FIFO Verification

Using SystemVerilog

Mahmoud Ghamry

SEP 2024

Table of Contents

1.Introduction

First-In-First-Out (FIFO) buffers are fundamental components in digital systems, serving as temporary storage areas that manage the flow of data between producer and consumer processes. Their inherent design allows data to be stored in the order it is received, ensuring that the first data to enter the buffer is the first to be retrieved. This characteristic makes FIFOs essential in various applications, including data streaming, buffering between asynchronous systems, and handling variable data rates.

As the complexity of digital systems increases, ensuring the correctness and reliability of FIFO implementations becomes paramount. Verification is a critical phase in the design process, aimed at identifying and resolving potential issues before deployment. A rigorous verification strategy not only validates the functional correctness of the FIFO but also checks for edge cases, performance metrics, and adherence to specifications.

This document explores the methodologies employed in verifying FIFO designs using SystemVerilog, a powerful hardware description and verification language. By leveraging SystemVerilog's capabilities, we can create comprehensive testbenches, cover a wide range of scenarios, and ensure the FIFO operates reliably under all conditions. Through systematic testing and validation, we aim to guarantee that the FIFO meets its design goals and performs efficiently in real-world applications.

2.**FIFO Architecture**

The architecture of a FIFO buffer typically consists of several key components that work together to facilitate data storage, retrieval, and management. Below is a description of the primary elements that make up a FIFO design:

1. Data Storage Array

- Description: The core component of the FIFO is a memory array that holds the data elements. This array is usually implemented using registers or RAM blocks.
- Functionality: The size of the array determines the capacity of the FIFO. The data is written to the next available location in the array and read from the oldest location.

2. Write Pointer

- Description: A pointer that indicates the next available position in the FIFO for writing new data.
- Functionality: The write pointer increments each time data is written, wrapping around to the beginning of the array when it reaches the end (circular buffer behavior).

3. Read Pointer

- Description: A pointer that indicates the location of the next data element to be read from the FIFO.
- Functionality: Similar to the write pointer, the read pointer increments each time data is read, wrapping around when reaching the end of the array.

4. Control Logic

- Description: A set of logic circuits that manage the behavior of the FIFO, including writing, reading, overflow, and underflow conditions.
- Functionality:
    - Write Enable (wr_en): Signals whether a write operation can occur.
    - Read Enable (rd_en): Signals whether a read operation can occur.

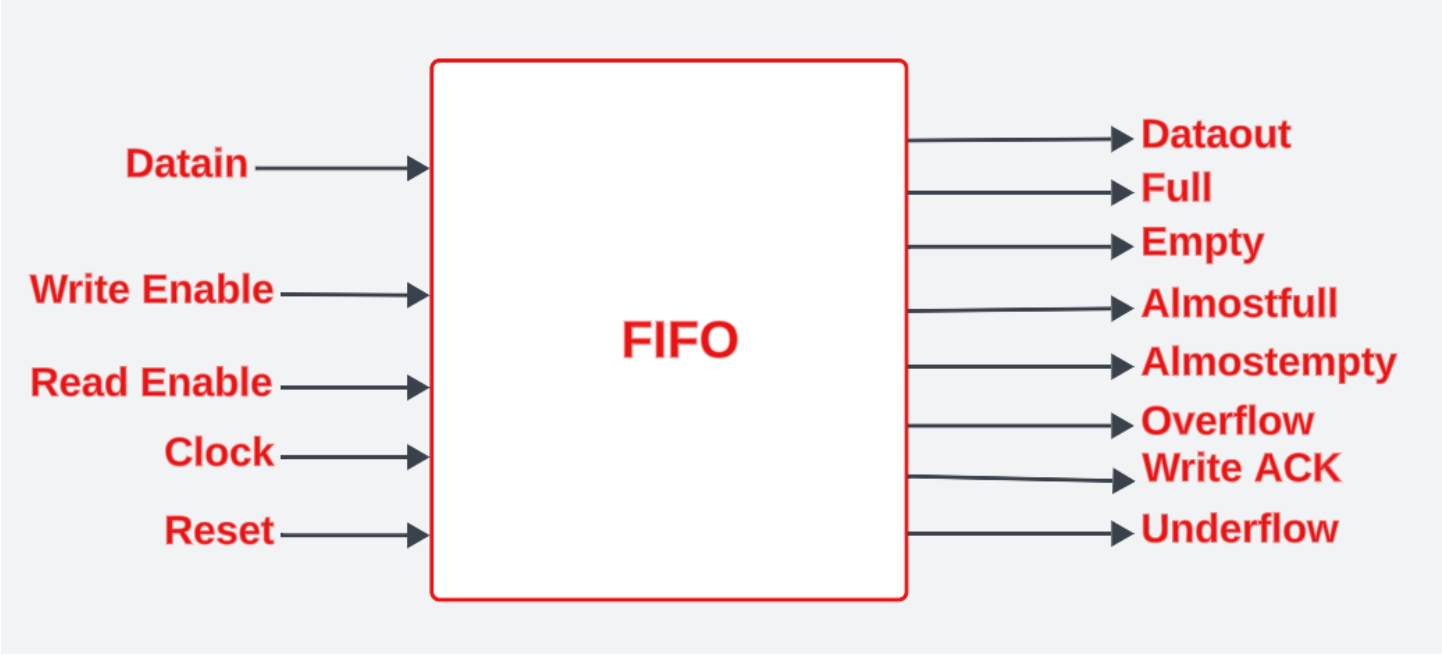
- Full and Empty Flags: Indicators that signal whether the FIFO is full or empty, respectively. These flags are essential for preventing overflow and underflow conditions.

