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

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

endmodule
```

Code Snippets

DESIGN

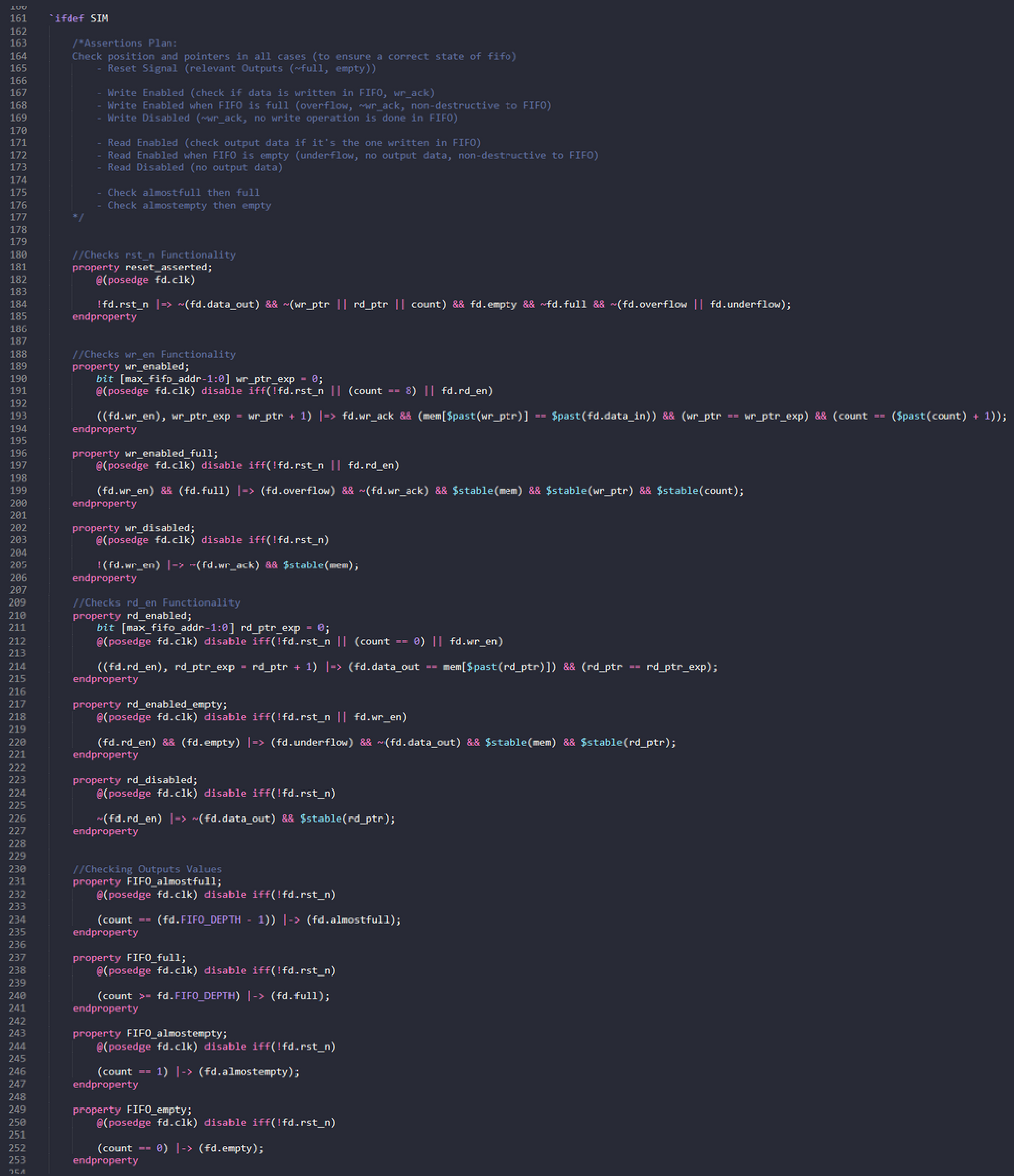
After (Edited)



```
module FIFO(FIFO_if.DUT fd);

localparam max_fifo_addr = $clog2(fd.FIFO_DEPTH);

logic [fd.FIFO_WIDTH-1:0] mem [fd.FIFO_DEPTH-1:0];

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

always @(posedge fd.clk or negedge fd.rst_n) begin
    if (!fd.rst_n) begin
        wr_ptr <= 0;
        fd.overflow <= 0;
        fd.wr_ack <= 0;
    end
    else begin
        //wr_en high and FIFO not full - or - wr_en and rd_en are high (read and write in parallel)
        if ( (fd.wr_en && (count < (fd.FIFO_DEPTH)) ) || (fd.wr_en && fd.rd_en /*&& (count == (fd.FIFO_DEPTH))*/) ) begin
            mem[wr_ptr] <= fd.data_in;
            fd.wr_ack <= 1;
            wr_ptr <= wr_ptr + 1;
            fd.overflow <= 0;
        end
        else begin
            fd.wr_ack <= 0;
            //else branch: FIFO is full or wr_en inactive
            if (/*fd.full & */fd.wr_en)
                fd.overflow <= 1;
            else
                fd.overflow <= 0;
        end

        if (!fd.rd_en) fd.data_out <= 0;
    end
end
always @(posedge fd.clk or negedge fd.rst_n) begin
    if (!fd.rst_n) begin
        rd_ptr <= 0;
        fd.underflow <= 0;
        fd.data_out <= 0;
    end
    //rd_en high and FIFO not empty - or - rd_en and wr_en are high (read and write in parallel)
    else if (fd.rd_en && count != 0  || (fd.wr_en && fd.rd_en/* && (count == 0)*/) ) begin
        //if FIFO is empty: wr_en and rd_en are active (read and write in parallel)
        if((fd.wr_en /*&& fd.rd_en*/ && (count == 0))) fd.data_out <= fd.data_in;
        else fd.data_out <= mem[rd_ptr];
        rd_ptr <= rd_ptr + 1;
        fd.underflow <= 0;
    end
    else begin
        if(fd.empty && fd.rd_en)
            fd.underflow <= 1;
        else
            fd.underflow <= 0;

