

REPORT

Project (1)

FIFO

TEAM #21

PREPARED BY
Nouran Hamdy

SUBMITTED TO
Eng Kareem Waseem

TABLE OF CONTENTS

01	Code Snippets	1
02	Bug Report	13
03	Waveform Snippets	14
04	Coverage Report	18

CODE SNIPPETS

TOP

```

1  module FIFO_top;
2      bit clk;
3
4      //clock generation
5      initial begin
6          forever #1 clk = ~clk;
7      end
8
9      FIFO_if F_if (clk);
10
11     FIFO FIFO_DUT (F_if);
12     FIFO_tb FIFO_TEST (F_if);
13     FIFO_monitor FIFO_MON (F_if);
14
15 endmodule

```

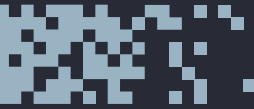
INTERFACE

```

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

```

CODE SNIPPETS



DESIGN

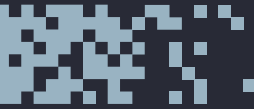
Before (Original)

```

24
25 always @(posedge clk or negedge rst_n) begin
26     if (!rst_n) begin
27         wr_ptr <= 0;
28     end
29     else if (wr_en && count < FIFO_DEPTH) begin
30         mem[wr_ptr] <= data_in;
31         wr_ack <= 1;
32         wr_ptr <= wr_ptr + 1;
33     end
34     else begin
35         wr_ack <= 0;
36         if (full & wr_en)
37             overflow <= 1;
38         else
39             overflow <= 0;
40     end
41 end
42
43 always @(posedge clk or negedge rst_n) begin
44     if (!rst_n) begin
45         rd_ptr <= 0;
46     end
47     else if (rd_en && count != 0) begin
48         data_out <= mem[rd_ptr];
49         rd_ptr <= rd_ptr + 1;
50     end
51 end
52
53 always @(posedge clk or negedge rst_n) begin
54     if (!rst_n) begin
55         count <= 0;
56     end
57     else begin
58         if ( ({wr_en, rd_en} == 2'b10) && !full)
59             count <= count + 1;
60         else if ( ({wr_en, rd_en} == 2'b01) && !empty)
61             count <= count - 1;
62     end
63 end
64
65 assign full = (count == FIFO_DEPTH)? 1 : 0;
66 assign empty = (count == 0)? 1 : 0;
67 assign underflow = (empty && rd_en)? 1 : 0;
68 assign almostfull = (count == FIFO_DEPTH-2)? 1 : 0;
69 assign almostempty = (count == 1)? 1 : 0;
70
71 endmodule

```

CODE SNIPPETS



DESIGN

After (Edited)

```

81 module FIFO(FIFO_if.DUT fd);
82
83 localparam max_fifo_addr = $clog2(fd.FIFO_DEPTH);
84
85 logic [fd.FIFO_WIDTH-1:0] mem [fd.FIFO_DEPTH-1:0];
86
87 bit [max_fifo_addr-1:0] wr_ptr, rd_ptr;
88 bit [max_fifo_addr:0] count;
89
90 always @(posedge fd.clk or negedge fd.rst_n) begin
91     if (!fd.rst_n) begin
92         wr_ptr <= 0;
93         fd.overflow <= 0;
94         fd.wr_ack <= 0;
95     end
96     else begin
97         //wr_en high and FIFO not full - or - wr_en and rd_en are high (read and write in parallel)
98         if ( (fd.wr_en && (count < (fd.FIFO_DEPTH))) || (fd.wr_en && fd.rd_en /*&& (count == (fd.FIFO_DEPTH))*/ ) ) begin
99             mem[wr_ptr] <= fd.data_in;
100             fd.wr_ack <= 1;
101             wr_ptr <= wr_ptr + 1;
102             fd.overflow <= 0;
103         end
104         else begin
105             fd.wr_ack <= 0;
106             //else branch: FIFO is full or wr_en inactive
107             if (/*fd.full & */fd.wr_en)
108                 fd.overflow <= 1;
109             else
110                 fd.overflow <= 0;
111         end
112
113         if (!fd.rd_en) fd.data_out <= 0;
114     end
115 end
116
117 always @(posedge fd.clk or negedge fd.rst_n) begin
118     if (!fd.rst_n) begin
119         rd_ptr <= 0;
120         fd.underflow <= 0;
121         fd.data_out <= 0;
122     end
123     //rd_en high and FIFO not empty - or - rd_en and wr_en are high (read and write in parallel)
124     else if (fd.rd_en && count != 0 || (fd.wr_en && fd.rd_en/* && (count == 0)*/) ) begin
125         //if FIFO is empty: wr_en and rd_en are active (read and write in parallel)
126         if((fd.wr_en /*&& fd.rd_en*/ && (count == 0))) fd.data_out <= fd.data_in;
127         else fd.data_out <= mem[rd_ptr];
128         rd_ptr <= rd_ptr + 1;
129         fd.underflow <= 0;
130     end
131     else begin
132         if(fd.empty && fd.rd_en)
133             fd.underflow <= 1;
134         else
135             fd.underflow <= 0;
136
137         fd.data_out <= 0;
138     end
139 end
140
141 always @(posedge fd.clk or negedge fd.rst_n) begin
142     if (!fd.rst_n) begin
143         count <= 0;
144     end
145     else begin
146         if ( ({fd.wr_en, fd.rd_en} == 2'b10) && !fd.full)
147             count <= count + 1;
148         if ( ({fd.wr_en, fd.rd_en} == 2'b01) && !fd.empty)
149             count <= count - 1;
150     end
151 end
152
153 assign fd.full = (count == fd.FIFO_DEPTH)? 1 : 0;
154 assign fd.empty = (count == 0)? 1 : 0;
155 assign fd.almostfull = (count == fd.FIFO_DEPTH-1)? 1 : 0;
156 assign fd.almostempty = (count == 1)? 1 : 0;
157

