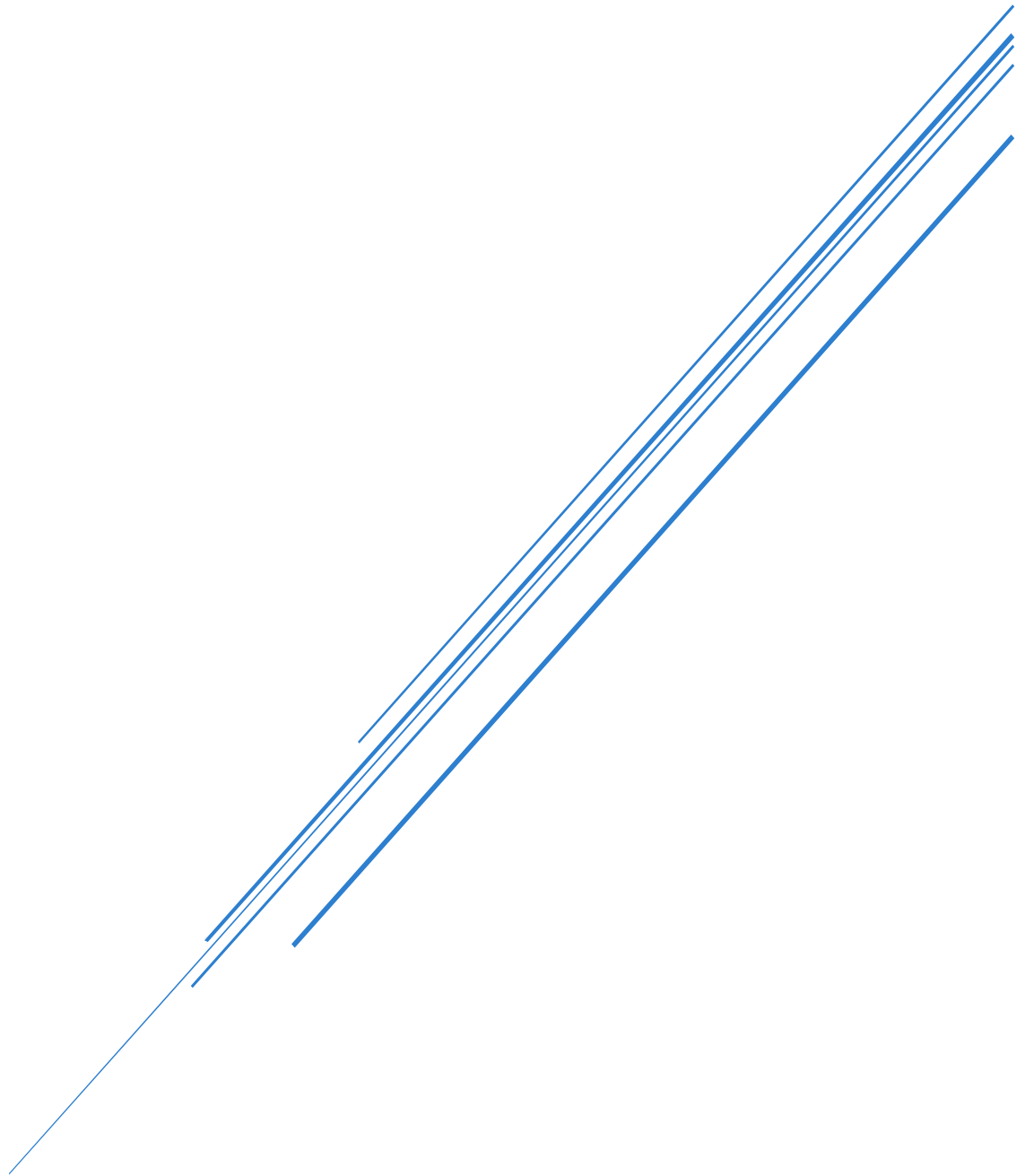


System Verilog project

FIFO

Digital Verification



Name: Nabil Ebrahim Abd_Elaty Abd_Elrazeq

TA: Mai Sherif

Table of contents

First Section: Code Implementation	Page 3
1. Interface Module (FIFO_IF.sv)	Page 3
2. Top Module (FIFO_TOP.sv)	Page 3
3. Shared Package (FIFO_SHARED.sv)	Page 4
4. Do File and Source Files	Page 4
5. Design Module (FIFO.sv)	Page 5
○ SystemVerilog Assertions (SVA)	Page 6
6. Transaction Package (FIFO_PKG_TRANS.sv)	Page 7
7. Coverage Package (FIFO_PKG_COV.sv)	Page 8
8. Scoreboard Package (FIFO_PKG_SCORE.sv)	Page 9
9. Monitor Module (FIFO_MONITOR.sv)	Page 10
10. Testbench Module (FIFO_TB.sv)	Page 11
<hr/>	
Second Section: Simulation Results	Page 12
1. Transcript Output	Page 12
2. Waveforms	Page 12
3. Assertions Coverage	Page 13
4. Functional Coverage	Page 13
5. Code Coverage	Page 14
○ Statement Coverage	Page 14
○ Branch Coverage	Page 15
○ Toggle Coverage	Page 15
<hr/>	
Third Section: Verification Plan	Page 16
1. Verification Plan Table	

First: Snippets from codes:

The Interface:

```
1 interface fifo_if (clk);
2
3 parameter FIFO_WIDTH = 16;
4 parameter FIFO_DEPTH = 8;
5 input bit clk;
6 logic [FIFO_WIDTH-1:0] data_in;
7 logic rst_n, wr_en, rd_en;
8
9 logic [FIFO_WIDTH-1:0] data_out;
10 logic wr_ack, overflow, underflow;
11 logic full, empty, almostfull, almostempty;
12
13 modport TEST (input clk, full, empty, almostfull, almostempty, wr_ack, overflow, underflow, data_out,
14               output data_in, wr_en, rd_en, rst_n);
15
16 modport DUT (input clk, data_in, wr_en, rd_en, rst_n, output full, empty, almostfull,
17              almostempty, wr_ack, overflow, underflow, data_out);
18
19 modport MONITOR (input clk, data_in, wr_en, rd_en, full, empty,
20                  rst_n, almostfull, almostempty, wr_ack, overflow, underflow, data_out);
21
22 endinterface
```

Top module:

```
1 module fifo_top;
2
3 bit clk;
4
5 initial begin
6     clk = 0;
7     forever #1 clk = ~clk;
8 end
9
10 fifo_if f_if (clk);
11 fifo_tb f_tb (f_if.TEST);
12 fifo f_dut(f_if.DUT);
13 fifo_monitor f_mon(f_if.MONITOR);
14
15 endmodule
```

Shared package:

```
1  package shared_pkg;
2
3      bit    test_finished = 0;
4      int    error_count = 0;
5      int    correct_count = 0;
6      event  sample_event;
7
8  endpackage
```

Do and source files:

```
1  vlib work
2
3  vlog -f src_files.list +cover +covercells +coveropt
4
5  vsim -voptargs=+acc work.fifo_top -cover
6
7  add wave *
8
9  coverage save fifo_top.ucdb -onexit
10
11 run -all
12
13 ## vcover report fifo_top.ucdb -details -annotate -all -output coverage_pro_rpt.txt
```

```
1  FIFO_IF.sv
2  FIFO_TOP.sv
3  FIFO_SHARED.sv
4  FIFO.sv
5  FIFO_PKG_TRANS.sv
6  FIFO_PKG_COV.sv
7  FIFO_PKG_SCORE.sv
8  FIFO_MONITOR.sv
9  FIFO_TB.sv
```