        fd.data_out <= 0;
    end
end

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

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

Code Snippets

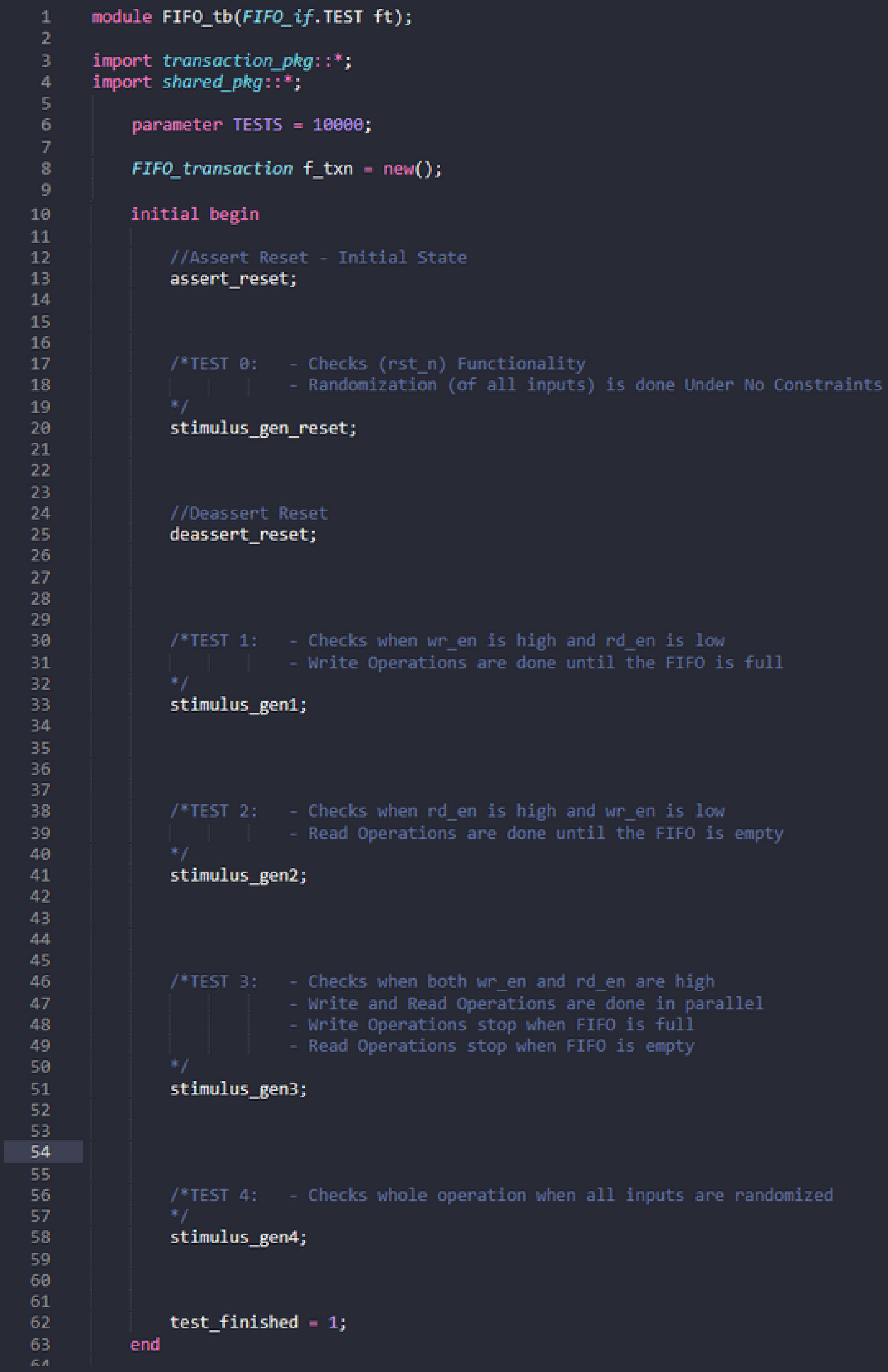
ASSERTIONS



```
`ifdef SIM

    /*Assertions Plan:
    Check position and pointers in all cases (to ensure a correct state of fifo)
        - Reset Signal (relevant Outputs (~full, empty))

        - Write Enabled (check if data is written in FIFO, wr_ack)
        - Write Enabled when FIFO is full (overflow, ~wr_ack, non-destructive to FIFO)
        - Write Disabled (~wr_ack, no write operation is done in FIFO)

        - Read Enabled (check output data if it's the one written in FIFO)
        - Read Enabled when FIFO is empty (underflow, no output data, non-destructive to FIFO)
        - Read Disabled (no output data)

        - Check almostfull then full
        - Check almostempty then empty
    */


    //Checks rst_n Functionality
    property reset_asserted;
        @(posedge fd.clk)

        !fd.rst_n |=> ~(fd.data_out) && ~(wr_ptr || rd_ptr || count) && fd.empty && ~fd.full && ~(fd.overflow || fd.underflow);
    endproperty


    //Checks wr_en Functionality
    property wr_enabled;
        bit [max_fifo_addr-1:0] wr_ptr_exp = 0;
        @(posedge fd.clk) disable iff(!fd.rst_n || (count == 8) || fd.rd_en)

        ((fd.wr_en), wr_ptr_exp = wr_ptr + 1) |=> fd.wr_ack && (mem[$past(wr_ptr)] == $past(fd.data_in)) && (wr_ptr == wr_ptr_exp) && (count == ($past(count) + 1));
    endproperty

    property wr_enabled_full;
        @(posedge fd.clk) disable iff(!fd.rst_n || fd.rd_en)

        (fd.wr_en) && (fd.full) |=> (fd.overflow) && ~(fd.wr_ack) && $stable(mem) && $stable(wr_ptr) && $stable(count);
    endproperty

    property wr_disabled;
        @(posedge fd.clk) disable iff(!fd.rst_n)

        !(fd.wr_en) |=> ~(fd.wr_ack) && $stable(mem);
    endproperty

    //Checks rd_en Functionality
    property rd_enabled;
        bit [max_fifo_addr-1:0] rd_ptr_exp = 0;
        @(posedge fd.clk) disable iff(!fd.rst_n || (count == 0) || fd.wr_en)

        ((fd.rd_en), rd_ptr_exp = rd_ptr + 1) |=> (fd.data_out == mem[$past(rd_ptr)]) && (rd_ptr == rd_ptr_exp);
    endproperty

    property rd_enabled_empty;
        @(posedge fd.clk) disable iff(!fd.rst_n || fd.wr_en)

        (fd.rd_en) && (fd.empty) |=> (fd.underflow) && ~(fd.data_out) && $stable(mem) && $stable(rd_ptr);
    endproperty

    property rd_disabled;
        @(posedge fd.clk) disable iff(!fd.rst_n)

        ~(fd.rd_en) |=> ~(fd.data_out) && $stable(rd_ptr);
    endproperty


    //Checking Outputs Values
    property FIFO_almostfull;
        @(posedge fd.clk) disable iff(!fd.rst_n)

        (count == (fd.FIFO_DEPTH - 1)) |-> (fd.almostfull);
    endproperty

    property FIFO_full;
        @(posedge fd.clk) disable iff(!fd.rst_n)

        (count >= fd.FIFO_DEPTH) |-> (fd.full);
    endproperty

    property FIFO_almostempty;
        @(posedge fd.clk) disable iff(!fd.rst_n)

        (count == 1) |-> (fd.almostempty);
    endproperty

    property FIFO_empty;
        @(posedge fd.clk) disable iff(!fd.rst_n)

        (count == 0) |-> (fd.empty);
    endproperty
```

CODE SNIPPETS

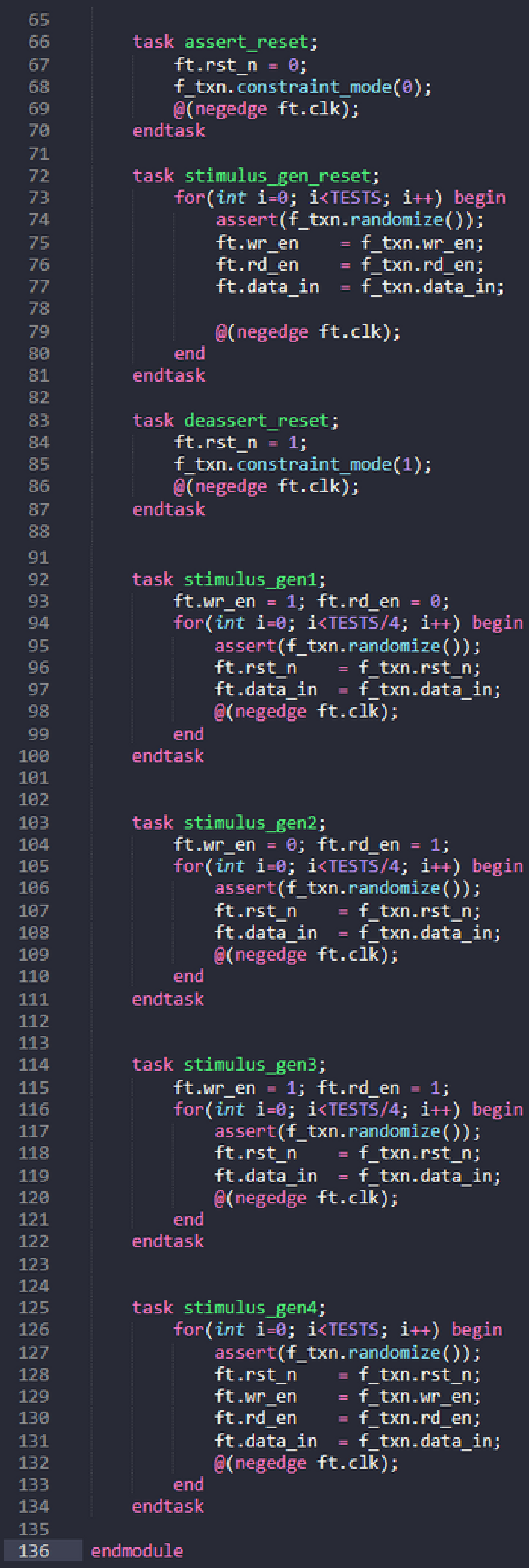
TESTBENCH



```systemverilog
module FIFO_tb(FIFO_if.TEST ft);

import transaction_pkg::*;
import shared_pkg::*;

    parameter TESTS = 10000;

    FIFO_transaction f_txn = new();

    initial begin

        //Assert Reset - Initial State
        assert_reset;


        /*TEST 0:   - Checks (rst_n) Functionality
                    - Randomization (of all inputs) is done Under No Constraints
        */
        stimulus_gen_reset;


        //Deassert Reset
        deassert_reset;



        /*TEST 1:   - Checks when wr_en is high and rd_en is low
                    - Write Operations are done until the FIFO is full
        */
        stimulus_gen1;



        /*TEST 2:   - Checks when rd_en is high and wr_en is low
                    - Read Operations are done until the FIFO is empty
        */
        stimulus_gen2;



        /*TEST 3:   - Checks when both wr_en and rd_en are high
                    - Write and Read Operations are done in parallel
                    - Write Operations stop when FIFO is full
                    - Read Operations stop when FIFO is empty
        */
        stimulus_gen3;



        /*TEST 4:   - Checks whole operation when all inputs are randomized
        */
        stimulus_gen4;


        test_finished = 1;
    end
```