5. Status Indicators

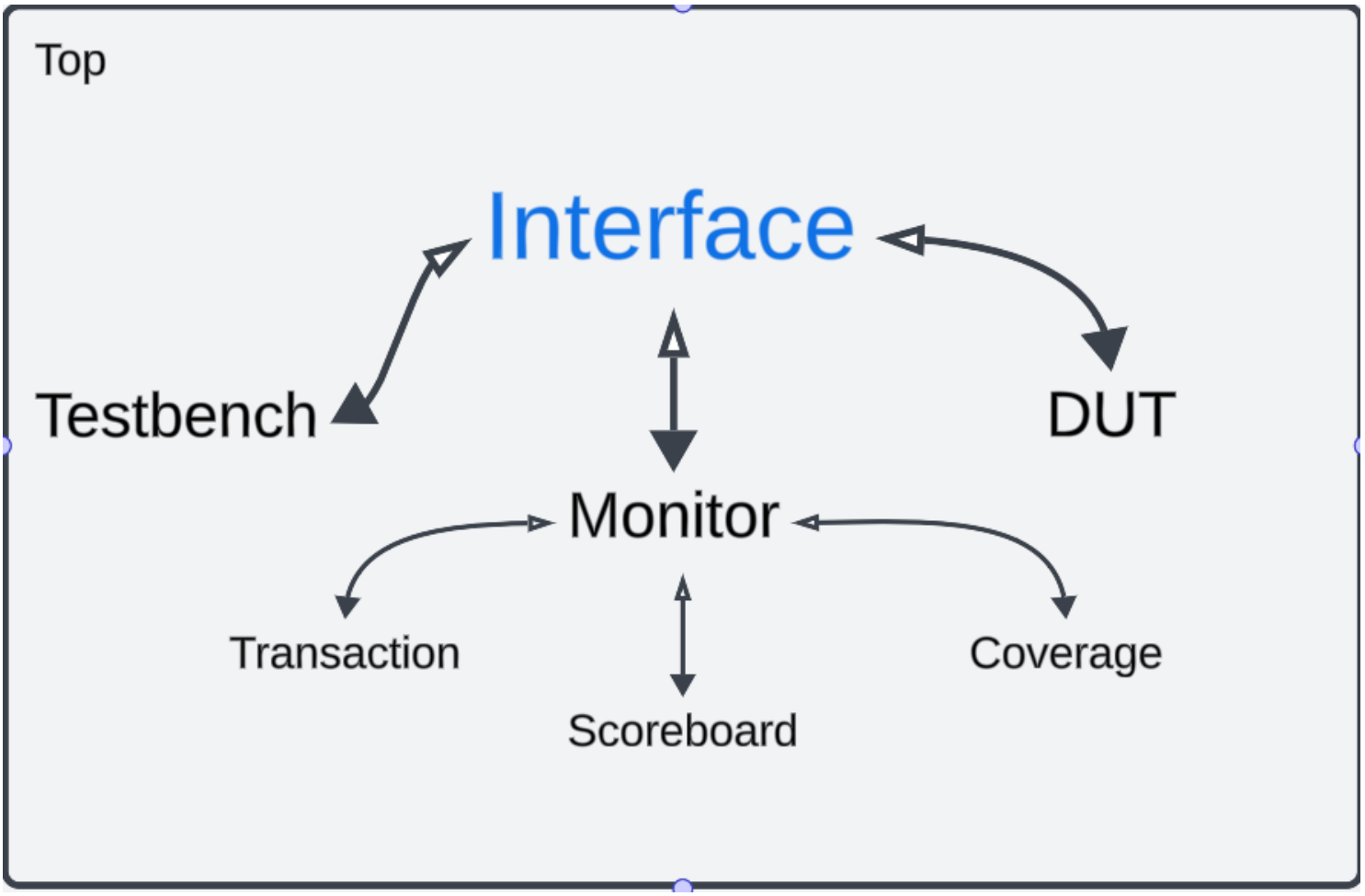
- Description: Additional signals that provide information about the FIFO's state.
- Functionality:
  - Full Flag: Set when the write pointer reaches the read pointer after wrapping around.
  - Empty Flag: Set when the write pointer equals the read pointer, indicating no data is available to read.
  - Almost Full / Almost Empty Flags: These may be included to provide early warning conditions for high-level control mechanisms.

6. Data Input/Output Interfaces

- Description: Interfaces for connecting the FIFO to other components in the system.
- Functionality: This includes input lines for writing data into the FIFO and output lines for reading data out of the FIFO.