Design and assertions module:

```
1  module fifo (fifo_if.DUT f_if);
2
3      // -----
4      // Local parameters (taken from interface)
5      // -----
6      localparam FIFO_WIDTH    = f_if.FIFO_WIDTH;
7      localparam FIFO_DEPTH    = f_if.FIFO_DEPTH;
8      localparam max_fifo_addr = $clog2(FIFO_DEPTH);
9
10     // -----
11     // Internal registers
12     // -----
13     reg [FIFO_WIDTH-1:0] mem [FIFO_DEPTH-1:0];
14     reg [max_fifo_addr-1:0] wr_ptr, rd_ptr;
15     reg [max_fifo_addr:0] count;
16
17     // -----
18     // WRITE logic
19     // -----
20     always @(posedge f_if.clk or negedge f_if.rst_n) begin
21         if (!f_if.rst_n) begin
22             wr_ptr    <= 0;
23             // Bug detected: Reset signals FIFO_IF.overflow & FIFO_IF.wr_ack
24             f_if.wr_ack <= 0;
25             f_if.overflow <= 0;
26         end
27         else if (f_if.wr_en && !f_if.full) begin
28             mem[wr_ptr] <= f_if.data_in;
29             wr_ptr    <= wr_ptr + 1;
30             f_if.wr_ack <= 1;
31             f_if.overflow <= 0;
32         end
33         else begin
34             f_if.wr_ack <= 0;
35             if (f_if.wr_en && f_if.full)
36                 f_if.overflow <= 1; // reject write when full
37             else
38                 f_if.overflow <= 0;
39         end
40     end
41
42     // -----
43     // READ logic
44     // -----
45     always @(posedge f_if.clk or negedge f_if.rst_n) begin
46         if (!f_if.rst_n) begin
47             rd_ptr    <= 0;
48             // Bug detected: Reset signals FIFO_IF.underflow
49             f_if.data_out <= 0;
50             f_if.underflow <= 0;
51         end
52         else if (f_if.rd_en && !f_if.empty) begin
53             f_if.data_out <= mem[rd_ptr];
54             rd_ptr    <= rd_ptr + 1;
55             f_if.underflow <= 0;
56         end
57         else begin
58             if (f_if.rd_en && f_if.empty)
59                 f_if.underflow <= 1; // reject read when empty
60             else
61                 f_if.underflow <= 0;
62         end
63     end
64
65     // -----
66     // COUNT logic
67     // -----
68     always @(posedge f_if.clk or negedge f_if.rst_n) begin
69         if (!f_if.rst_n) begin
70             count <= 0;
71         end
72         else begin
73             case ({f_if.wr_en, f_if.rd_en})
74                 2'b10: if (!f_if.full) count <= count + 1; // write only
75                 2'b01: if (!f_if.empty) count <= count - 1; // read only
76                 2'b11: begin
77                     // simultaneous read and write
78                     // Bug detected: Unhandled case, If a read and write enables were high and the FIFO was FIFO_IF.empty, only writing will take place.
79                     // Bug detected: Unhandled cases, If a read and write enables were high and the FIFO was FIFO_IF.full, only reading will take place.
80                     if (f_if.empty) count <= count + 1; // write takes effect
81                     else if (f_if.full) count <= count - 1; // read takes effect
82                     // else: count unchanged (balanced read/write)
83                 end
84                 default: count <= count; // no operation
85             endcase
86         end
87     end
```

```

1 // -----
2 // Status signals
3 // -----
4 assign f_if.full      = (count == FIFO_DEPTH);
5 assign f_if.empty     = (count == 0);
6 assign f_if.almostfull = (count == FIFO_DEPTH-1); // Bug detected: f_if.FIFO_DEPTH-2 --> f_if.FIFO_DEPTH-1
7
8 assign f_if.almostempty = (count == 1);
9
10 always_comb begin
11     if(!f_if.rst_n) begin
12         reset_rd :assert final(rd_ptr==1'b0);
13         reset_wr :assert final(wr_ptr==1'b0);
14         reset_co :assert final(count==0);
15     end
16 end
17
18 property ack;
19     @(posedge f_if.clk) disable iff (!f_if.rst_n) (f_if.wr_en && !f_if.full) |> f_if.wr_ack ;
20 endproperty
21
22 property ovr;
23     @(posedge f_if.clk) disable iff (!f_if.rst_n) (f_if.wr_en && f_if.full) |> f_if.overflow;
24 endproperty
25
26 property udr;
27     @(posedge f_if.clk) disable iff (!f_if.rst_n) (f_if.rd_en && f_if.empty) |> f_if.underflow;
28 endproperty
29
30 property emp;
31     @(posedge f_if.clk) disable iff (!f_if.rst_n) count==0 |> f_if.empty;
32 endproperty
33
34 property ful;
35     @(posedge f_if.clk) disable iff (!f_if.rst_n) count==FIFO_DEPTH |> f_if.full;
36 endproperty
37
38 property Aful;
39     @(posedge f_if.clk) disable iff (!f_if.rst_n) (count==FIFO_DEPTH-1) |> f_if.almostfull;
40 endproperty
41
42 property Aemp;
43     @(posedge f_if.clk) disable iff (!f_if.rst_n) (count==1) |> f_if.almostempty;
44 endproperty
45
46 property wrap1;
47     @(posedge f_if.clk) disable iff (!f_if.rst_n) (f_if.wr_en && !f_if.full && wr_ptr == FIFO_DEPTH-1) |> (wr_ptr == 0);
48 endproperty
49
50 property wrap2;
51     @(posedge f_if.clk) disable iff (!f_if.rst_n) (f_if.rd_en && !f_if.empty && rd_ptr == FIFO_DEPTH-1) |> (rd_ptr == 0);
52 endproperty
53
54 property wrap3;
55     @(posedge f_if.clk) disable iff (!f_if.rst_n) (count<=FIFO_DEPTH && count>=0);
56 endproperty
57 property FIFO_internal_bounds;
58     @(posedge f_if.clk) disable iff (!f_if.rst_n) (wr_ptr < FIFO_DEPTH) && (rd_ptr < FIFO_DEPTH) && (count <= FIFO_DEPTH);
59 endproperty
60
61 assert property (ack);
62 assert property (ovr);
63 assert property (udr);
64 assert property (emp);
65 assert property (ful);
66 assert property (Aful);
67 assert property (Aemp);
68 assert property (wrap1);
69 assert property (wrap2);
70 assert property (wrap3);
71 assert property (FIFO_internal_bounds);
72 cover property (ack);
73 cover property (ovr);
74 cover property (udr);
75 cover property (emp);
76 cover property (ful);
77 cover property (Aful);
78 cover property (Aemp);
79 cover property (wrap1);
80 cover property (wrap2);
81 cover property (wrap3);
82 cover property (FIFO_internal_bounds);
83
84 endmodule

```

Transaction package:

```
1 package fifo_pkg_trans;
2 import shared_pkg::*;
3
4 class FIFO_transaction;
5     parameter FIFO_WIDTH = 16;
6     parameter FIFO_DEPTH = 8;
7     localparam max_fifo_addr = $clog2(FIFO_DEPTH);
8
9     rand logic [15:0] data_in;
10    rand logic wr_en;
11    rand logic rd_en;
12    rand logic rst_n;
13
14    logic wr_ack;
15    logic overflow;
16    logic underflow;
17    logic full;
18    logic empty;
19    logic almostfull;
20    logic almostempty;
21    logic [FIFO_WIDTH-1:0] data_out;
22
23    int RD_EN_ON_DIST;
24    int WR_EN_ON_DIST;
25    function new(int rd_on = 30, int wr_on = 70);
26        this.RD_EN_ON_DIST = rd_on;
27        this.WR_EN_ON_DIST = wr_on;
28    endfunction
29
30    constraint rst_less_often {
31        rst_n dist {1 :/ 95, 0 :/ 5};
32    }
33
34    constraint wr_dist {
35        wr_en dist {1 :/ WR_EN_ON_DIST, 0 :/ (100-WR_EN_ON_DIST)};
36    }
37    constraint rd_dist {
38        rd_en dist {1 :/ RD_EN_ON_DIST, 0 :/ (100-RD_EN_ON_DIST)};
39    }
40
41 endclass
42
43 endpackage
```