```

CODE SNIPPETS

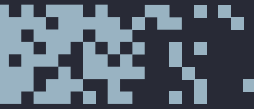
ASSERTIONS

```

160
161 `ifdef SIM
162
163 /*Assertions Plan:
164 Check position and pointers in all cases (to ensure a correct state of fifo)
165 - Reset Signal (relevant Outputs (~full, empty))
166
167 - Write Enabled (check if data is written in FIFO, wr_ack)
168 - Write Enabled when FIFO is full (overflow, ~wr_ack, non-destructive to FIFO)
169 - Write Disabled (~wr_ack, no write operation is done in FIFO)
170
171 - Read Enabled (check output data if it's the one written in FIFO)
172 - Read Enabled when FIFO is empty (underflow, no output data, non-destructive to FIFO)
173 - Read Disabled (no output data)
174
175 - Check almostfull then full
176 - Check almostempty then empty
177 */
178
179 //Checks rst_n Functionality
180 property reset_asserted;
181 @(posedge fd.clk)
182
183 !fd.rst_n |> ~(fd.data_out) && ~(wr_ptr || rd_ptr || count) && fd.empty && ~fd.full && ~(fd.overflow || fd.underflow);
184 endproperty
185
186 //Checks wr_en Functionality
187 property wr_enabled;
188 bit [max_fifo_addr-1:0] wr_ptr_exp = 0;
189 @(posedge fd.clk) disable iff(!fd.rst_n || (count == 8) || fd.rd_en)
190
191 ((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));
192 endproperty
193
194 property wr_enabled_full;
195 @(posedge fd.clk) disable iff(!fd.rst_n || fd.rd_en)
196
197 (fd.wr_en) && (fd.full) |> (fd.overflow) && ~(fd.wr_ack) && $stable(mem) && $stable(wr_ptr) && $stable(count);
198 endproperty
199
200 property wr_disabled;
201 @(posedge fd.clk) disable iff(!fd.rst_n)
202
203 !(fd.wr_en) |> ~(fd.wr_ack) && $stable(mem);
204 endproperty
205
206 //Checks rd_en Functionality
207 property rd_enabled;
208 bit [max_fifo_addr-1:0] rd_ptr_exp = 0;
209 @(posedge fd.clk) disable iff(!fd.rst_n || (count == 0) || fd.wr_en)
210
211 ((fd.rd_en), rd_ptr_exp = rd_ptr + 1) |> (fd.data_out == mem[$past(rd_ptr)]) && (rd_ptr == rd_ptr_exp);
212 endproperty
213
214 property rd_enabled_empty;
215 @(posedge fd.clk) disable iff(!fd.rst_n || fd.wr_en)
216
217 (fd.rd_en) && (fd.empty) |> (fd.underflow) && ~(fd.data_out) && $stable(mem) && $stable(rd_ptr);
218 endproperty
219
220 property rd_disabled;
221 @(posedge fd.clk) disable iff(!fd.rst_n)
222
223 ~(fd.rd_en) |> ~(fd.data_out) && $stable(rd_ptr);
224 endproperty
225
226 //Checking Outputs Values
227 property FIFO_almostfull;
228 @(posedge fd.clk) disable iff(!fd.rst_n)
229
230 (count == (fd.FIFO_DEPTH - 1)) |> (fd.almostfull);
231 endproperty
232
233 property FIFO_full;
234 @(posedge fd.clk) disable iff(!fd.rst_n)
235
236 (count >= fd.FIFO_DEPTH) |> (fd.full);
237 endproperty
238
239 property FIFO_almostempty;
240 @(posedge fd.clk) disable iff(!fd.rst_n)
241
242 (count == 1) |> (fd.almostempty);
243 endproperty
244
245 property FIFO_empty;
246 @(posedge fd.clk) disable iff(!fd.rst_n)
247
248 (count == 0) |> (fd.empty);
249 endproperty
250
251
252
253
254