3. Verification Flow



4. Verification Plan

	Description	Stimulus Generation	Functional Coverage	Functionality Check
FIFO_1	When the reset is asserted, the output C value not change and empty flag is high	Directed at the start of the simulation then randomzied with constraint that drive the reset to off most the simulation time	-	A checker in the testbench to make sure the output is correct and concurrent assertion to check the empty
FIFO_2	when the write and read asserted and the fifo is empty ,the write is higher priority so the output value for almostempty is high	Randomization under constraint on the write enable to be high randomized from the simulation time	cover all values of flags	A checker in the testbench to make sure the output is correct and concurrent assertion to check the almostempty
FIFO_3	when the write and read asserted and the fifo is not empty nor full so the output value make the two order in the same time	Randomization under constraint on the write enable to be high randomized from the simulation time	cover all values of flags	A checker in the testbench to make sure the output is correct
FIFO_4	when the write and read asserted and the fifo is full ,the read is higher priority so the output value for almostfull is high	Randomization under constraint on the write enable to be high randomized from the simulation time	cover all values of flags	A checker in the testbench to make sure the output is correct and concurrent assertion to check the almostfull
FIFO_5	when the write is high and read is low and the fifo is full , so the output value for overflow is high	Randomization under constraint on the write enable to be high randomized from the simulation time	cover all values of flags	A checker in the testbench to make sure the output is correct and concurrent assertion to check the overflow
FIFO_6	when the write islow and read is high and the fifo is full , so the output value for underflow is high	Randomization under constraint on the write enable to be high randomized from the simulation time	cover all values of flags	A checker in the testbench to make sure the output is correct and concurrent assertion to check the underflow

5.Design

```
module FIFO(data_in, wr_en, rd_en, clk, rst_n, full, empty, almostfull, almostempty, wr_ack, overflow, underflow, data_out);
parameter FIFO_WIDTH = 16;
parameter FIFO_DEPTH = 8;
input [FIFO_WIDTH-1:0] data_in;
input clk, rst_n, wr_en, rd_en;
output reg [FIFO_WIDTH-1:0] data_out;
output reg wr_ack, overflow;
output full, empty, almostfull, almostempty, underflow;

localparam max_fifo_addr = $clog2(FIFO_DEPTH);

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

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

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
```

6.Bugs

- The underflow flag should be registered.
- Reset the flags (empty – almostempty – wr_ack – full – alomstfull).
- Add cases
    - If the wr_en , rd_en and empty is high so the write is the higher priority and counter decreases .
    - If the wr_en , rd_en and full is high so the Read is the higher priority and counter increases .
    - If the wr_en and rd_en is high and not full nor empty so the two cases run together but the counter not changed.
    - If the wr_en and rd_en is low so everything not changed.

- The almostfull must be high if the FIFO need one input to be high So I edit it

7. Verification Files

7.1 interface

```
interface fifo(clk);

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

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

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


endinterface
```

7.2 Share_pkg

```
package share_pkg;

integer error_count = 0 ;
integer correct_count = 0 ;
bit test_finsh = 0 ;

endpackage
```

7.3 Top

```
module top;

bit clk;

initial begin
    forever begin
        #10 clk =~clk;
    end
end

fifo vif(clk);
FIFO_tb tb(vif);
FIFO dut(vif);
fifo_monitor mon(vif);



endmodule
```

7.4 Transaction

```
package fifo_pkg;
    class FIFO_transaction;


    parameter FIFO_WIDTH = 16;
    bit clk;
    logic data_in;
    rand logic  rst_n, wr_en, rd_en;
    logic [FIFO_WIDTH-1:0] data_out;
    logic  wr_ack, overflow,full, empty, almostfull, almostempty, underflow;

    rand int WR_EN_ON_DIST;
    rand int RD_EN_ON_DIST;

    function new(int WR_EN_ON_DIST = 70   , int RD_EN_ON_DIST = 30 );
        this.WR_EN_ON_DIST=WR_EN_ON_DIST;
        this.RD_EN_ON_DIST=RD_EN_ON_DIST;
    endfunction

    constraint assert_reset{
        rst_n dist{0:=2,1:=98};
    }

    constraint write_enable{
        wr_en dist{1:=WR_EN_ON_DIST,0:=100-WR_EN_ON_DIST};
    }

    constraint read_enable{
        rd_en dist{1:=RD_EN_ON_DIST,0:=100-RD_EN_ON_DIST};
    }

    endclass
endpackage
```

7.5 Testbench

```
import share_pkg::*;
import fifo_pkg::*;
module FIFO_tb(fifo.TEST vif);

FIFO_transaction q = new () ;

initial begin

        vif.rst_n=0;
        @(negedge vif.clk);
        vif.rst_n=1;
    for (int i = 0  ; i < 30000 ; i++ ) begin

        @(negedge vif.clk);

        assert(q.randomize());

        vif.rst_n=q.rst_n;
        vif.wr_en=q.wr_en;
        vif.rd_en=q.rd_en;
        vif.data_in=i;


    end
    test_finsh = 1 ;

end

endmodule
```