Code Snippets

TESTBENCH



```
    task assert_reset;
        ft.rst_n = 0;
        f_txn.constraint_mode(0);
        @(negedge ft.clk);
    endtask

    task stimulus_gen_reset;
        for(int i=0; i<TESTS; i++) begin
            assert(f_txn.randomize());
            ft.wr_en     = f_txn.wr_en;
            ft.rd_en     = f_txn.rd_en;
            ft.data_in   = f_txn.data_in;

            @(negedge ft.clk);
        end
    endtask

    task deassert_reset;
        ft.rst_n = 1;
        f_txn.constraint_mode(1);
        @(negedge ft.clk);
    endtask


    task stimulus_gen1;
        ft.wr_en = 1; ft.rd_en = 0;
        for(int i=0; i<TESTS/4; i++) begin
            assert(f_txn.randomize());
            ft.rst_n     = f_txn.rst_n;
            ft.data_in   = f_txn.data_in;
            @(negedge ft.clk);
        end
    endtask


    task stimulus_gen2;
        ft.wr_en = 0; ft.rd_en = 1;
        for(int i=0; i<TESTS/4; i++) begin
            assert(f_txn.randomize());
            ft.rst_n     = f_txn.rst_n;
            ft.data_in   = f_txn.data_in;
            @(negedge ft.clk);
        end
    endtask


    task stimulus_gen3;
        ft.wr_en = 1; ft.rd_en = 1;
        for(int i=0; i<TESTS/4; i++) begin
            assert(f_txn.randomize());
            ft.rst_n     = f_txn.rst_n;
            ft.data_in   = f_txn.data_in;
            @(negedge ft.clk);
        end
    endtask


    task stimulus_gen4;
        for(int i=0; i<TESTS; i++) begin
            assert(f_txn.randomize());
            ft.rst_n     = f_txn.rst_n;
            ft.wr_en     = f_txn.wr_en;
            ft.rd_en     = f_txn.rd_en;
            ft.data_in   = f_txn.data_in;
            @(negedge ft.clk);
        end
    endtask

endmodule
```

Code Snippets

MONITOR

```systemverilog
import transaction_pkg::*;
import coverage_pkg::*;
import scoreboard_pkg::*;
import shared_pkg::*;


module FIFO_monitor(FIFO_if.MON fm);
    FIFO_transaction f_txn   = new();
    FIFO_coverage    f_cov   = new();
    FIFO_scoreboard  f_score = new();

    FIFO_transaction f_tcov  = new();

    initial begin
        forever begin
            fork
                //Transaction for Checker
                begin
                    @(edge fm.clk);
                    //Inputs to DUT
                    f_txn.rst_n = fm.rst_n;
                    f_txn.wr_en = fm.wr_en;
                    f_txn.rd_en = fm.rd_en;
                    f_txn.data_in = fm.data_in;

                    //Combinational Outputs of DUT
                    if(!fm.rst_n) begin
                        f_txn.wr_ack = fm.wr_ack;
                        f_txn.overflow = fm.overflow;
                        f_txn.underflow = fm.underflow;
                        f_txn.data_out = fm.data_out;
                    end
                    f_txn.full = fm.full;
                    f_txn.empty = fm.empty;
                    f_txn.almostfull = fm.almostfull;
                    f_txn.almostempty = fm.almostempty;


                    if(~fm.clk) begin
                        //Sequential Outputs of DUT
                        if(fm.rst_n) begin
                            f_txn.wr_ack = fm.wr_ack;
                            f_txn.overflow = fm.overflow;
                            f_txn.underflow = fm.underflow;
                            f_txn.data_out = fm.data_out;
                        end
                    end

                    if(~fm.clk) f_score.check_data(f_txn);


                    if(shared_pkg::test_finished) begin
                        $display("--------------------------------------------------------------------");
                        $display("----------------Correct Count and Error Count Summary---------------");
                        $display("--------------------------------------------------------------------");
                        $display("wr_ack:       \t\tCorrect Count = %0d,    Error Count = %0d", shared_pkg::correct_count_wr_ack,       shared_pkg::error_count_wr_ack);
                        $display("full:         \t\tCorrect Count = %0d,    Error Count = %0d", shared_pkg::correct_count_full,         shared_pkg::error_count_full);
                        $display("empty:        \t\tCorrect Count = %0d,    Error Count = %0d", shared_pkg::correct_count_empty,        shared_pkg::error_count_empty);
                        $display("almostfull:   \t\t\tCorrect Count = %0d,  Error Count = %0d", shared_pkg::correct_count_almostfull,   shared_pkg::error_count_almostfull);
                        $display("almostempty:  \t\t\tCorrect Count = %0d,  Error Count = %0d", shared_pkg::correct_count_almostempty, shared_pkg::error_count_almostempty);
                        $display("overflow:     \t\t\tCorrect Count = %0d,  Error Count = %0d", shared_pkg::correct_count_overflow,     shared_pkg::error_count_overflow);
                        $display("underflow:    \t\t\tCorrect Count = %0d,  Error Count = %0d", shared_pkg::correct_count_underflow,    shared_pkg::error_count_underflow);
                        $display("data_out:     \t\t\tCorrect Count = %0d,  Error Count = %0d", shared_pkg::correct_count_dout,         shared_pkg::error_count_dout);

                        @(edge fm.clk); $stop;
                    end
                end
                //Transaction for Covergroup
                begin
                    @(edge fm.clk);
                    f_tcov.rst_n = fm.rst_n;
                    f_tcov.wr_en = fm.wr_en;
                    f_tcov.rd_en = fm.rd_en;
                    f_tcov.data_in = fm.data_in;

                    f_tcov.wr_ack = fm.wr_ack;
                    f_tcov.overflow = fm.overflow;
                    f_tcov.underflow = fm.underflow;
                    f_tcov.data_out = fm.data_out;

                    f_tcov.full = fm.full;
                    f_tcov.empty = fm.empty;
                    f_tcov.almostfull = fm.almostfull;
                    f_tcov.almostempty = fm.almostempty;

                    f_cov.sample_data(f_tcov);
                end
            join_any
        end
    end
endmodule
```

Code Snippets

MONITOR

In the Monitor Module:
A different approach is used instead of the suggested one, to be able to compare the sequential output of the DUT with the combinational output of the Reference Model.

Two Issues I have encountered using the suggested approach in the Project File:

- Sampling time: Driving Inputs in the testbench at clock’s negative edge and sampling inputs and outputs in the monitor module at clock’s negative edge, too. The sampling could not capture the values of inputs changing at clock’s negative edge.

- Outputs’ Checking: Some of the outputs of the DUT are sequential, and would be a little hard (Please Share your thoughts if there is a way) to compare new values of DUT’s outputs with past values of Reference-Model’s outputs. So this issue is handled in the sampling phase, where inputs and combinational outputs are sampled each pos/neg edge (to capture rapid changes of inputs) while sequential outputs are captured at negative edge (to capture the new output value that arrived last positive edge).

Code Snippets

PACKAGES

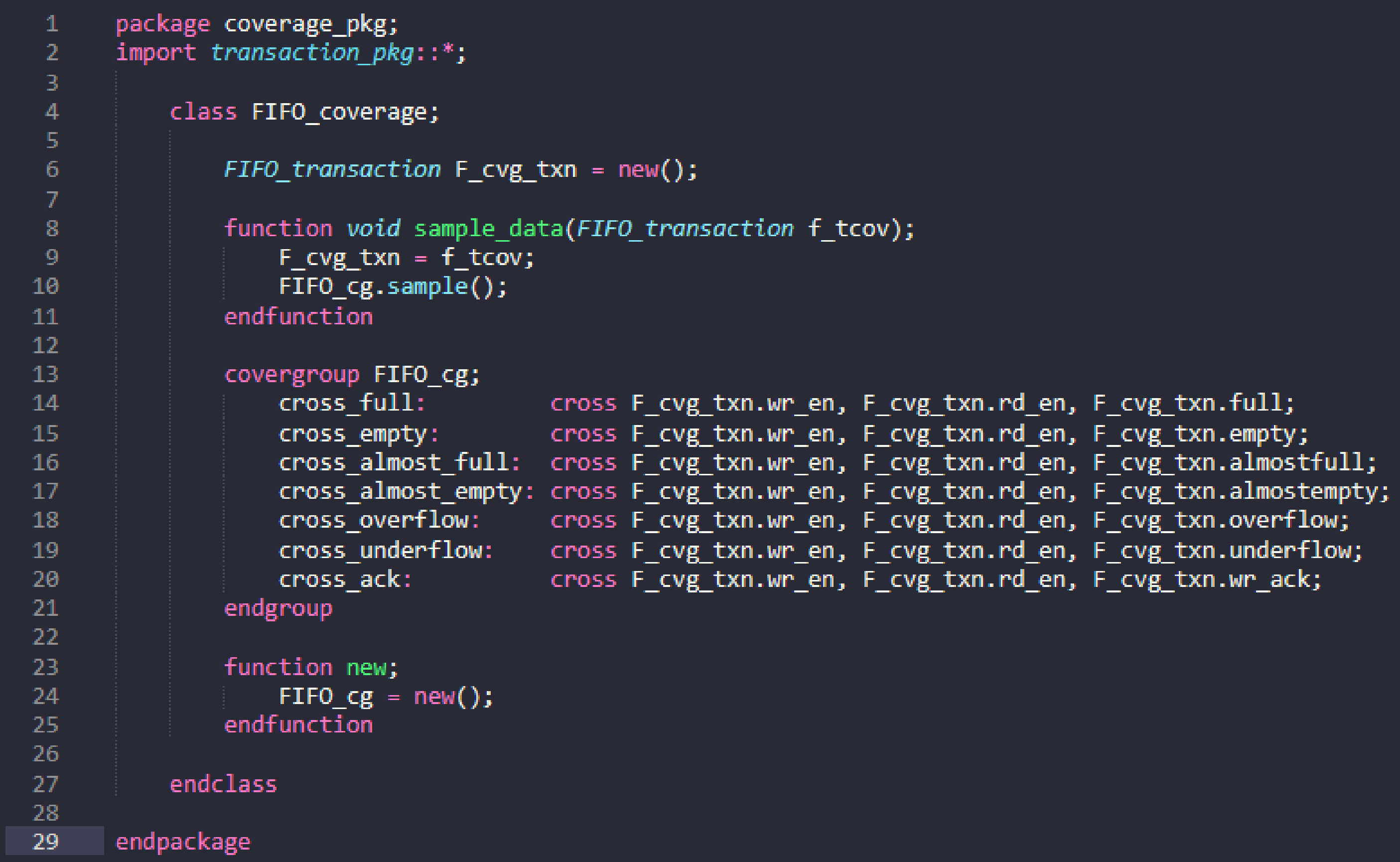
TRANSACTION

```
package transaction_pkg;
    class FIFO_transaction;
        //Parameters
        parameter FIFO_WIDTH = 16;
        parameter FIFO_DEPTH = 8;

        //Inputs to DUT
        rand bit rst_n, wr_en, rd_en;
        rand logic [FIFO_WIDTH-1:0] data_in;

        //Outputs of DUT
        bit wr_ack, full, empty, almostfull, almostempty, overflow, underflow;
        logic [FIFO_WIDTH-1:0] data_out;

        //Constraints (wr_en, rd_en)
        int WR_EN_ON_DIST = 70;
        int RD_EN_ON_DIST = 30;

        constraint rst_c {
            rst_n           dist {0:=5,     1:=95};
        }

        constraint wr_en_c {
            wr_en           dist {0:=(100 - WR_EN_ON_DIST),     1:=WR_EN_ON_DIST};
        }

        constraint rd_en_c {
            rd_en           dist {0:=(100 - RD_EN_ON_DIST),     1:=RD_EN_ON_DIST};
        }

    endclass
endpackage
```