```

CODE SNIPPETS



TESTBENCH

```

1  module FIFO_tb(FIFO_if.TEST ft);
2
3  import transaction_pkg::*;
4  import shared_pkg::*;
5
6  parameter TESTS = 10000;
7
8  FIFO_transaction f_txn = new();
9
10 initial begin
11
12     //Assert Reset - Initial State
13     assert_reset;
14
15
16
17     /*TEST 0:  - Checks (rst_n) Functionality
18     | | |      - Randomization (of all inputs) is done Under No Constraints
19     */
20     stimulus_gen_reset;
21
22
23
24     //Deassert Reset
25     deassert_reset;
26
27
28
29
30     /*TEST 1:  - Checks when wr_en is high and rd_en is low
31     | | |      - Write Operations are done until the FIFO is full
32     */
33     stimulus_gen1;
34
35
36
37
38     /*TEST 2:  - Checks when rd_en is high and wr_en is low
39     | | |      - Read Operations are done until the FIFO is empty
40     */
41     stimulus_gen2;
42
43
44
45
46     /*TEST 3:  - Checks when both wr_en and rd_en are high
47     | | |      - Write and Read Operations are done in parallel
48     | | |      - Write Operations stop when FIFO is full
49     | | |      - Read Operations stop when FIFO is empty
50     */
51     stimulus_gen3;
52
53
54
55
56     /*TEST 4:  - Checks whole operation when all inputs are randomized
57     */
58     stimulus_gen4;
59
60
61
62     test_finished = 1;
63 end
64

```

CODE SNIPPETS

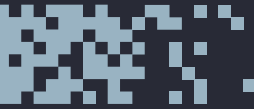
TESTBENCH

```

65
66 task assert_reset;
67     ft.rst_n = 0;
68     f_txn.constraint_mode(0);
69     @(negedge ft.clk);
70 endtask
71
72 task stimulus_gen_reset;
73     for(int i=0; i<TESTS; i++) begin
74         assert(f_txn.randomize());
75         ft.wr_en = f_txn.wr_en;
76         ft.rd_en = f_txn.rd_en;
77         ft.data_in = f_txn.data_in;
78
79         @(negedge ft.clk);
80     end
81 endtask
82
83 task deassert_reset;
84     ft.rst_n = 1;
85     f_txn.constraint_mode(1);
86     @(negedge ft.clk);
87 endtask
88
89
90
91
92 task stimulus_gen1;
93     ft.wr_en = 1; ft.rd_en = 0;
94     for(int i=0; i<TESTS/4; i++) begin
95         assert(f_txn.randomize());
96         ft.rst_n = f_txn.rst_n;
97         ft.data_in = f_txn.data_in;
98         @(negedge ft.clk);
99     end
100 endtask
101
102
103 task stimulus_gen2;
104     ft.wr_en = 0; ft.rd_en = 1;
105     for(int i=0; i<TESTS/4; i++) begin
106         assert(f_txn.randomize());
107         ft.rst_n = f_txn.rst_n;
108         ft.data_in = f_txn.data_in;
109         @(negedge ft.clk);
110     end
111 endtask
112
113
114 task stimulus_gen3;
115     ft.wr_en = 1; ft.rd_en = 1;
116     for(int i=0; i<TESTS/4; i++) begin
117         assert(f_txn.randomize());
118         ft.rst_n = f_txn.rst_n;
119         ft.data_in = f_txn.data_in;
120         @(negedge ft.clk);
121     end
122 endtask
123
124
125 task stimulus_gen4;
126     for(int i=0; i<TESTS; i++) begin
127         assert(f_txn.randomize());
128         ft.rst_n = f_txn.rst_n;
129         ft.wr_en = f_txn.wr_en;
130         ft.rd_en = f_txn.rd_en;
131         ft.data_in = f_txn.data_in;
132         @(negedge ft.clk);
133     end
134 endtask
135
136 endmodule