7.6 Monitor

```
import fifo_pkg::*;
import fifo_scoreboard::*;
import package_fifo::*;
import share_pkg::*;
module fifo_monitor(fifo.MONITOR vif);

FIFO_transaction q = new () ;
fifo_scoreboard score  = new();
FIFO_coverage cove = new();

initial begin

    forever begin
    @(negedge vif.clk);
    q.rst_n=vif.rst_n;
    q.wr_en=vif.wr_en;
    q.rd_en=vif.rd_en;
    q.data_in=vif.data_in;
    q.data_out=vif.data_out;
    q.wr_ack=vif.wr_ack;
    q.full=vif.full;
    q.empty=vif.empty;
    q.almostfull=vif.almostfull;
    q.almostempty=vif.almostempty;
    q.underflow=vif.underflow;
    q.overflow=vif.overflow;
    q.clk=vif.clk;
```

```
    fork
        begin
          cove.sample_data(q);
        end
        begin
            @(posedge vif.clk);
           score.check_data(q);

        end
    join
    if (test_finsh) begin
        $display("END OF SIMULATION ");
        $display("ERROR COUNT IS %0d , CORRECT COUNT IS %0d ",error_count,correct_count);
        $stop;
    end

    end
end
endmodule
```

7.7 Coverage

```
package package_fifo;
    import fifo_pkg::*;
    class FIFO_coverage;
        FIFO_transaction F_cvg_txn;

        covergroup g1;
        write :     coverpoint F_cvg_txn.wr_en;
        read:       coverpoint F_cvg_txn.rd_en;
        write_ack:  coverpoint F_cvg_txn.wr_ack;
        OVERFFLOW:  coverpoint F_cvg_txn.overflow;
        UNDERFLOW:  coverpoint F_cvg_txn.underflow;
        FULL:       coverpoint F_cvg_txn.full;
        EMPTY:      coverpoint F_cvg_txn.empty;
        Almost_empty:coverpoint F_cvg_txn.almostempty;
        Almost_full:coverpoint F_cvg_txn.almostfull;

        crossofwriteack : cross write,read,write_ack{
            ignore_bins rd = binsof(write)intersect{0}&&binsof(read)intersect{1}&&binsof(write_ack)intersect{1};}
        crossoffull     : cross write,read,FULL{
            ignore_bins rd_full = binsof(write)intersect{0}&&binsof(read)intersect{1}&&binsof(FULL)intersect{1};
            ignore_bins wr_rd_full = binsof(write)intersect{1}&&binsof(read)intersect{1}&&binsof(FULL)intersect{1};}
        crossofempty     : cross write,read,EMPTY{
            ignore_bins wr_empty = binsof(write)intersect{1}&&binsof(read)intersect{0}&&binsof(EMPTY)intersect{1};
            ignore_bins w_r_empty = binsof(write)intersect{1}&&binsof(read)intersect{1}&&binsof(EMPTY)intersect{1};}
        crossofOVERFFLOW     : cross write,read,OVERFFLOW{
            ignore_bins rd_overflow = binsof(write)intersect{0}&&binsof(read)intersect{1}&&binsof(OVERFFLOW)intersect{1};
            ignore_bins w_r_overflow = binsof(write)intersect{1}&&binsof(read)intersect{1}&&binsof(OVERFFLOW)intersect{1};}
        crossofUNDERFLOW     : cross write,read,UNDERFLOW{
            ignore_bins wr_underflow = binsof(write)intersect{1}&&binsof(read)intersect{0}&&binsof(UNDERFLOW)intersect{1};
            ignore_bins wr_ed_underflow = binsof(write)intersect{1}&&binsof(read)intersect{1}&&binsof(UNDERFLOW)intersect{1};}
        crossofAlmost_empty     : cross write,read,Almost_empty;
        crossofAlmost_full     : cross write,read,Almost_full;

        endgroup
```

```
        function new();
            g1=new();
        endfunction

        function void sample_data(FIFO_transaction F_txn );
            this.F_cvg_txn = F_txn;
            g1.sample();
        endfunction
    endclass


endpackage
```