Code Snippets

PACKAGES

COVERAGE

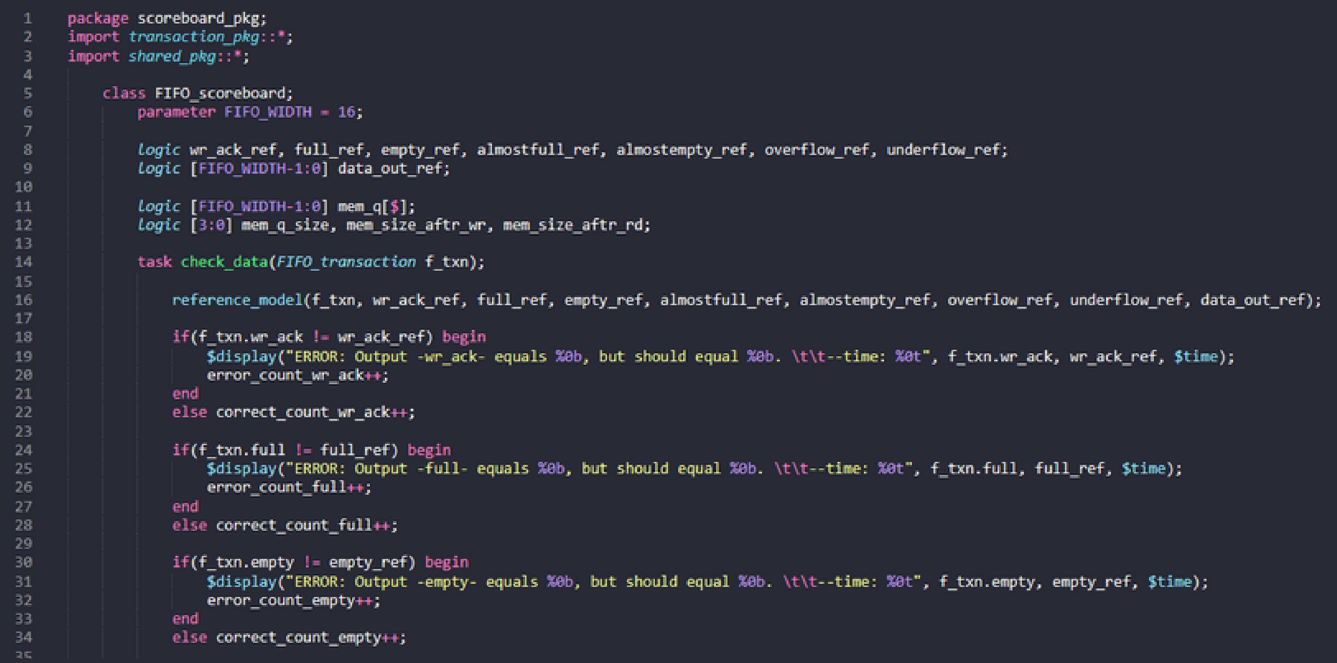


```
package coverage_pkg;
import transaction_pkg::*;

    class FIFO_coverage;

        FIFO_transaction F_cvg_txn = new();

        function void sample_data(FIFO_transaction f_tcov);
            F_cvg_txn = f_tcov;
            FIFO_cg.sample();
        endfunction

        covergroup FIFO_cg;
            cross_full:         cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.full;
            cross_empty:        cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.empty;
            cross_almost_full:  cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.almostfull;
            cross_almost_empty: cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.almostempty;
            cross_overflow:     cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.overflow;
            cross_underflow:    cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.underflow;
            cross_ack:          cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.wr_ack;
        endgroup

        function new;
            FIFO_cg = new();
        endfunction

    endclass

endpackage
```