Coverage package:

```
1 package fifo_pkg_COV;
2 import fifo_pkg_trans::*;
3
4 class FIFO_coverage;
5     FIFO_transaction F_cvg_txn;
6
7     covergroup cg;
8
9         cp_wr_en : coverpoint F_cvg_txn.wr_en;
10        cp_rd_en : coverpoint F_cvg_txn.rd_en;
11
12        cp_wr_ack: coverpoint F_cvg_txn.wr_ack;
13        cp_overflow: coverpoint F_cvg_txn.overflow;
14        cp_underflow: coverpoint F_cvg_txn.underflow;
15        cp_full: coverpoint F_cvg_txn.full;
16        cp_empty: coverpoint F_cvg_txn.empty;
17        cp_almostfull: coverpoint F_cvg_txn.almostfull;
18        cp_almostempty: coverpoint F_cvg_txn.almostempty;
19
20        cross_wre_rd_wrack: cross cp_wr_en, cp_rd_en, cp_wr_ack {
21            // wr_ack can only be 1 when wr_en is 1
22            illegal_bins wr_ack_without_wr_en = binsof(cp_wr_en) intersect {0} &&
23                                                binsof(cp_wr_ack) intersect {1};
24        }
25
26        cross_wre_rd_oflow: cross cp_wr_en, cp_rd_en, cp_overflow {
27            // overflow can only be 1 when wr_en is 1
28            illegal_bins overflow_without_wr_en = binsof(cp_wr_en) intersect {0} &&
29                                                binsof(cp_overflow) intersect {1};
30        }
31
32        cross_wre_rd_uflow: cross cp_wr_en, cp_rd_en, cp_underflow {
33            // underflow can only be 1 when rd_en is 1
34            illegal_bins underflow_without_rd_en = binsof(cp_rd_en) intersect {0} &&
35                                                binsof(cp_underflow) intersect {1};
36        }
37
38        cross_wre_rd_full: cross cp_wr_en, cp_rd_en, cp_full {
39            illegal_bins full_with_only_read = binsof(cp_rd_en) intersect {1} &&
40                                                binsof(cp_full) intersect {1};
41        }
42
43        cross_wre_rd_empty: cross cp_wr_en, cp_rd_en, cp_empty;
44
45        cross_wre_rd_afull: cross cp_wr_en, cp_rd_en, cp_almostfull;
46
47        cross_wre_rd_aempty: cross cp_wr_en, cp_rd_en, cp_almostempty;
48
49    endgroup : cg
50
51
52    function new();
53        cg = new();
54    endfunction
55
56    function void sample_data(FIFO_transaction F_txn);
57        this.F_cvg_txn = F_txn;
58
59        cg.sample();
60    endfunction
61
62 endclass
63
64
65 endpackage
```


Scoreboard package:

```
1 package fifo_pkg_score;
2 import fifo_pkg_trans::*;
3 import shared_pkg::*;
4
5 FIFO_transaction FIFO_transaction_object = new();
6
7 class FIFO_scoreboard;
8
9     bit [FIFO_transaction_object.FIFO_WIDTH-1:0] fifo_q[$];
10
11     logic [FIFO_transaction_object.FIFO_WIDTH-1:0] data_out_ref;
12     logic full_ref, empty_ref, almostfull_ref, almostempty_ref;
13     logic wr_ack_ref, overflow_ref, underflow_ref;
14     int count_ref;
15
16 function void reference_model(FIFO_transaction tr);
17
18     if (!tr.rst_n) begin
19         fifo_q.delete();
20         data_out_ref = '0;
21         full_ref = 0;
22         empty_ref = 1;
23         almostfull_ref = 0;
24         almostempty_ref = 0;
25         count_ref = 0;
26         return;
27     end else begin
28         if (tr.wr_en && (!full_ref)) begin
29             fifo_q.push_back(tr.data_in);
30         end
31         if (tr.rd_en && (!empty_ref)) begin
32             data_out_ref = fifo_q.pop_front();
33         end
34
35         count_ref = fifo_q.size();
36         full_ref = (count_ref == tr.FIFO_DEPTH);
37         empty_ref = (count_ref == 0);
38     end
39 endfunction
40
41
42 function void check_data(input FIFO_transaction ob1);
43     reference_model(ob1);
44
45     if ((ob1.data_out != data_out_ref)&&(ob1.wr_en)&&(ob1.rst_n)) begin
46         $display("Error!!, At time %t, data_out %d doesn't equal data_out_ref %d !!", $time, ob1.data_out, data_out_ref);
47         error_count++;
48     end
49     else begin
50         $display("Success, At time %t, data_out= %d equals data_out_ref= %d", $time, ob1.data_out, data_out_ref);
51         correct_count++;
52     end
53 endfunction
54
55 function new();
56     data_out_ref = '0;
57     full_ref = 0;
58     empty_ref = 1;
59     almostfull_ref = 0;
60     almostempty_ref = 0;
61     count_ref = 0;
62 endfunction
63
64 endclass
65 endpackage
```