```


CODE SNIPPETS



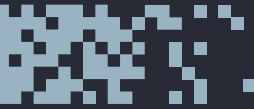
MONITOR

```

1  import transaction_pkg::*;
2  import coverage_pkg::*;
3  import scoreboard_pkg::*;
4  import shared_pkg::*;
5
6
7  module FIFO_monitor(FIFO_if.MON fm);
8      FIFO_transaction f_txn = new();
9      FIFO_coverage f_cov = new();
10     FIFO_scoreboard f_score = new();
11
12     FIFO_transaction f_tcov = new();
13
14     initial begin
15         forever begin
16             fork
17                 //Transaction for Checker
18                 begin
19                     @(edge fm.clk);
20                     //Inputs to DUT
21                     f_txn.rst_n = fm.rst_n;
22                     f_txn.wr_en = fm.wr_en;
23                     f_txn.rd_en = fm.rd_en;
24                     f_txn.data_in = fm.data_in;
25
26                     //Combinational Outputs of DUT
27                     if(!fm.rst_n) begin
28                         f_txn.wr_ack = fm.wr_ack;
29                         f_txn.overflow = fm.overflow;
30                         f_txn.underflow = fm.underflow;
31                         f_txn.data_out = fm.data_out;
32                     end
33                     f_txn.full = fm.full;
34                     f_txn.empty = fm.empty;
35                     f_txn.almostfull = fm.almostfull;
36                     f_txn.almostempty = fm.almostempty;
37
38                     if(~fm.clk) begin
39                         //Sequential Outputs of DUT
40                         if(fm.rst_n) begin
41                             f_txn.wr_ack = fm.wr_ack;
42                             f_txn.overflow = fm.overflow;
43                             f_txn.underflow = fm.underflow;
44                             f_txn.data_out = fm.data_out;
45                         end
46                     end
47                 end
48                 if(~fm.clk) f_score.check_data(f_txn);
49
50                 if(shared_pkg::test_finished) begin
51                     $display("-----");
52                     $display("-----Correct Count and Error Count Summary-----");
53                     $display("-----");
54                     $display("wr_ack: \t\tCorrect Count = %0d, Error Count = %0d", shared_pkg::correct_count_wr_ack, shared_pkg::error_count_wr_ack);
55                     $display("full: \t\tCorrect Count = %0d, Error Count = %0d", shared_pkg::correct_count_full, shared_pkg::error_count_full);
56                     $display("empty: \t\tCorrect Count = %0d, Error Count = %0d", shared_pkg::correct_count_empty, shared_pkg::error_count_empty);
57                     $display("almostfull: \t\tCorrect Count = %0d, Error Count = %0d", shared_pkg::correct_count_almostfull, shared_pkg::error_count_almostfull);
58                     $display("almostempty: \t\tCorrect Count = %0d, Error Count = %0d", shared_pkg::correct_count_almostempty, shared_pkg::error_count_almostempty);
59                     $display("overflow: \t\tCorrect Count = %0d, Error Count = %0d", shared_pkg::correct_count_overflow, shared_pkg::error_count_overflow);
60                     $display("underflow: \t\tCorrect Count = %0d, Error Count = %0d", shared_pkg::correct_count_underflow, shared_pkg::error_count_underflow);
61                     $display("data_out: \t\tCorrect Count = %0d, Error Count = %0d", shared_pkg::correct_count_dout, shared_pkg::error_count_dout);
62
63                     @(edge fm.clk); $stop;
64                 end
65             end
66             //Transaction for Covergroup
67             begin
68                 @(edge fm.clk);
69                 f_tcov.rst_n = fm.rst_n;
70                 f_tcov.wr_en = fm.wr_en;
71                 f_tcov.rd_en = fm.rd_en;
72                 f_tcov.data_in = fm.data_in;
73
74                 f_tcov.wr_ack = fm.wr_ack;
75                 f_tcov.overflow = fm.overflow;
76                 f_tcov.underflow = fm.underflow;
77                 f_tcov.data_out = fm.data_out;
78
79                 f_tcov.full = fm.full;
80                 f_tcov.empty = fm.empty;
81                 f_tcov.almostfull = fm.almostfull;
82                 f_tcov.almostempty = fm.almostempty;
83
84                 f_cov.sample_data(f_tcov);
85             end
86         end
87     end
88     join_any
89 end
90
91 endmodule

```

CODE SNIPPETS



MONITOR

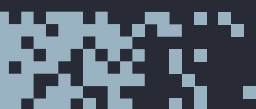
In the Monitor Module:

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

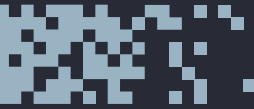
TRANSACTION

```

1  package transaction_pkg;
2      class FIFO_transaction;
3          //Parameters
4          parameter FIFO_WIDTH = 16;
5          parameter FIFO_DEPTH = 8;
6
7          //Inputs to DUT
8          rand bit rst_n, wr_en, rd_en;
9          rand logic [FIFO_WIDTH-1:0] data_in;
10
11         //Outputs of DUT
12         bit wr_ack, full, empty, almostfull, almostempty, overflow, underflow;
13         logic [FIFO_WIDTH-1:0] data_out;
14
15         //Constraints (wr_en, rd_en)
16         int WR_EN_ON_DIST = 70;
17         int RD_EN_ON_DIST = 30;
18
19         constraint rst_c {
20             rst_n          dist {0:=5,      1:=95};
21         }
22
23         constraint wr_en_c {
24             wr_en          dist {0:=(100 - WR_EN_ON_DIST),      1:=WR_EN_ON_DIST};
25         }
26
27         constraint rd_en_c {
28             rd_en          dist {0:=(100 - RD_EN_ON_DIST),      1:=RD_EN_ON_DIST};
29         }
30
31     endclass
32 endpackage

```

CODE SNIPPETS



PACKAGES

COVERAGE

```

1  package coverage_pkg;
2  import transaction_pkg::*;
3
4      class FIFO_coverage;
5
6          FIFO_transaction F_cvg_txn = new();
7
8          function void sample_data(FIFO_transaction f_tcov);
9              F_cvg_txn = f_tcov;
10             FIFO_cg.sample();
11         endfunction
12
13         covergroup FIFO_cg;
14             cross_full:      cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.full;
15             cross_empty:     cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.empty;
16             cross_almost_full: cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.almostfull;
17             cross_almost_empty: cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.almostempty;
18             cross_overflow:   cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.overflow;
19             cross_underflow:  cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.underflow;
20             cross_ack:        cross F_cvg_txn.wr_en, F_cvg_txn.rd_en, F_cvg_txn.wr_ack;
21         endgroup
22
23         function new;
24             FIFO_cg = new();
25         endfunction
26
27     endclass
28
29 endpackage

```

CODE SNIPPETS

PACKAGES

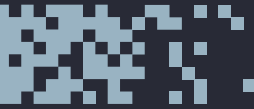
SCOREBOARD

```

1 package scoreboard_pkg;
2 import transaction_pkg::*;
3 import shared_pkg::*;
4
5 class FIFO_scoreboard;
6     parameter FIFO_WIDTH = 16;
7
8     logic wr_ack_ref, full_ref, empty_ref, almostfull_ref, almostempty_ref, overflow_ref, underflow_ref;
9     logic [FIFO_WIDTH-1:0] data_out_ref;
10
11     logic [FIFO_WIDTH-1:0] mem_q[$];
12     logic [3:0] mem_q_size, mem_size_aftr_wr, mem_size_aftr_rd;
13
14     task check_data(FIFO_transaction f_txn);
15
16         reference_model(f_txn, wr_ack_ref, full_ref, empty_ref, almostfull_ref, almostempty_ref, overflow_ref, underflow_ref, data_out_ref);
17
18         if(f_txn.wr_ack != wr_ack_ref) begin
19             $display("ERROR: Output -wr_ack- equals %0b, but should equal %0b. \t\t--time: %0t", f_txn.wr_ack, wr_ack_ref, $time);
20             error_count_wr_ack++;
21         end
22         else correct_count_wr_ack++;
23
24         if(f_txn.full != full_ref) begin
25             $display("ERROR: Output -full- equals %0b, but should equal %0b. \t\t--time: %0t", f_txn.full, full_ref, $time);
26             error_count_full++;
27         end
28         else correct_count_full++;
29
30         if(f_txn.empty != empty_ref) begin
31             $display("ERROR: Output -empty- equals %0b, but should equal %0b. \t\t--time: %0t", f_txn.empty, empty_ref, $time);
32             error_count_empty++;
33         end
34         else correct_count_empty++;
35
36     endtask
37
38     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);
39
40         if(!f_txn.rst_n) begin
41             mem_q.delete();
42             data_out_ref = 0;
43             overflow_ref = 0;
44             underflow_ref = 0;
45             wr_ack_ref = 0;
46         end
47         else begin
48             mem_q_size = mem_q.size();
49             fork
50                 //Write Operation
51                 begin
52                     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
53                         mem_q.push_front(f_txn.data_in);
54                         wr_ack_ref = 1;
55                         mem_size_aftr_wr = mem_q.size();
56                     end
57                     else begin
58                         if(f_txn.wr_en) overflow_ref = 1;
59                         else overflow_ref = 0;
60                         wr_ack_ref = 0;
61                     end
62                 end
63                 //Read Operation
64                 begin
65                     if( (f_txn.rd_en && (mem_q.size() != 0)) || (f_txn.wr_en && f_txn.rd_en && (mem_q.size() == 0)) ) begin
66                         data_out_ref = mem_q.pop_back();
67                         mem_size_aftr_rd = mem_q.size();
68                     end
69                     else begin
70                         if(f_txn.rd_en) underflow_ref = 1;
71                         else underflow_ref = 0;
72                         data_out_ref = 0;
73                     end
74                 end
75             join
76         end
77
78         if(mem_q.size() == f_txn.FIFO_DEPTH) full_ref = 1;
79         else full_ref = 0;
80
81         if(mem_q.size() == (f_txn.FIFO_DEPTH - 1)) almostfull_ref = 1;
82         else almostfull_ref = 0;
83
84         if(mem_q.size() == 0) empty_ref = 1;
85         else empty_ref = 0;
86
87         if(mem_q.size() == 1) almostempty_ref = 1;
88         else almostempty_ref = 0;
89
90     endtask
91
92 endclass