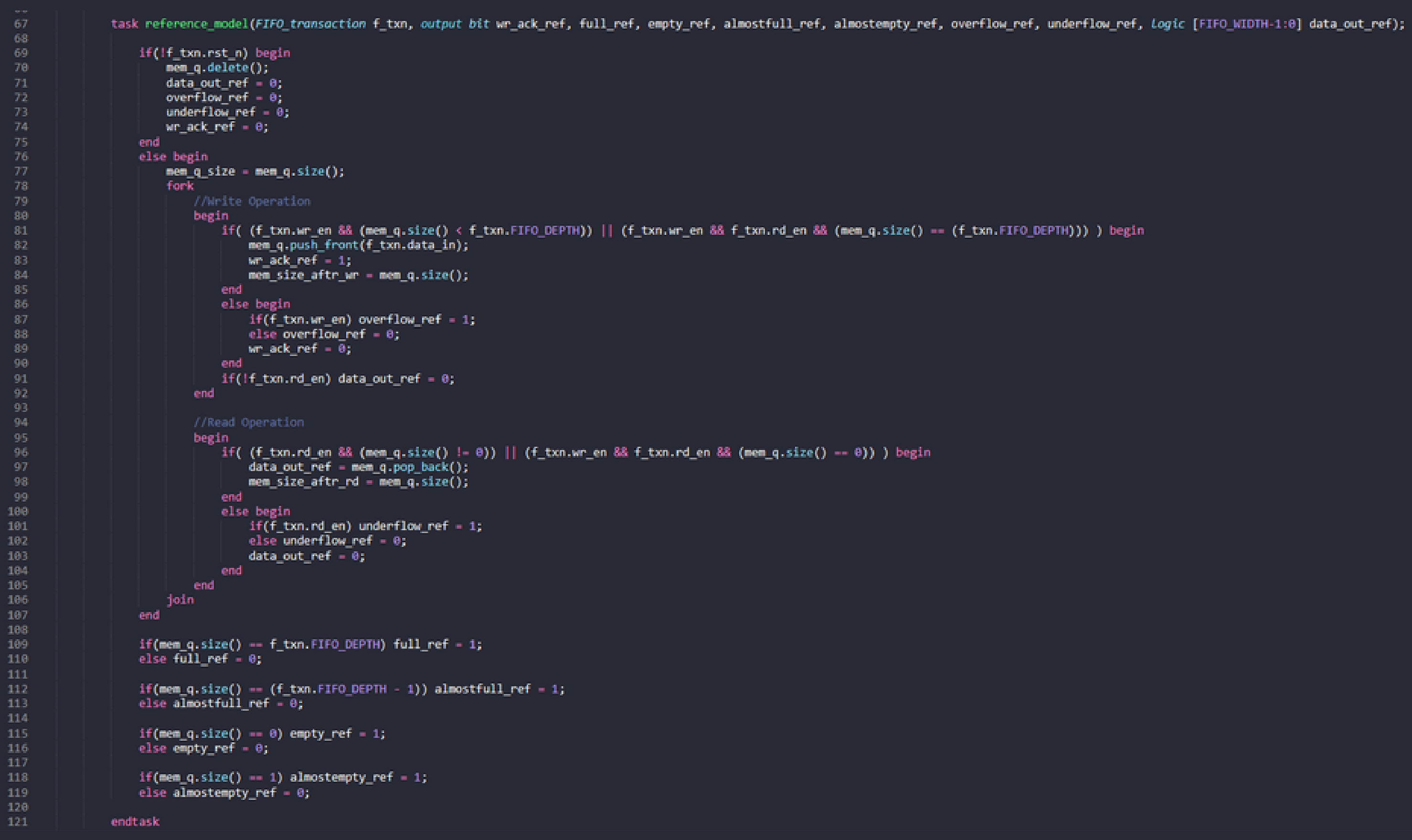
CODE SNIPPETS

PACKAGES

SCOREBOARD



```systemverilog
package scoreboard_pkg;
import transaction_pkg::*;
import shared_pkg::*;

    class FIFO_scoreboard;
        parameter FIFO_WIDTH = 16;

        logic wr_ack_ref, full_ref, empty_ref, almostfull_ref, almostempty_ref, overflow_ref, underflow_ref;
        logic [FIFO_WIDTH-1:0] data_out_ref;

        logic [FIFO_WIDTH-1:0] mem_q[$];
        logic [3:0] mem_q_size, mem_size_aftr_wr, mem_size_aftr_rd;

        task check_data(FIFO_transaction f_txn);

            reference_model(f_txn, wr_ack_ref, full_ref, empty_ref, almostfull_ref, almostempty_ref, overflow_ref, underflow_ref, data_out_ref);

            if(f_txn.wr_ack != wr_ack_ref) begin
                $display("ERROR: Output -wr_ack- equals %0b, but should equal %0b. \t\t--time: %0t", f_txn.wr_ack, wr_ack_ref, $time);
                error_count_wr_ack++;
            end
            else correct_count_wr_ack++;

            if(f_txn.full != full_ref) begin
                $display("ERROR: Output -full- equals %0b, but should equal %0b. \t\t--time: %0t", f_txn.full, full_ref, $time);
                error_count_full++;
            end
            else correct_count_full++;

            if(f_txn.empty != empty_ref) begin
                $display("ERROR: Output -empty- equals %0b, but should equal %0b. \t\t--time: %0t", f_txn.empty, empty_ref, $time);
                error_count_empty++;
            end
            else correct_count_empty++;
```



```systemverilog
        task reference_model(FIFO_transaction f_txn, output bit wr_ack_ref, full_ref, empty_ref, almostfull_ref, almostempty_ref, overflow_ref, underflow_ref, logic [FIFO_WIDTH-1:0] data_out_ref);

            if(!f_txn.rst_n) begin
                mem_q.delete();
                data_out_ref = 0;
                overflow_ref = 0;
                underflow_ref = 0;
                wr_ack_ref = 0;
            end
            else begin
                mem_q_size = mem_q.size();
                fork
                    //Write Operation
                    begin
                        if( (f_txn.wr_en && (mem_q.size() < f_txn.FIFO_DEPTH)) || (f_txn.wr_en && f_txn.rd_en && (mem_q.size() == (f_txn.FIFO_DEPTH))) ) begin
                            mem_q.push_front(f_txn.data_in);
                            wr_ack_ref = 1;
                            mem_size_aftr_wr = mem_q.size();
                        end
                        else begin
                            if(f_txn.wr_en) overflow_ref = 1;
                            else overflow_ref = 0;
                            wr_ack_ref = 0;
                        end
                        if(!f_txn.rd_en) data_out_ref = 0;
                    end

                    //Read Operation
                    begin
                        if( (f_txn.rd_en && (mem_q.size() != 0)) || (f_txn.wr_en && f_txn.rd_en && (mem_q.size() == 0)) ) begin
                            data_out_ref = mem_q.pop_back();
                            mem_size_aftr_rd = mem_q.size();
                        end
                        else begin
                            if(f_txn.rd_en) underflow_ref = 1;
                            else underflow_ref = 0;
                            data_out_ref = 0;
                        end
                    end
                join
            end

            if(mem_q.size() == f_txn.FIFO_DEPTH) full_ref = 1;
            else full_ref = 0;

            if(mem_q.size() == (f_txn.FIFO_DEPTH - 1)) almostfull_ref = 1;
            else almostfull_ref = 0;

            if(mem_q.size() == 0) empty_ref = 1;
            else empty_ref = 0;

            if(mem_q.size() == 1) almostempty_ref = 1;
            else almostempty_ref = 0;

        endtask
```

Code Snippets

PACKAGES

SHARED

```systemverilog
package shared_pkg;

    bit test_finished;

    int correct_count_wr_ack, error_count_wr_ack,
        correct_count_full, error_count_full,
        correct_count_empty, error_count_empty,
        correct_count_almostfull, error_count_almostfull,
        correct_count_almostempty, error_count_almostempty,
        correct_count_overflow, error_count_overflow,
        correct_count_underflow, error_count_underflow,
        correct_count_dout, error_count_dout;

endpackage
```

Bug Report

Bug	Original Code	Fix	Lines (Original Version)	Lines (Edited Version)

Sequential Outputs (wr_ack, overflow, underflow, data_out) should reset to zero on reset assetion.	always @(posedge clk or negedge rst_n) begin
	if (!rst_n) begin
	wr_ptr <= 0;
	end
	
	always @(posedge clk or negedge rst_n) begin
	if (!rst_n) begin
	rd_ptr <= 0;
	end
		always @(posedge fd.clk or negedge fd.rst_n) begin
		if (!fd.rst_n) begin
		wr_ptr <= 0;
		fd.overflow <= 0;
		fd.wr_ack <= 0;
		end
		
		always @(posedge fd.clk or negedge fd.rst_n) begin
		if (!fd.rst_n) begin
		rd_ptr <= 0;
		fd.underflow <= 0;
		fd.data_out <= 0;
		end
			[28:31]
			[46:49]
				[90:95]
				[117:122]

When both wr_en and rd_en are high, a read and a write operation should occur in parallel (Even if FIFO is full or empty).
If FIFO full: read op - empty address - write op.
If FIFO empty: write op - filled address - read op.
 	
overflow and underflow should be low in these cases.
	else if (wr_en && count < FIFO_DEPTH) begin
	mem[wr_ptr] <= data_in;
	wr_ack <= 1;
	wr_ptr <= wr_ptr + 1;
	end
	
	else if (rd_en && count != 0) begin
	data_out <= mem[rd_ptr];
	rd_ptr <= rd_ptr + 1;
	end
		else begin
		//wr_en high and FIFO not full - or - wr_en and rd_en are high (read and write in parallel)
		if ((fd.wr_en && (count < (fd.FIFO_DEPTH))) | | (fd.wr_en && fd.rd_en /*&& (count == (fd.FIFO_DEPTH))*/)) begin
		mem[wr_ptr] <= fd.data_in;
		fd.wr_ack <= 1;
		wr_ptr <= wr_ptr + 1;
		fd.overflow <= 0;
		end
		
		//rd_en high and FIFO not empty - or - rd_en and wr_en are high (read and write in parallel)
		else if (fd.rd_en && count != 0 | | (fd.wr_en && fd.rd_en/* && (count == 0)*/)) begin
		//if FIFO is empty: wr_en and rd_en are active (read and write in parallel)
		if((fd.wr_en /*&& fd.rd_en*/ && (count == 0))) fd.data_out <= fd.data_in;
		else fd.data_out <= mem[rd_ptr];
		rd_ptr <= rd_ptr + 1;
		fd.underflow <= 0;
		end
			[32:36]
			[50:53]
				[96:103]
				[123:130]

data_out should be low except when reading data.		if (!fd.rd_en) fd.data_out <= 0;
		
		fd.data_out <= 0;
				113
				137

underflow is a sequential output not a combinational one.	assign underflow = (empty && rd_en)? 1 : 0;	else begin
		if(fd.empty && fd.rd_en)
		fd.underflow <= 1;
		else
		fd.underflow <= 0;
			70	[131:135]

almostfull should be high when count equals (FIFO_DEPTH - 1) not (FIFO_DEPTH - 2).	assign almostfull = (count == FIFO_DEPTH-2)? 1 : 0;	assign fd.almostfull = (count == fd.FIFO_DEPTH-1)? 1 : 0;	71	155

CLOSER LOOK

Bug	Original Code	Fix

Sequential Outputs (wr_ack, overflow, underflow, data_out) should reset to zero on reset assetion.	always @(posedge clk or negedge rst_n) begin
	if (!rst_n) begin
	wr_ptr <= 0;
	end
	
	always @(posedge clk or negedge rst_n) begin
	if (!rst_n) begin
	rd_ptr <= 0;
	end
		always @(posedge fd.clk or negedge fd.rst_n) begin
		if (!fd.rst_n) begin
		wr_ptr <= 0;
		fd.overflow <= 0;
		fd.wr_ack <= 0;
		end
		
		always @(posedge fd.clk or negedge fd.rst_n) begin
		if (!fd.rst_n) begin
		rd_ptr <= 0;
		fd.underflow <= 0;
		fd.data_out <= 0;
		end