Monitor module:

```
1  import fifo_pkg_trans::*;
2  import fifo_pkg_COV::*;
3  import fifo_pkg_score::*;
4  import shared_pkg::*;
5
6  module fifo_monitor(fifo_if.MONITOR f_if);
7
8      FIFO_transaction mon_txn;
9      FIFO_coverage  mon_cvg;
10     FIFO_scoreboard mon_sb;
11
12     initial begin
13
14         mon_txn = new();
15         mon_cvg = new();
16         mon_sb  = new();
17
18         forever begin
19             @(sample_event);
20             mon_txn.data_in  = f_if.data_in;
21             mon_txn.wr_en    = f_if.wr_en;
22             mon_txn.rd_en    = f_if.rd_en;
23             mon_txn.rst_n    = f_if.rst_n;
24
25             @(negedge f_if.clk);
26
27             mon_txn.data_out = f_if.data_out;
28             mon_txn.wr_ack   = f_if.wr_ack;
29             mon_txn.overflow = f_if.overflow;
30             mon_txn.underflow = f_if.underflow;
31             mon_txn.full     = f_if.full;
32             mon_txn.empty    = f_if.empty;
33             mon_txn.almostfull = f_if.almostfull;
34             mon_txn.almostempty = f_if.almostempty;
35
36
37             fork
38                 begin
39                     mon_cvg.sample_data(mon_txn);
40                 end
41                 begin
42                     mon_sb.check_data(mon_txn);
43                 end
44             join
45
46
47             if (test_finished) begin
48                 $display("[%0t] MONITOR: Test finished. correct=
49 errors=$finish;
50 ", $time$ correct_count, error_count);
51             end
52         end
53     end
54 endmodule
```

Testbench module:

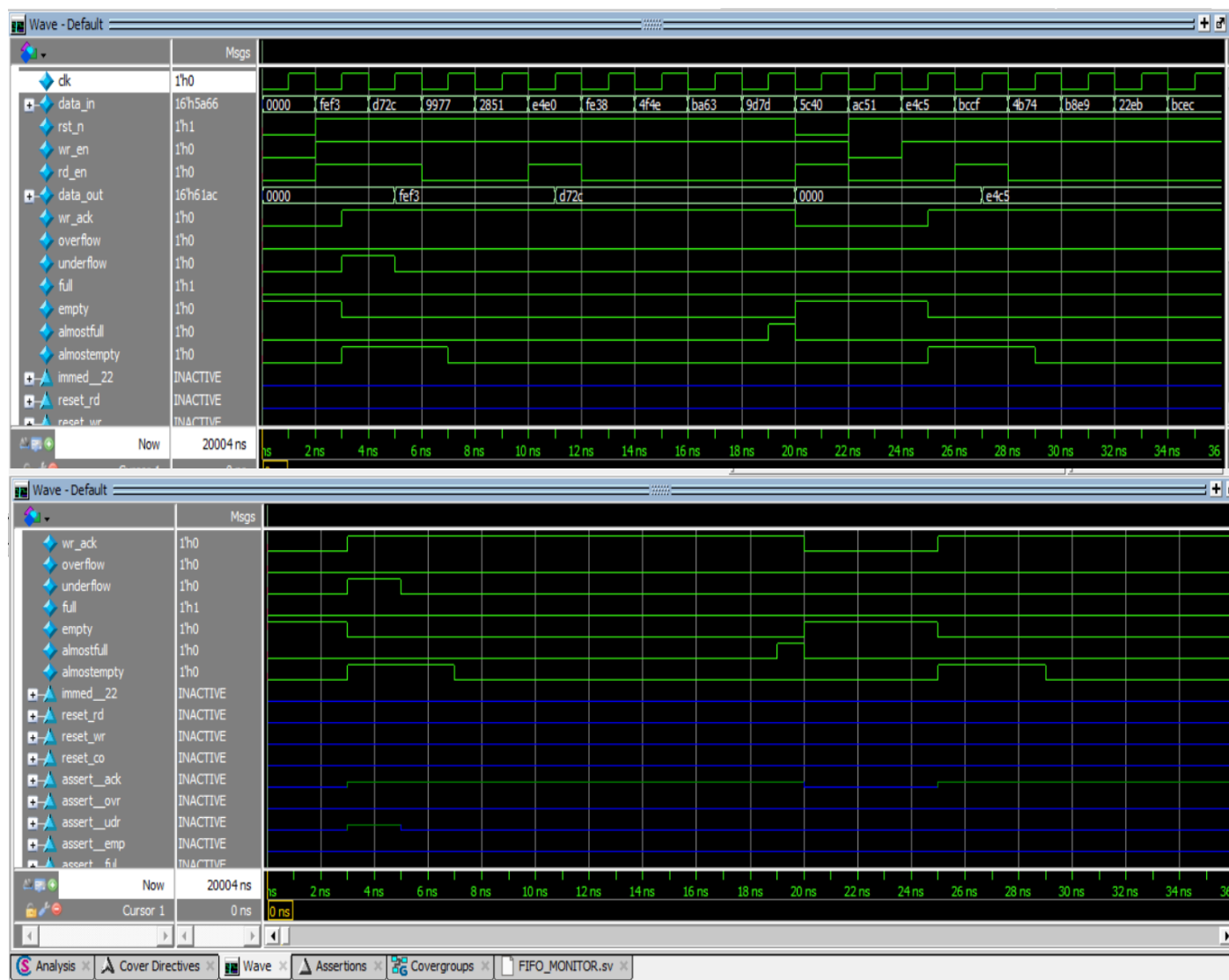
```
1  module fifo_tb(fifo_if.TEST f_if);
2
3      import fifo_pkg_trans::*;
4      import shared_pkg::*;
5      import fifo_pkg_COV::*;
6      import fifo_pkg_score::*;
7
8      FIFO_transaction drv_txn;
9
10     int unsigned NUM_CYCLES = 10000;
11
12     initial begin
13
14         drv_txn = new();
15
16         f_if.rst_n=0; f_if.rd_en=0; f_if.wr_en=0; f_if.data_in=0;
17         @(negedge f_if.clk);
18         -> sample_event;
19         f_if.rst_n=1;
20
21         repeat(NUM_CYCLES) begin
22             assert(drv_txn.randomize());
23             f_if.rst_n=drv_txn.rst_n;
24             f_if.rd_en=drv_txn.rd_en;
25             f_if.wr_en=drv_txn.wr_en;
26             f_if.data_in=drv_txn.data_in;
27             @(negedge f_if.clk);
28             -> sample_event;
29         end
30
31         test_finished =1;
32
33     end
34
35 endmodule
```