7.8 Scoreboard

```
package fifo_scoreboard;
    import fifo_pkg::*;
    import share_pkg::*;

    parameter FIFO_WIDTH = 16;
    parameter FIFO_DEPTH = 8;

    class fifo_scoreboard;

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

    localparam max_fifo_addr = $clog2(FIFO_DEPTH);

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

    logic [max_fifo_addr-1:0] wr_ptr_ref, rd_ptr_ref;
    logic [max_fifo_addr:0] count_ref;

    function void check_data(FIFO_transaction f_ref);
    reference_model(f_ref);
```

```
    if (
        (wr_ack_ref != f_ref.wr_ack) ||
        (overflow_ref != f_ref.overflow) ||
        (underflow_ref != f_ref.underflow) ||
        (full_ref != f_ref.full) ||
        (empty_ref != f_ref.empty) ||
        (almostempty_ref != f_ref.almostempty) ||
        (almostfull_ref != f_ref.almostfull)) begin

        error_count = error_count + 1;
        $display("ERROR DETECTED!!!");
        $display("Expected values: ");

        $display("  wr_ack         = %0b | Received: %0b", wr_ack_ref, f_ref.wr_ack);
        $display("  overflow       = %0b | Received: %0b", overflow_ref, f_ref.overflow);
        $display("  underflow      = %0b | Received: %0b", underflow_ref, f_ref.underflow);
        $display("  full           = %0b | Received: %0b", full_ref, f_ref.full);
        $display("  empty          = %0b | Received: %0b", empty_ref, f_ref.empty);
        $display("  almostempty    = %0b | Received: %0b", almostempty_ref, f_ref.almostempty);
        $display("  almostfull     = %0b | Received: %0b", almostfull_ref, f_ref.almostfull);

        $stop;
    end else begin
        correct_count = correct_count + 1;

    end
endfunction
```

```
function void reference_model(FIFO_transaction f_ref);

    if (!f_ref.rst_n) begin


        full_ref = 0;
        empty_ref = 1;
        almostfull_ref = 0;
        almostempty_ref = 0;
        overflow_ref = 0;
        underflow_ref = 0;
        wr_ack_ref = 0 ;
        wr_ptr_ref = 0 ;
        rd_ptr_ref = 0 ;
        count_ref = 0 ;
    end
    else begin

        if (({f_ref.wr_en, f_ref.rd_en} == 2'b11) && empty_ref  ) begin
        mem[wr_ptr_ref] = f_ref.data_in;
        wr_ack_ref = 1;
        wr_ptr_ref = wr_ptr_ref + 1;

    end
    else if (({f_ref.wr_en, f_ref.rd_en} == 2'b11) && !empty_ref && !full_ref ) begin
        mem[wr_ptr_ref] = f_ref.data_in;
        wr_ack_ref = 1;
        wr_ptr_ref = wr_ptr_ref + 1;

    end
```

```
else if (({f_ref.wr_en, f_ref.rd_en}  == 2'b10) && count_ref < FIFO_DEPTH) begin
    mem[wr_ptr_ref] = f_ref.data_in;
    wr_ack_ref = 1;
    wr_ptr_ref = wr_ptr_ref + 1;
end
else if ({f_ref.wr_en, f_ref.rd_en}  == 2'b00) begin

end
else begin
    wr_ack_ref = 0;
    if (full_ref && f_ref.wr_en&&!f_ref.rd_en) //3
        overflow_ref = 1;
    else
        overflow_ref = 0;
end

 if ((({f_ref.wr_en, f_ref.rd_en} == 2'b11) && full_ref ) ) begin
    data_out_ref = mem[rd_ptr_ref];
    rd_ptr_ref = rd_ptr_ref + 1;
end
else if ((({f_ref.wr_en, f_ref.rd_en} == 2'b11) && !full_ref &&!empty_ref ) ) begin
    data_out_ref = mem[rd_ptr_ref];
    rd_ptr_ref = rd_ptr_ref + 1;
end
else if ((({f_ref.wr_en, f_ref.rd_en} == 2'b11) && !empty_ref ) ) begin
    data_out_ref = mem[rd_ptr_ref];
    rd_ptr_ref = rd_ptr_ref + 1;
    count_ref = count_ref - 1 ;
end
```

```
else if (({f_ref.wr_en, f_ref.rd_en} == 2'b01) && count_ref != 0&&!empty_ref) begin
    data_out_ref = mem[rd_ptr_ref];
    rd_ptr_ref = rd_ptr_ref + 1;
end
else if ({f_ref.wr_en, f_ref.rd_en} == 2'b00) begin

end
else begin
  if ((!empty_ref && ({f_ref.wr_en, f_ref.rd_en} == 2'b11)) ||({f_ref.wr_en, f_ref.rd_en} == 2'b01)&&empty_ref)) begin
    underflow_ref = 1 ;
  end
  else begin
    underflow_ref = 0 ;
  end
end

    if  ( ({f_ref.wr_en, f_ref.rd_en} == 2'b10) && !full_ref)
        count_ref = count_ref + 1;
    else if ( ({f_ref.wr_en, f_ref.rd_en} == 2'b01) && !empty_ref)
        count_ref = count_ref - 1;
    else if (({f_ref.wr_en, f_ref.rd_en} == 2'b11) && full_ref  ) begin
        count_ref = count_ref - 1;
    end
    else if (({f_ref.wr_en, f_ref.rd_en} == 2'b11) && empty_ref ) begin
        count_ref = count_ref + 1;
    end
    else begin

    end
```

```
        full_ref = (count_ref == FIFO_DEPTH)? 1 : 0;
        almostempty_ref = (count_ref == 1)? 1 : 0;
        almostfull_ref = (count_ref == FIFO_DEPTH-1)? 1 : 0;
        empty_ref = (count_ref == 0)? 1 : 0;
    end

    endfunction

    endclass

endpackage
```