When both wr_en and rd_en are high, a read and a write operation should occur in parallel (Even if FIFO is full or empty).
If FIFO full: read op - empty address - write op.
If FIFO empty: write op - filled address - read op.
 	
overflow and underflow should be low in these cases.
	else if (wr_en && count < FIFO_DEPTH) begin
	mem[wr_ptr] <= data_in;
	wr_ack <= 1;
	wr_ptr <= wr_ptr + 1;
	end
	
	else if (rd_en && count != 0) begin
	data_out <= mem[rd_ptr];
	rd_ptr <= rd_ptr + 1;
	end
		else begin
		//wr_en high and FIFO not full - or - wr_en and rd_en are high (read and write in parallel)
		if ((fd.wr_en && (count < (fd.FIFO_DEPTH))) | | (fd.wr_en && fd.rd_en /*&& (count == (fd.FIFO_DEPTH))*/)) begin
		mem[wr_ptr] <= fd.data_in;
		fd.wr_ack <= 1;
		wr_ptr <= wr_ptr + 1;
		fd.overflow <= 0;
		end
		
		//rd_en high and FIFO not empty - or - rd_en and wr_en are high (read and write in parallel)
		else if (fd.rd_en && count != 0 | | (fd.wr_en && fd.rd_en/* && (count == 0)*/)) begin
		//if FIFO is empty: wr_en and rd_en are active (read and write in parallel)
		if((fd.wr_en /*&& fd.rd_en*/ && (count == 0))) fd.data_out <= fd.data_in;
		else fd.data_out <= mem[rd_ptr];
		rd_ptr <= rd_ptr + 1;
		fd.underflow <= 0;
		end

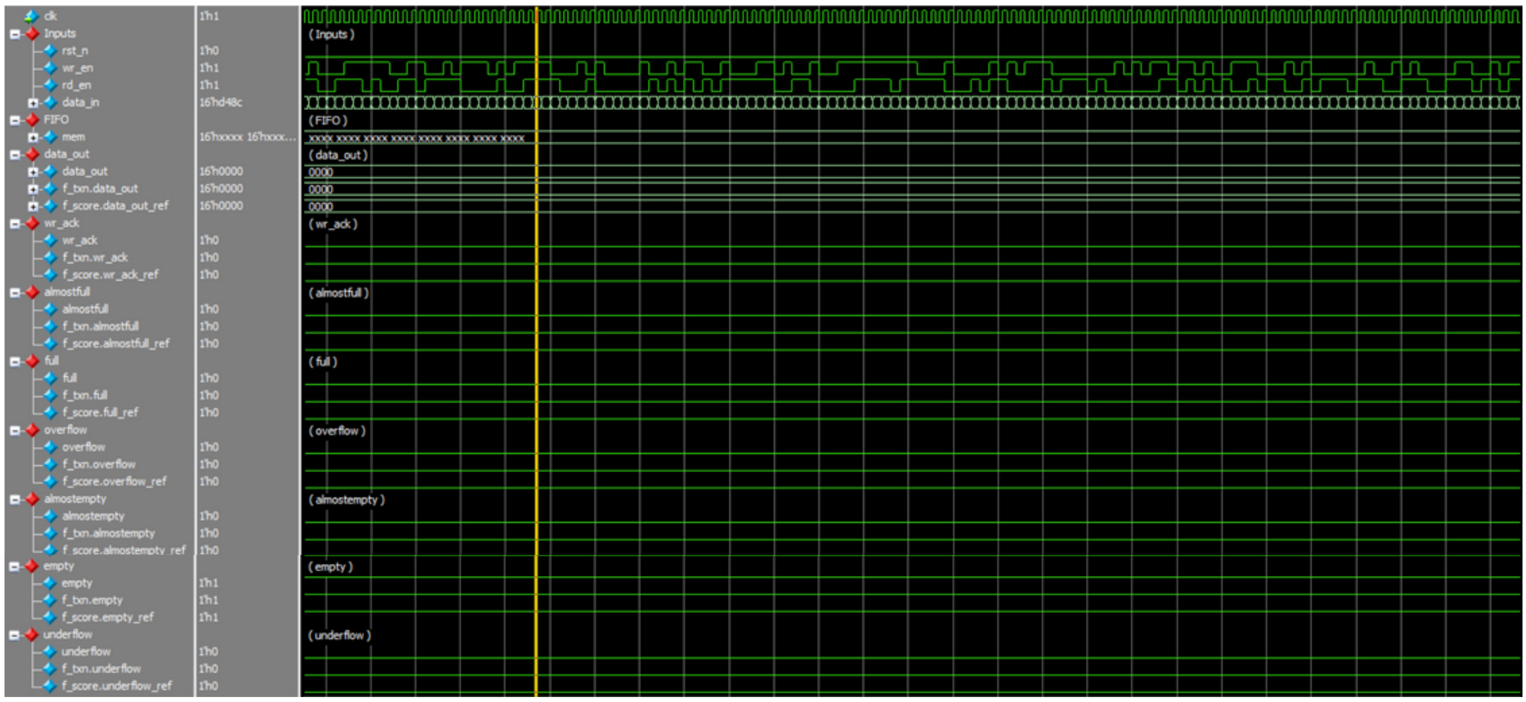
data_out should be low except when reading data.		if (!fd.rd_en) fd.data_out <= 0;
		
		fd.data_out <= 0;

underflow is a sequential output not a combinational one.	assign underflow = (empty && rd_en)? 1 : 0;	else begin
		if(fd.empty && fd.rd_en)
		fd.underflow <= 1;
		else
		fd.underflow <= 0;

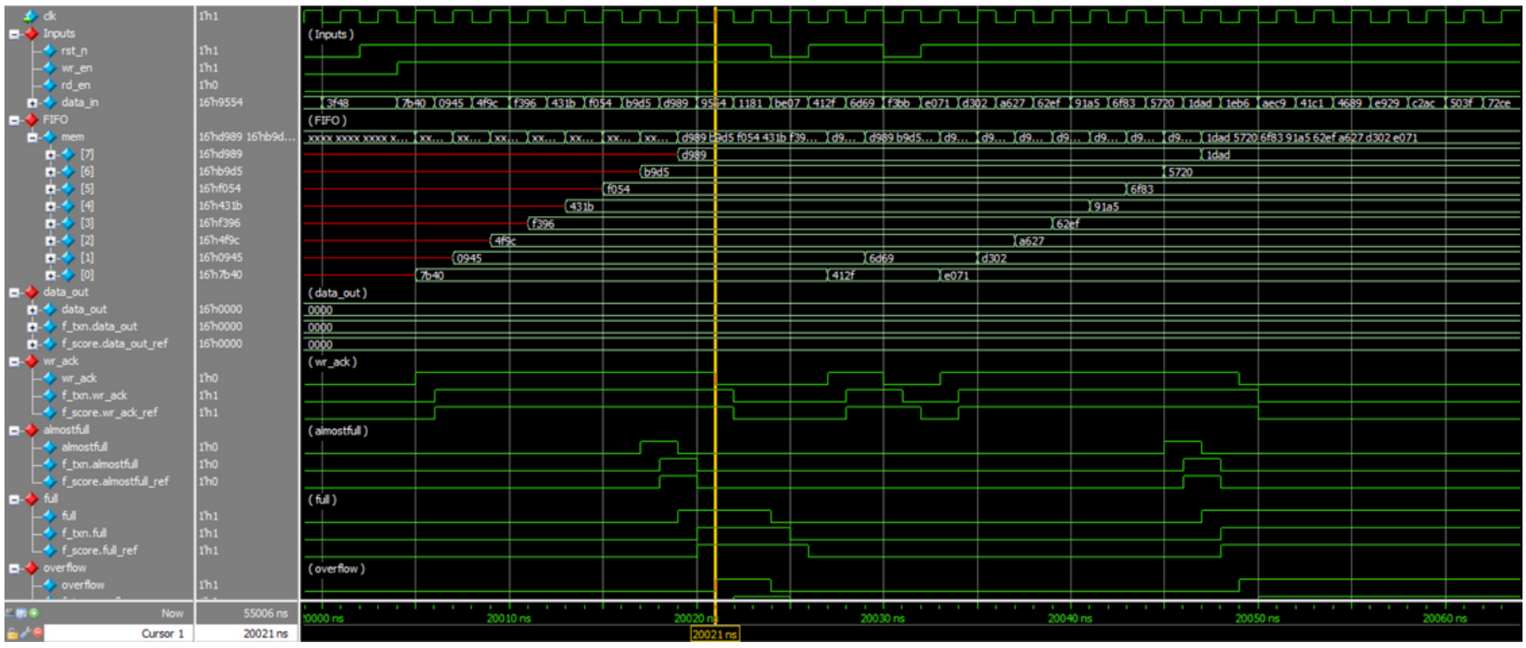
almostfull should be high when count equals (FIFO_DEPTH - 1) not (FIFO_DEPTH - 2).	assign almostfull = (count == FIFO_DEPTH-2)? 1 : 0;	assign fd.almostfull = (count == fd.FIFO_DEPTH-1)? 1 : 0;