```

CODE SNIPPETS



PACKAGES

SHARED

```
1  package shared_pkg;
2
3      bit test_finished;
4
5      int correct_count_wr_ack, error_count_wr_ack,
6          correct_count_full, error_count_full,
7          correct_count_empty, error_count_empty,
8          correct_count_almostfull, error_count_almostfull,
9          correct_count_almostempty, error_count_almostempty,
10         correct_count_overflow, error_count_overflow,
11         correct_count_underflow, error_count_underflow,
12         correct_count_dout, error_count_dout;
13
14  endpackage
```

BUG REPORT

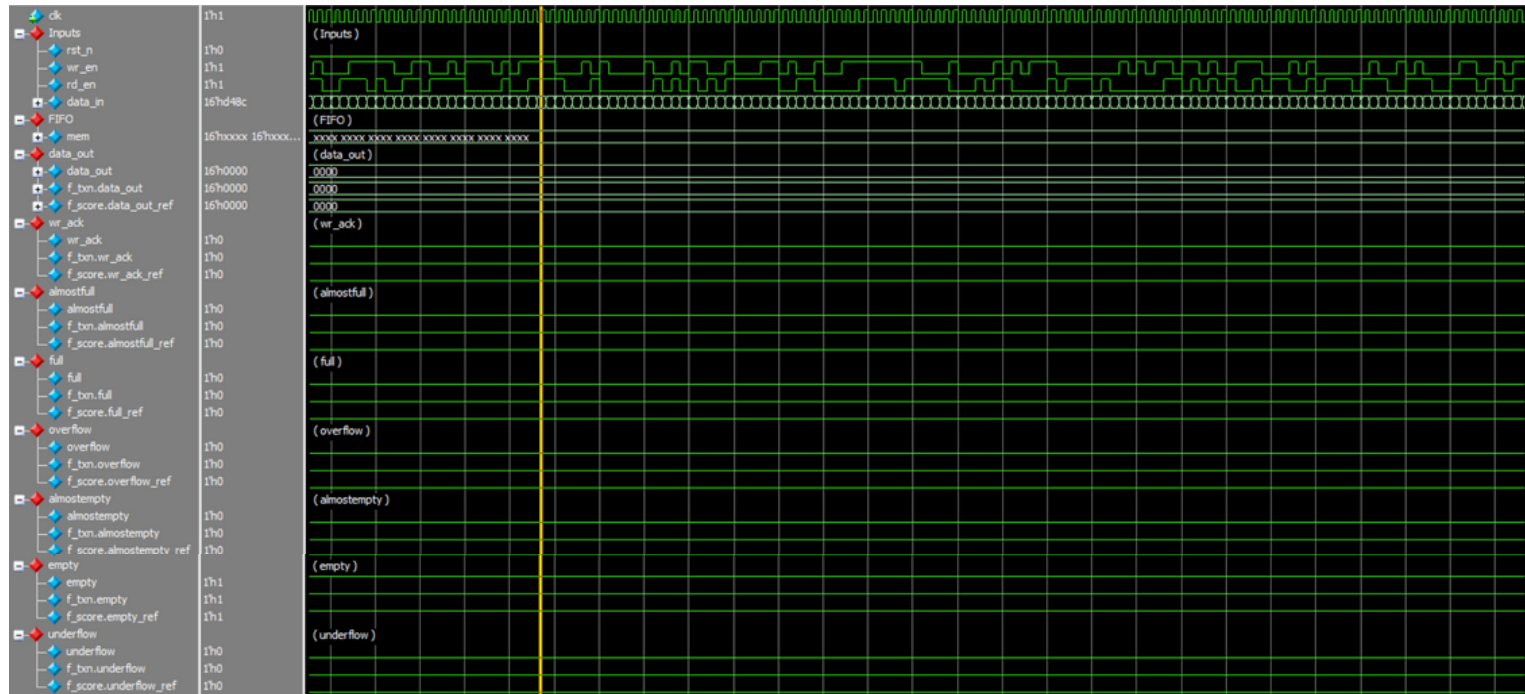
Bug	Original Code	Fix	Lines (Original Version)	Lines (Edited Version)
Sequential Outputs (wr_ack, overflow, underflow, data_out) should reset to zero on reset assertion.	<pre>always @(posedge clk or negedge rst_n) begin if (rst_n) begin wr_ptr <= 0; end always @(posedge clk or negedge rst_n) begin if (rst_n) begin rd_ptr <= 0; end end</pre>	<pre>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 end</pre>	[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.	<pre>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</pre>	<pre>//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</pre>	[32:36] [50:53]	[96:103] [123:130]
data_out should be low except when reading data.		<pre>if (fd.rd_en) fd.data_out <= 0; fd.data_out <= 0;</pre>		113 137
underflow is a sequential output not a combinational one.	assign underflow = (empty && rd_en)? 1 : 0;	<pre>else begin if (fd.empty && fd.rd_en) fd.underflow <= 1; else fd.underflow <= 0; end</pre>	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