7.9 Design After Edit Bugs and Assertion

```
module FIFO(fifo.DUT vif);

localparam max_fifo_addr = $clog2(vif.FIFO_DEPTH);

reg [vif.FIFO_WIDTH-1:0] mem [vif.FIFO_DEPTH-1:0];
reg [max_fifo_addr-1:0] wr_ptr, rd_ptr;
reg [max_fifo_addr:0] count;

always @(posedge vif.clk or negedge vif.rst_n) begin
    if (!vif.rst_n) begin
        wr_ptr <= 0;
        vif.wr_ack   <= 0 ;
        vif.overflow <= 0 ;
    end

    else if (vif.wr_en && count < vif.FIFO_DEPTH) begin
        mem[wr_ptr] <= vif.data_in;
        vif.wr_ack <= 1;
        wr_ptr <= wr_ptr + 1;
    end
```

```
    else if ({vif.wr_en, vif.rd_en} == 2'b00) begin

    end
    else begin
        vif.wr_ack <= 0;
        if (vif.full && vif.wr_en&&!vif.rd_en)
            vif.overflow <= 1;
        else
            vif.overflow <= 0;
    end
end
```

```
always @(posedge vif.clk or negedge vif.rst_n) begin
    if (!vif.rst_n) begin
        rd_ptr <= 0;
        vif.underflow <= 0 ;
    end

    else if (vif.rd_en && count != 0&&!vif.empty) begin
        vif.data_out <= mem[rd_ptr];
        rd_ptr <= rd_ptr + 1;
    end
    else if ({vif.wr_en, vif.rd_en} == 2'b00) begin

    end
    else begin
      if (vif.rd_en&&!vif.wr_en&&vif.empty) begin
        vif.underflow <= 1 ;
      end
      else begin
        vif.underflow <= 0 ;
      end
    end
end
```

```
always @(posedge vif.clk or negedge vif.rst_n) begin
    if (!vif.rst_n) begin
        count <= 0;
    end
    else begin
        if  ( ({vif.wr_en, vif.rd_en} == 2'b10) && !vif.full)
            count <= count + 1;
        else if ( ({vif.wr_en, vif.rd_en} == 2'b01) && !vif.empty)
            count <= count - 1;
        else if (({vif.wr_en, vif.rd_en} == 2'b11) && vif.full  ) begin
            count <= count - 1;
        end
        else if (({vif.wr_en, vif.rd_en} == 2'b11) && vif.empty ) begin
            count <= count + 1;
        end
        else begin

        end
    end
end
```

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

Assertion

```
`ifdef ASSERTIONS_ON


almostempty:assert property (@(posedge vif.clk) (count==1) |-> (vif.almostempty==1));
almostfull:assert property (@(posedge vif.clk) (count == vif.FIFO_DEPTH-1) |-> (vif.almostfull==1));
empty:assert property (@(posedge vif.clk) (count == 0) |-> (vif.empty==1));
full:assert property (@(posedge vif.clk) (count == vif.FIFO_DEPTH) |-> (vif.full==1));
overflow:assert property (@(posedge vif.clk) disable iff(!vif.rst_n)(vif.full && vif.wr_en && !vif.rd_en) |=> vif.overflow == 1)
else $error("Overflow should be 1 when FIFO is full, write enable is asserted, and read enable is not asserted.");
underflow:assert property (@(posedge vif.clk)disable iff(!vif.rst_n) (vif.rd_en&&!vif.wr_en&&vif.empty) |=>vif.underflow==1);


`endif

endmodule
```

Do file

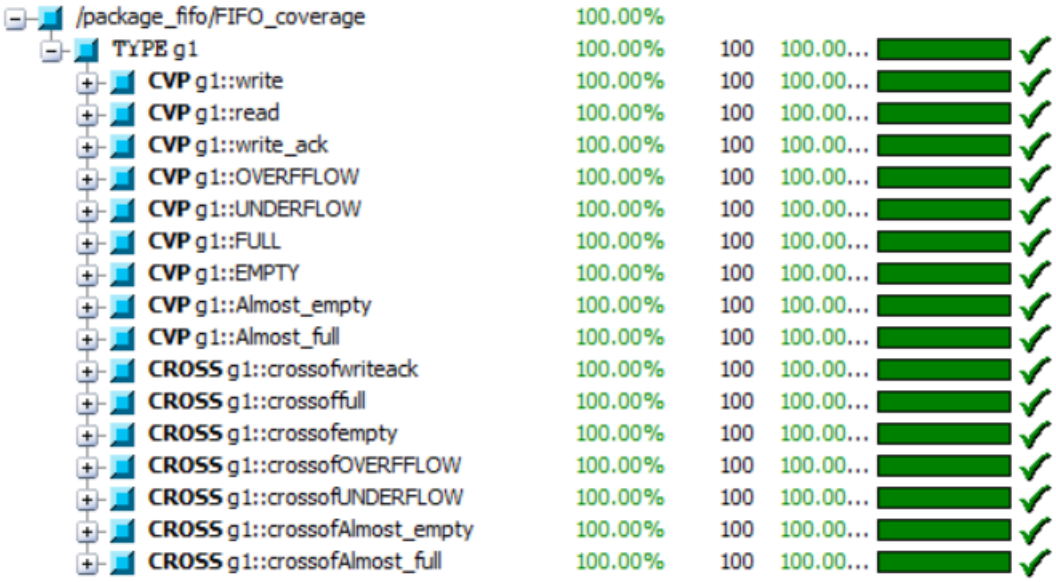
```
vlib work
vlog +define+ASSERTIONS_ON fif0_monitor.sv FIFO.sv FIFO_scoreboard_pkg.sv
 fifo_tb.sv interface.sv package.sv pkg.sv top.sv share_pkg.sv +cover -covercells
vsim -voptargs=+acc work.top -cover -sv_seed wlftikv888
add wave *
add wave -position insertpoint sim:/top/tb/vif/*
coverage save top.ucdb -onexit
run -all
```