Second: Snippets from Simulation results:

Transcript:

```
Transcript
# Success, At time 19980, data_out= 22658 equals data_out_ref= 22658
# Success, At time 19982, data_out= 52733 equals data_out_ref= 52733
# Success, At time 19984, data_out= 52733 equals data_out_ref= 52733
# Success, At time 19986, data_out= 15563 equals data_out_ref= 15563
# Success, At time 19988, data_out= 15563 equals data_out_ref= 15563
# Success, At time 19990, data_out= 15563 equals data_out_ref= 15563
# Success, At time 19992, data_out= 15563 equals data_out_ref= 15563
# Success, At time 19994, data_out= 15563 equals data_out_ref= 15563
# Success, At time 19996, data_out= 25004 equals data_out_ref= 25004
# Success, At time 19998, data_out= 25004 equals data_out_ref= 25004
# Success, At time 20000, data_out= 25004 equals data_out_ref= 25004
# Success, At time 20002, data_out= 25004 equals data_out_ref= 25004
# Success, At time 20004, data_out= 25004 equals data_out_ref= 25004
# [20004] MONITOR: Test finished. correct=10001 errors=0
# ** Note: $finish : FIFO_MONITOR.sv(49)
# Time: 20004 ns Iteration: 1 Instance: /fifo_top/f_mon
# 1
```

Waveforms:



Assertions coverage:

```
====
=== Instance: /fifo_top/f_dut
=== Design Unit: work.fifo
=====
```

Assertion Coverage:

Assertions	14	14	0	100.00%
------------	----	----	---	---------

Name	Assertion Type	Language	Enable	Failure Count	Pass Count	Active Count	Memory	Peak Memory	Peak Memory Time	Cumulative Threads	ATV	Assertion Expression	Included
/fifo_top/f_tb/#Jbkl#217929410#21/Immed_22	Immediate	SVA	on	0	1	-	-	-	-	-	off	assert (randomize(...))	✓
/fifo_top/f_dut/reset_rd	Immediate	SVA	on	0	1	-	-	-	-	-	off	assert (rd_ptr==0)	✓
/fifo_top/f_dut/reset_wr	Immediate	SVA	on	0	1	-	-	-	-	-	off	assert (wr_ptr==0)	✓
/fifo_top/f_dut/reset_co	Immediate	SVA	on	0	1	-	-	-	-	-	off	assert (count==0)	✓
/fifo_top/f_dut/assert_ack	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓
/fifo_top/f_dut/assert_ovr	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓
/fifo_top/f_dut/assert_udr	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓
/fifo_top/f_dut/assert_emp	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓
/fifo_top/f_dut/assert_ful	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓
/fifo_top/f_dut/assert_Aful	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓
/fifo_top/f_dut/assert_Aemp	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓
/fifo_top/f_dut/assert_wrap1	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓
/fifo_top/f_dut/assert_wrap2	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓
/fifo_top/f_dut/assert_wrap3	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓
/fifo_top/f_dut/assert_FIFO_internal_bounds	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(posedge f_jf.clk) disable...	✓

Function coverage:

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_i
/fifo_pkg_COV/FIFO_coverage		100.00%					
TYPE cg		100.00%	100	100.00...	✓	✓	
CVP cg::cp_wr_en		100.00%	100	100.00...	✓	✓	
CVP cg::cp_rd_en		100.00%	100	100.00...	✓	✓	
CVP cg::cp_wr_ack		100.00%	100	100.00...	✓	✓	
CVP cg::cp_overflow		100.00%	100	100.00...	✓	✓	
CVP cg::cp_underflow		100.00%	100	100.00...	✓	✓	
CVP cg::cp_full		100.00%	100	100.00...	✓	✓	
CVP cg::cp_empty		100.00%	100	100.00...	✓	✓	
CVP cg::cp_almostfull		100.00%	100	100.00...	✓	✓	
CVP cg::cp_almostempty		100.00%	100	100.00...	✓	✓	
CROSS cg::cross_wre_rd_wrack		100.00%	100	100.00...	✓	✓	
CROSS cg::cross_wre_rd_oflow		100.00%	100	100.00...	✓	✓	
CROSS cg::cross_wre_rd_uflow		100.00%	100	100.00...	✓	✓	
CROSS cg::cross_wre_rd_full		100.00%	100	100.00...	✓	✓	
CROSS cg::cross_wre_rd_empty		100.00%	100	100.00...	✓	✓	
CROSS cg::cross_wre_rd_aful		100.00%	100	100.00...	✓	✓	
CROSS cg::cross_wre_rd_aempty		100.00%	100	100.00...	✓	✓	

```

=====
=== Instance: /fifo_pkg_COV
=== Design Unit: work.fifo_pkg_COV
=====

Covergroup Coverage:
  Covergroups          1      na      na    100.00%
  Coverpoints/Crosses  16      na      na      na
  Covergroup Bins      66      66      0    100.00%
=====