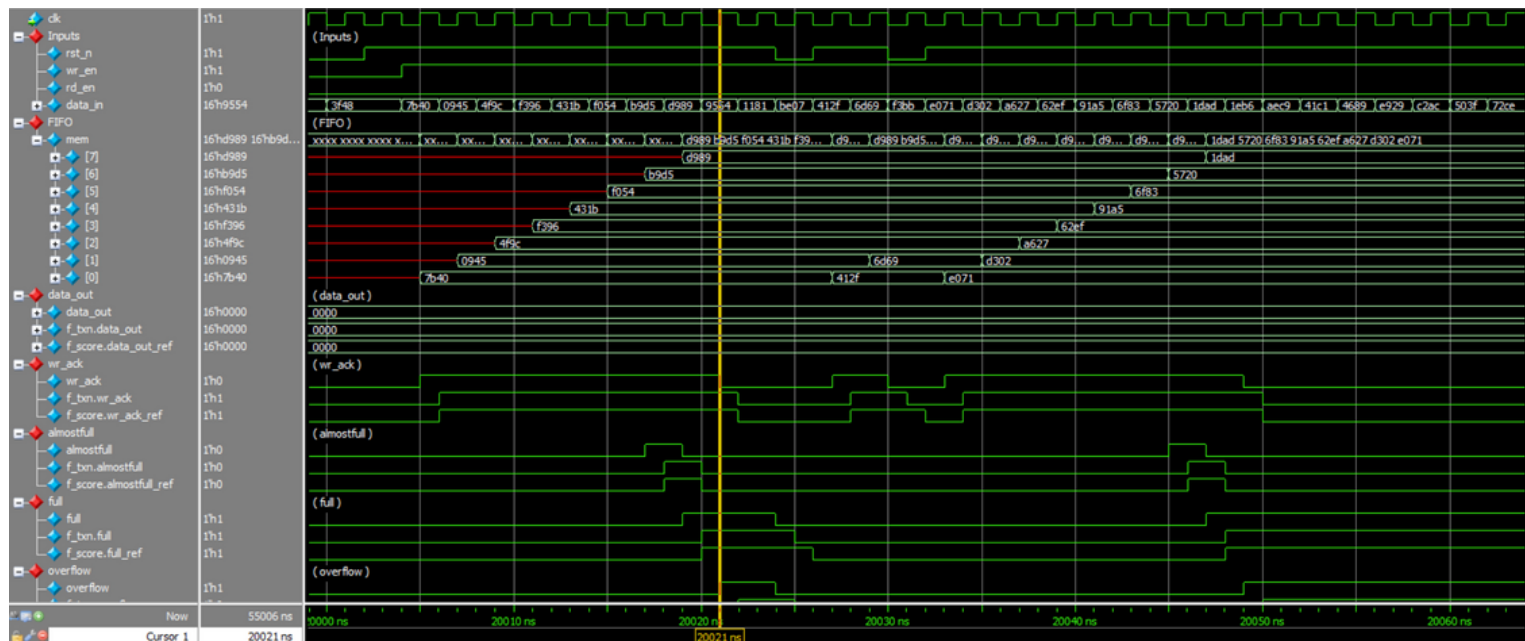
Bug	Original Code	Fix
Sequential Outputs (wr_ack, overflow, underflow, data_out) should reset to zero on reset assertion.	<pre>always @(posedge clk or negedge rst_n) begin if (rst_n) begin wr_ptr <= 0; end always @(posedge clk or negedge rst_n) begin if (rst_n) begin rd_ptr <= 0; end end</pre>	<pre>always @(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 end</pre>
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.	<pre>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</pre>	<pre>//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</pre>
data_out should be low except when reading data.		<pre>if (fd.rd_en) fd.data_out <= 0; fd.data_out <= 0;</pre>
underflow is a sequential output not a combinational one.	assign underflow = (empty && rd_en)? 1 : 0;	<pre>else begin if (fd.empty && fd.rd_en) fd.underflow <= 1; else fd.underflow <= 0; end</pre>
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