8.Result

8.1Trascript

```
END OF SIMULATION
ERROR COUNT IS 0 , CORRECT COUNT IS 30001
```

8.2 Function Coverage



8.3 Assertion


```
Assertion Coverage:
    Assertions                       6         6         0   100.00%
--------------------------------------------------------------------
Name                 File(Line)                   Failure      Pass
                                                  Count        Count
--------------------------------------------------------------------
/top/dut/almostempty FIFO.sv(115)                    0            1
/top/dut/almostfull  FIFO.sv(116)                    0            1
/top/dut/empty       FIFO.sv(117)                    0            1
/top/dut/full        FIFO.sv(118)                    0            1
/top/dut/overflow    FIFO.sv(119)                    0            1
/top/dut/underflow   FIFO.sv(123)                    0            1
```

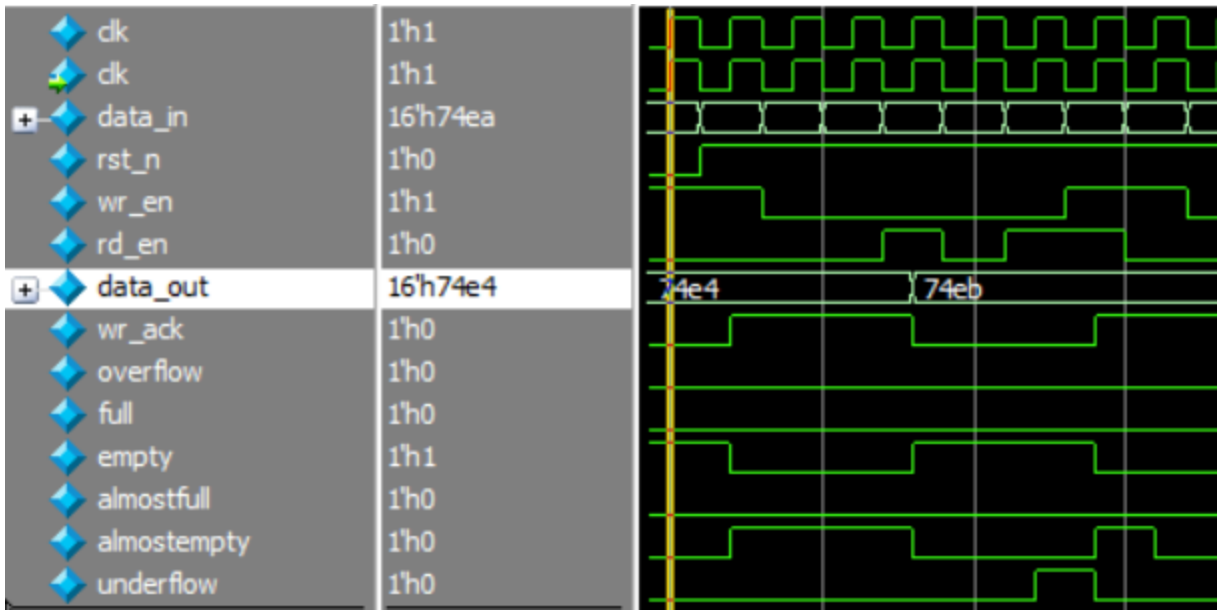
8.4 Cross Coverage

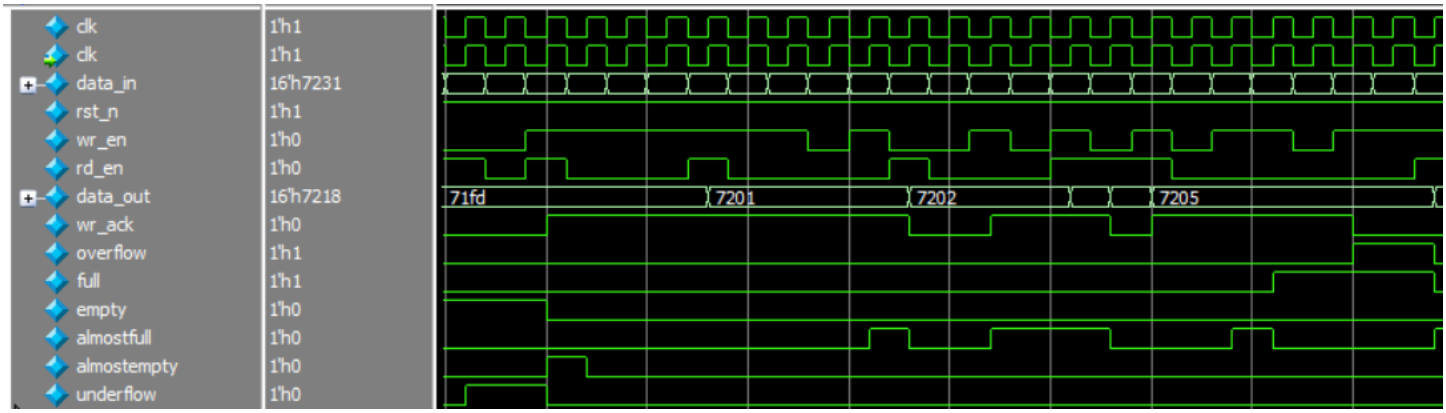
```
Statement Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Statements                      39        39         0   100.00%
```