```

Code coverage:

-Statements:

Code Coverage Analysis - Statements - by instance (/fifo_top/f_dut)

Statement: [✓] [✗] [E]

FIFO.sv

```

20 always @(posedge f_if.clk or negedge f_if.rst_n) begin
22   wr_ptr      <= 0;
24   f_if.wr_ack <= 0;
25   f_if.overflow <= 0;
28   mem[wr_ptr] <= f_if.data_in;
29   wr_ptr      <= wr_ptr + 1;
30   f_if.wr_ack <= 1;
31   f_if.overflow <= 0;
34   f_if.wr_ack <= 0;
36   f_if.overflow <= 1; // reject write when full
38   f_if.overflow <= 0;
45 always @(posedge f_if.clk or negedge f_if.rst_n) begin
47   rd_ptr      <= 0;
49   f_if.data_out <= 0;
50   f_if.underflow <= 0;
53   f_if.data_out <= mem[rd_ptr];
54   rd_ptr      <= rd_ptr + 1;
55   f_if.underflow <= 0;
59   f_if.underflow <= 1; // reject read when empty
61   f_if.underflow <= 0;
68 always @(posedge f_if.clk or negedge f_if.rst_n) begin
70   count <= 0;
74   2'b10: if (!f_if.full) count <= count + 1; // write only
75   2'b01: if (!f_if.empty) count <= count - 1; // read only
80   if (f_if.empty) count <= count + 1; // write takes effect
81   else if (f_if.full) count <= count - 1; // read takes effect
84 default: count <= count; // no operation
92 assign f_if.full = (count == FIFO_DEPTH);
93 assign f_if.empty = (count == 0);
94 assign f_if.almostfull = (count == FIFO_DEPTH-1); // Bug detected: f_if.FIFO_DEPTH-2 --> f_if.FIFO_DEPTH-1
96 assign f_if.almostempty = (count == 1);
98 always_comb begin

```

Analysis | Cover Directives | Wave | Assertions | Covergroups | FIFO_MONITOR.sv

Statement Coverage:

Enabled Coverage	Bins	Hits	Misses	Coverage
Statements	32	32	0	100.00%

=====Statement Details=====

-Branches:

Code Coverage Analysis

Branches - by instance (/fifo_top/f_dut)

FIFO.sv

- 21 if (!f_if.rst_n) begin
- 27 else if (f_if.wr_en && !f_if.full) begin
- 33 else begin
- 35 if (f_if.wr_en && f_if.full)
- 37 else
- 46 if (!f_if.rst_n) begin
- 52 else if (f_if.rd_en && !f_if.empty) begin
- 57 else begin
- 58 if (f_if.rd_en && f_if.empty)
- 60 else
- 69 if (!f_if.rst_n) begin
- 72 else begin
- 74 2'b10: if (!f_if.full) count <= count + 1; // write only
- 75 2'b01: if (!f_if.empty) count <= count - 1; // read only
- 76 2'b11: begin
- 80 if (f_if.empty) count <= count + 1; // write takes effect
- 81 else if (f_if.full) count <= count - 1; // read takes effect
- 84 default: count <= count; // no operation
- 99 if (!f_if.rst_n) begin

FIFO.SV(102)		0		1
Branch Coverage:				
Enabled Coverage	Bins	Hits	Misses	Coverage
-----	----	----	-----	-----
Branches	25	25	0	100.00%
=====Branch Details=====				

-Toggles:

Code Coverage Analysis

Toggles - by instance (/fifo_top/f_dut)

sim:/fifo_top/f_dut

- count
- rd_ptr
- wr_ptr

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

Third: Snippet from The Verification plan:

Label	Design Requirement Description	Stimulus Generation	Functional Coverage	Functionality Check
RESET_1	When reset is asserted (rst_n=0), all internal registers (wr_ptr, rd_ptr, count) should be cleared to 0	Apply rst_n=0 at start of simulation, then randomize with 5% probability during test using constraint rst_less_often	Cover reset assertion with all possible FIFO states (empty, full, partial)	Assert that wr_ptr=0, rd_ptr=0, count=0, empty=1, full=0 immediately after reset
RESET_2	When reset is asserted, all output signals (wr_ack, overflow, underflow, data_out) should be cleared	Apply rst_n=0 with various combinations of wr_en and rd_en active	Cover reset with active write/read attempts	Assert that wr_ack=0, overflow=0, underflow=0, data_out=0 during reset
WRITE_1	When wr_en=1 and FIFO is not full, data should be written to memory at wr_ptr location	Randomize data_in with wr_en=1 using 70% distribution, ensure FIFO not full	Cover write operations at different fill levels: empty, partial, almost full	Check that wr_ack=1 on next cycle and data is stored correctly in reference model
WRITE_2	When wr_en=1 and FIFO is not full, wr_ptr should increment (with wrap-around at FIFO_DEPTH-1)	Generate consecutive writes until FIFO is full	Cover wr_ptr increment from 0 to FIFO_DEPTH-1, cover wrap-around case	Assert property wrap1: wr_ptr wraps to 0 after reaching FIFO_DEPTH-1
WRITE_3	When wr_en=1 and FIFO is not full, count should increment by 1	Generate single writes with no reads	Cover count transitions from 0 to FIFO_DEPTH	Verify count increments correctly in scoreboard reference model
WRITE_4	When wr_en=1 and FIFO is full, overflow flag should be set and write rejected	Generate write attempts when count=FIFO_DEPTH	Cover overflow condition: cross cp_wr_en, cp_full, cp_overflow	Assert property ovr: overflow=1 when wr_en=1 and full=1
READ_1	When rd_en=1 and FIFO is not empty, data should be read from memory at rd_ptr location	Randomize rd_en=1 using 30% distribution, ensure FIFO not empty	Cover read operations at different fill levels: full, partial, almost empty	Check that data_out matches expected value from reference queue
READ_2	When rd_en=1 and FIFO is not empty, rd_ptr should increment (with wrap-around at FIFO_DEPTH-1)	Generate consecutive reads until FIFO is empty	Cover rd_ptr increment from 0 to FIFO_DEPTH-1, cover wrap-around case	Assert property wrap2: rd_ptr wraps to 0 after reaching FIFO_DEPTH-1
READ_3	When rd_en=1 and FIFO is not empty, count should decrement by 1	Generate single reads with no writes	Cover count transitions from FIFO_DEPTH down to 0	Verify count decrements correctly in scoreboard reference model
READ_4	When rd_en=1 and FIFO is empty, underflow flag should be set and read rejected	Generate read attempts when count=0	Cover underflow condition: cross cp_rd_en, cp_empty, cp_underflow	Assert property udr: underflow=1 when rd_en=1 and empty=1
SIMUL_W R_RD_1	When wr_en=1 and rd_en=1 simultaneously and FIFO is neither empty nor full, count should remain unchanged	Generate simultaneous wr_en=1 and rd_en=1 when 0<count<FIFO_DEPTH	Cross coverage: cp_wr_en=1, cp_rd_en=1 with various count values	Verify count remains stable and both operations succeed
SIMUL_W R_RD_2	When wr_en=1 and rd_en=1 simultaneously and FIFO is empty, only write should occur and count should increment	Generate simultaneous wr_en=1 and rd_en=1 when count=0	Cross coverage: cp_wr_en=1, cp_rd_en=1, cp_empty=1	Verify count increments by 1, underflow=0, write succeeds
SIMUL_W R_RD_3	When wr_en=1 and rd_en=1 simultaneously and FIFO is full, only read should occur and count should decrement	Generate simultaneous wr_en=1 and rd_en=1 when count=FIFO_DEPTH	Cross coverage: cp_wr_en=1, cp_rd_en=1, cp_full=1	Verify count decrements by 1, overflow=0, read succeeds
STATUS_F ULL	Full flag should be asserted when count equals FIFO_DEPTH	Fill FIFO completely by writing FIFO_DEPTH times without reads	Cover full flag assertion: cp_full=1	Assert property ful: full=1 when count=FIFO_DEPTH
STATUS_E MPY	Empty flag should be asserted when count equals 0	Empty FIFO completely by reading all entries	Cover empty flag assertion: cp_empty=1	Assert property emp: empty=1 when count=0

STATUS_A FULL	Almost full flag should be asserted when count equals FIFO_DEPTH-1	Write FIFO_DEPTH-1 entries	Cover almost full: cross_wre_rd_afull with count=FIFO_DEPTH-1	Assert property Aful: almostfull=1 when count=FIFO_DEPTH-1
STATUS_A EMPTY	Almost empty flag should be asserted when count equals 1	Read until only 1 entry remains	Cover almost empty: cross_wre_rd_aempty with count=1	Assert property Aemp: almostempty=1 when count=1
WR_ACK	Write acknowledge should be asserted only when wr_en=1 and FIFO is not full	Generate various wr_en patterns with different FIFO states	Cover wr_ack with illegal bins: wr_ack=1 when wr_en=0	Assert property ack: wr_ack=1 follows successful write
DATA_INT TEGRITY	Data written to FIFO should match data read out in FIFO order	Write sequence of unique random data, then read back	Cover full range of data_in values (16-bit)	Scoreboard compares data_out with expected values from reference queue
PTR_BOU NDS	Write and read pointers should always remain within valid range [0, FIFO_DEPTH-1]	Generate extensive random sequences over 10000 cycles	Cover boundary cases: wr_ptr and rd_ptr at min/max values	Assert property FIFO_internal_bounds: wr_ptr<FIFO_DEPTH and rd_ptr<FIFO_DEPTH
COUNT_B OUNDS	Count should always remain within valid range [0, FIFO_DEPTH]	Generate random write/read patterns	Cover all count values from 0 to FIFO_DEPTH	Assert property wrap3: 0<=count<=FIFO_DEPTH at all times
STRESS_T EST	FIFO should handle continuous random operations without errors	Run 10000 random cycles with 70% write / 30% read distribution	Achieve 100% coverage on all coverpoints and crosses	Monitor reports correct_count vs error_count, expect 0 errors

الحمد لله