Waveform Snippets

Reset Active: empty = 1, all other outputs equal zero

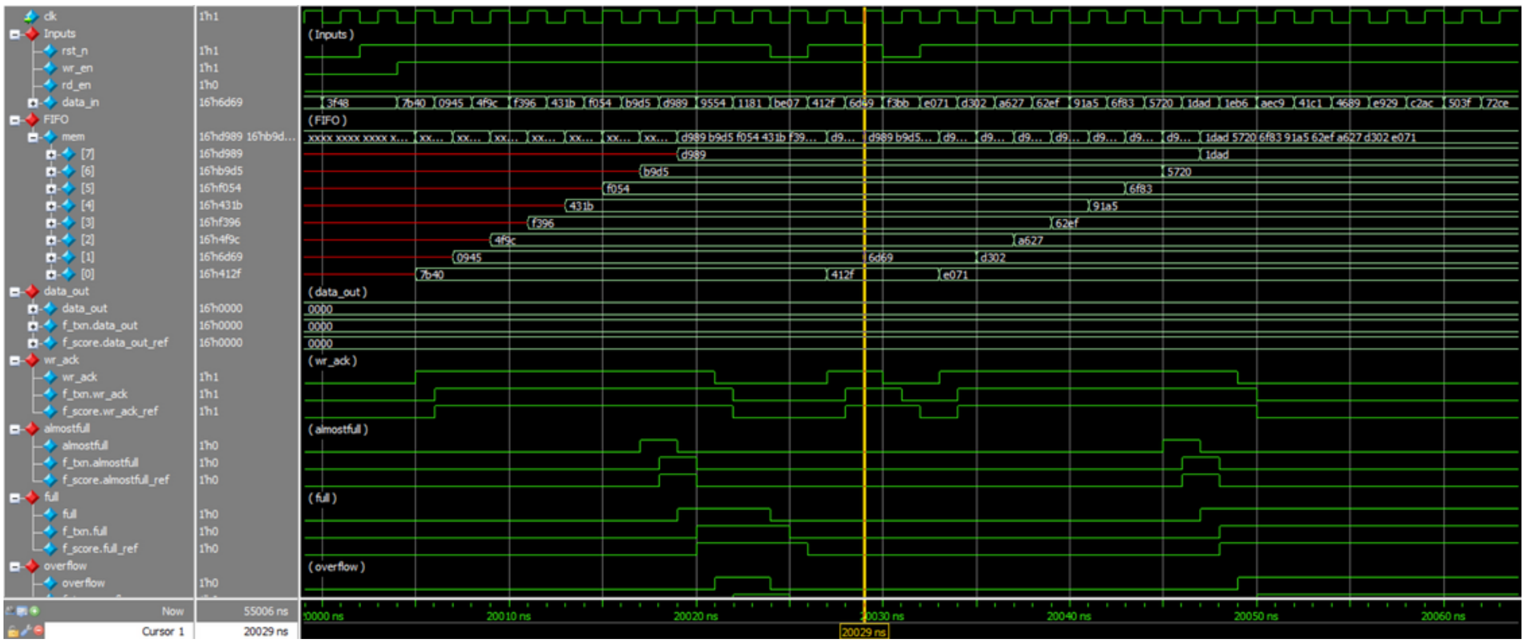


wr_en = 1, rd_en = 0: Write Op untill FIFO is full then overflow is high if wr_en is still high

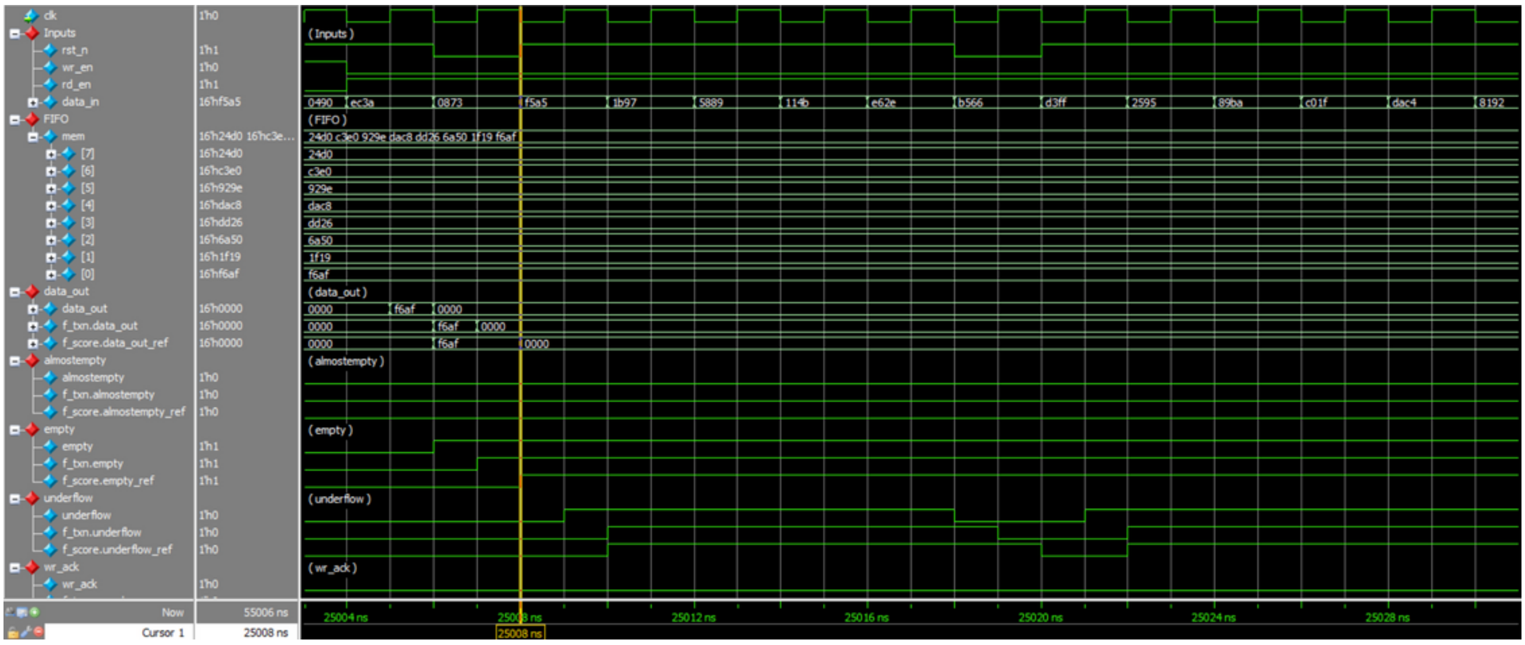


Waveform Snippets

After reset, FIFO is writable and data_in starts filling FIFO addresses starting from 0



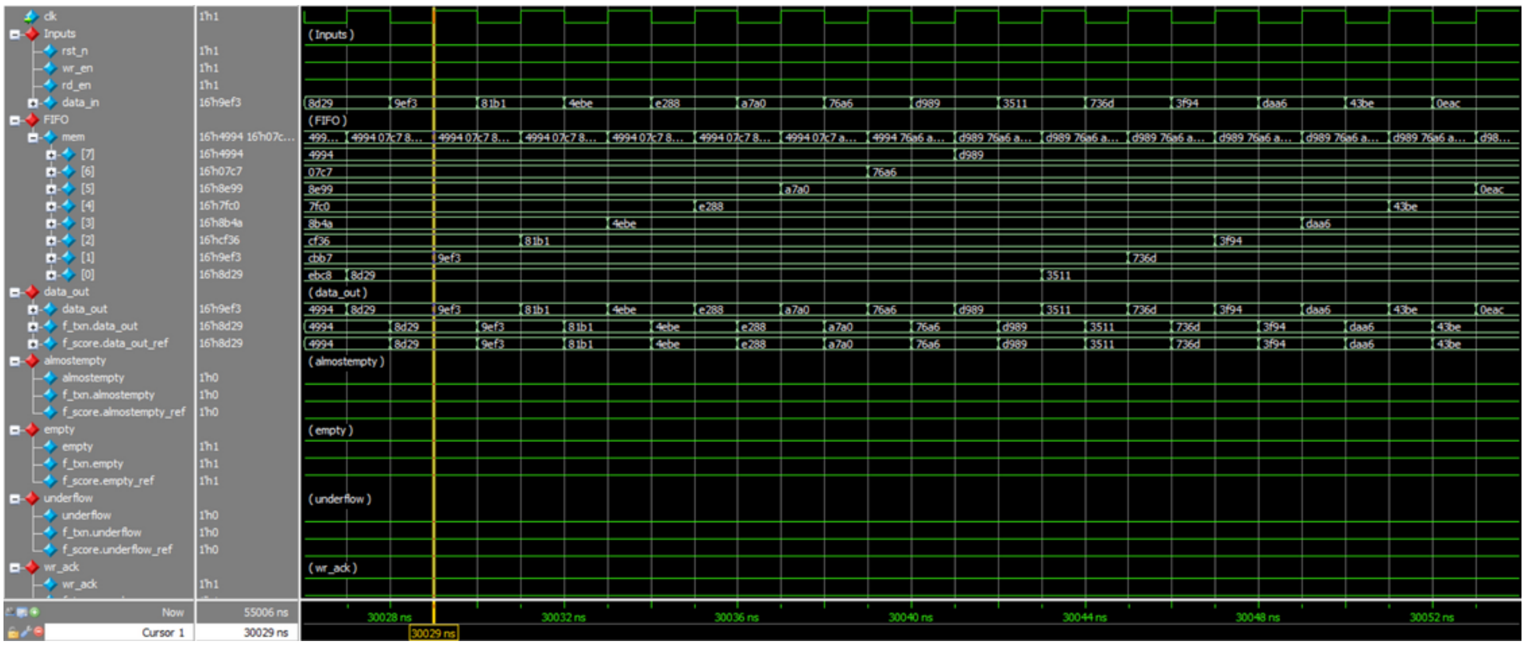
rd_en = 1, wr_en = 0: Read Op until FIFO is empty then underflow is high if rd_en is still high