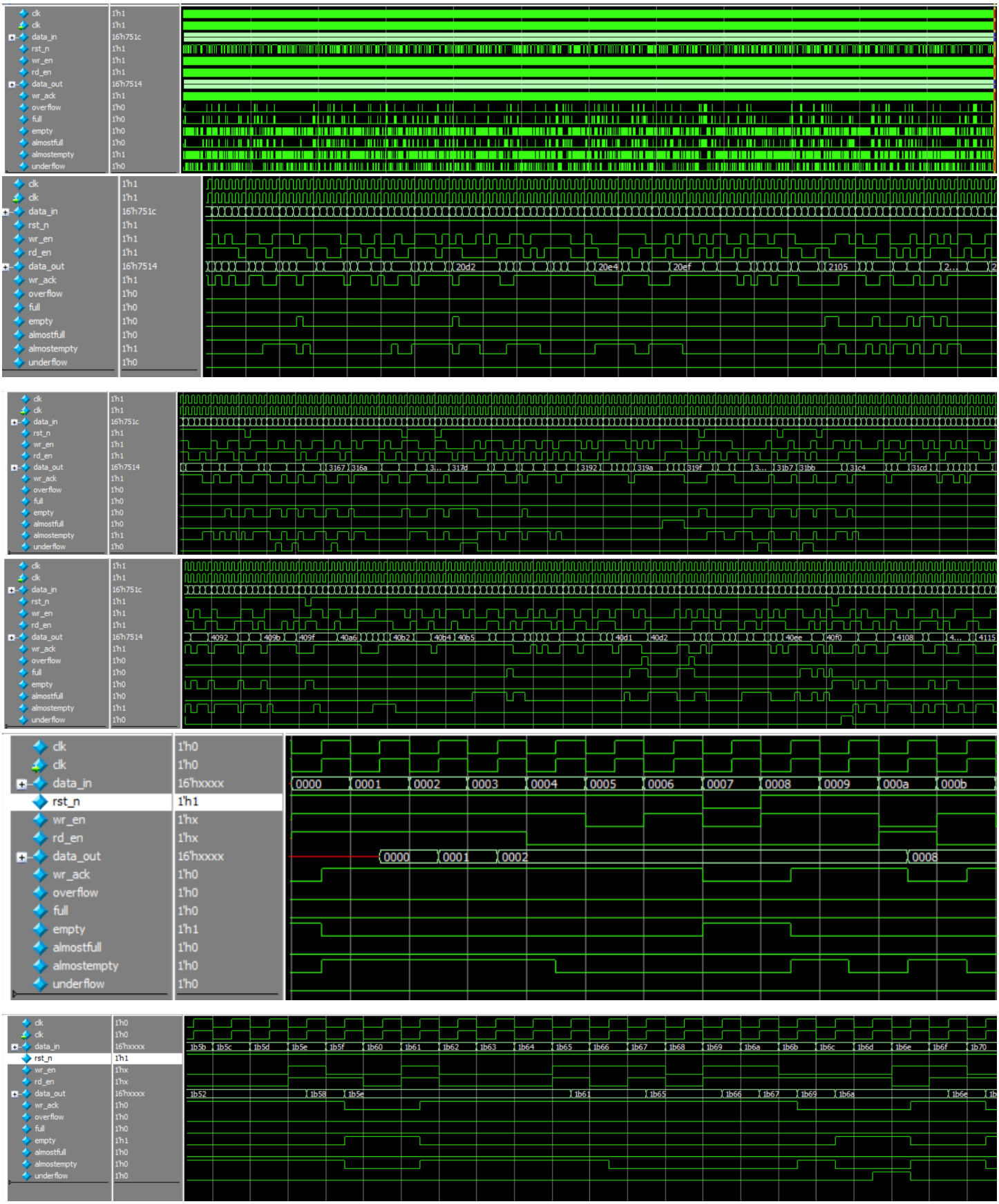
```
Toggle Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                         20        20         0   100.00%
```

```
Branch Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                        31        31         0   100.00%
```

8.5 Waveform







Go to my Github to show more : [repo]