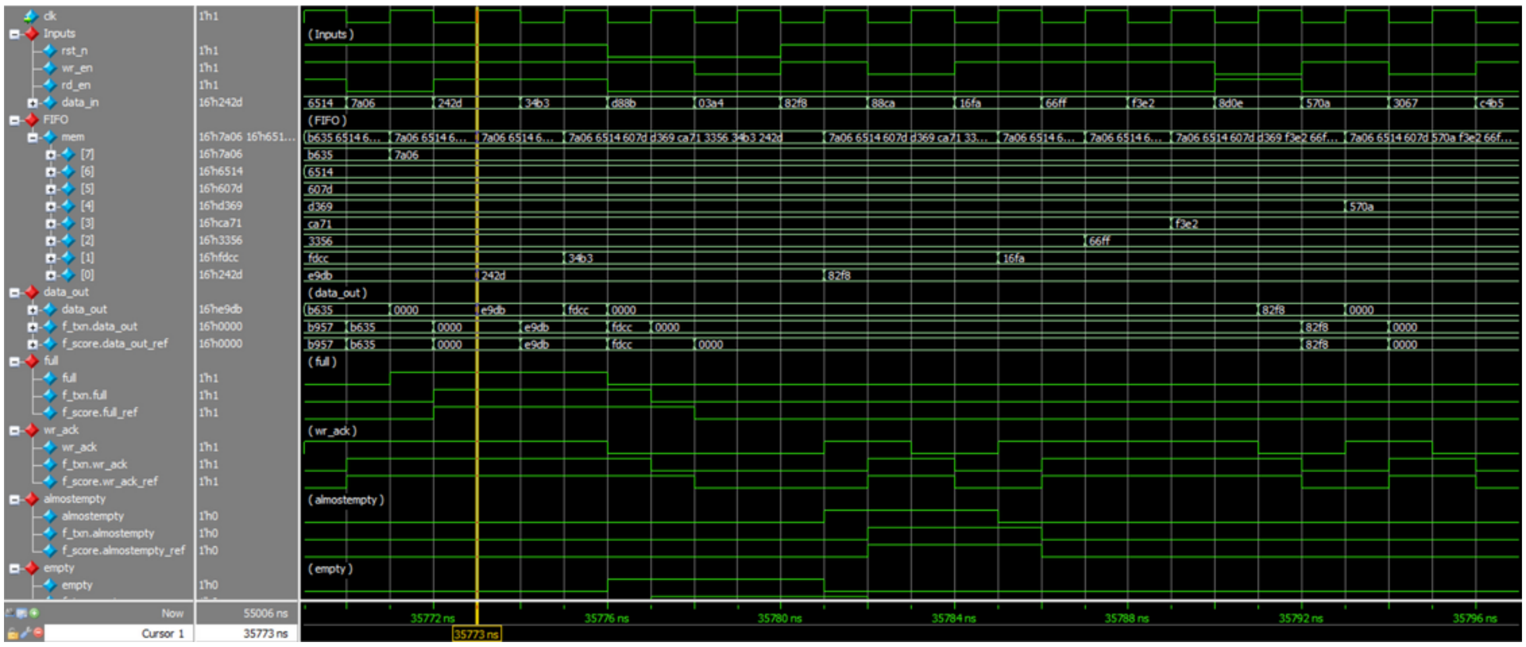
FIFO is empty on reset assertion, so data has to be written in the FIFO for a read op to occur

Waveform Snippets

wr_en = 1, rd_en = 1: As FIFO is now empty any data written (in data_in) is read out (in data_out)

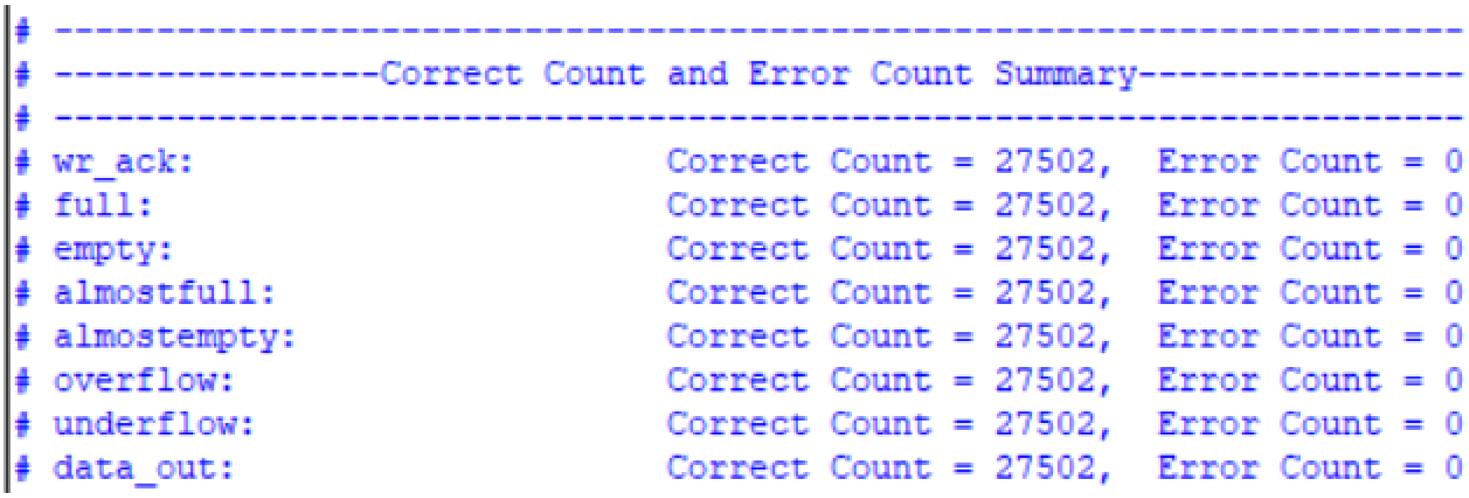


wr_en = 1, rd_en = 1: As FIFO is now full data is read out (in data_out) then the now-empty-address takes the value of data_in



Waveform Snippets

Correct Count and Error Count Summary



```
# ----------------------------------------------------------------------
# ----------------Correct Count and Error Count Summary----------------
# ----------------------------------------------------------------------
# wr_ack:                     Correct Count = 27502,  Error Count = 0
# full:                       Correct Count = 27502,  Error Count = 0
# empty:                      Correct Count = 27502,  Error Count = 0
# almostfull:                 Correct Count = 27502,  Error Count = 0
# almostempty:                Correct Count = 27502,  Error Count = 0
# overflow:                   Correct Count = 27502,  Error Count = 0
# underflow:                  Correct Count = 27502,  Error Count = 0
# data_out:                   Correct Count = 27502,  Error Count = 0
```

Coverage Report

```
Assertion Coverage:
    Assertions                     11         11          0   100.00%
```

```
Directive Coverage:
    Directives                     11         11          0   100.00%
```

```
Branch Coverage:
    Enabled Coverage              Bins       Hits     Misses  Coverage
    ----------------              ----       ----     ------  --------
    Branches                        29         29          0   100.00%
```

```
Condition Coverage:
    Enabled Coverage              Bins    Covered     Misses  Coverage
    ----------------              ----       ----     ------  --------
    Conditions                      20         20          0   100.00%
```

```
Statement Coverage:
    Enabled Coverage              Bins       Hits     Misses  Coverage
    ----------------              ----       ----     ------  --------
    Statements                      31         31          0   100.00%
```

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

```
Covergroup Coverage:
    Covergroups                      1         na         na   100.00%
        Coverpoints/Crosses         28         na         na        na
            Covergroup Bins         98         98          0   100.00%
```