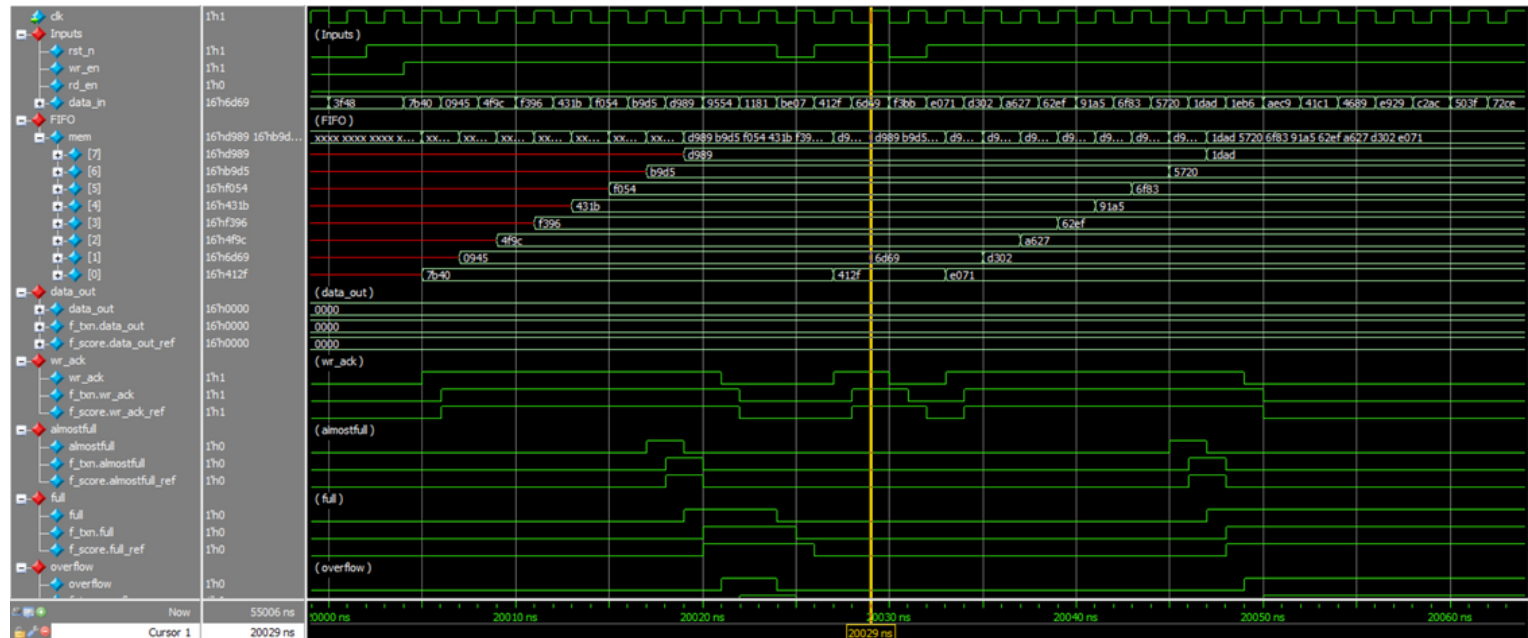


wr_en = 1, rd_en = 0: Write Op untill FIFO is full then overflow is high if wr_en is still high

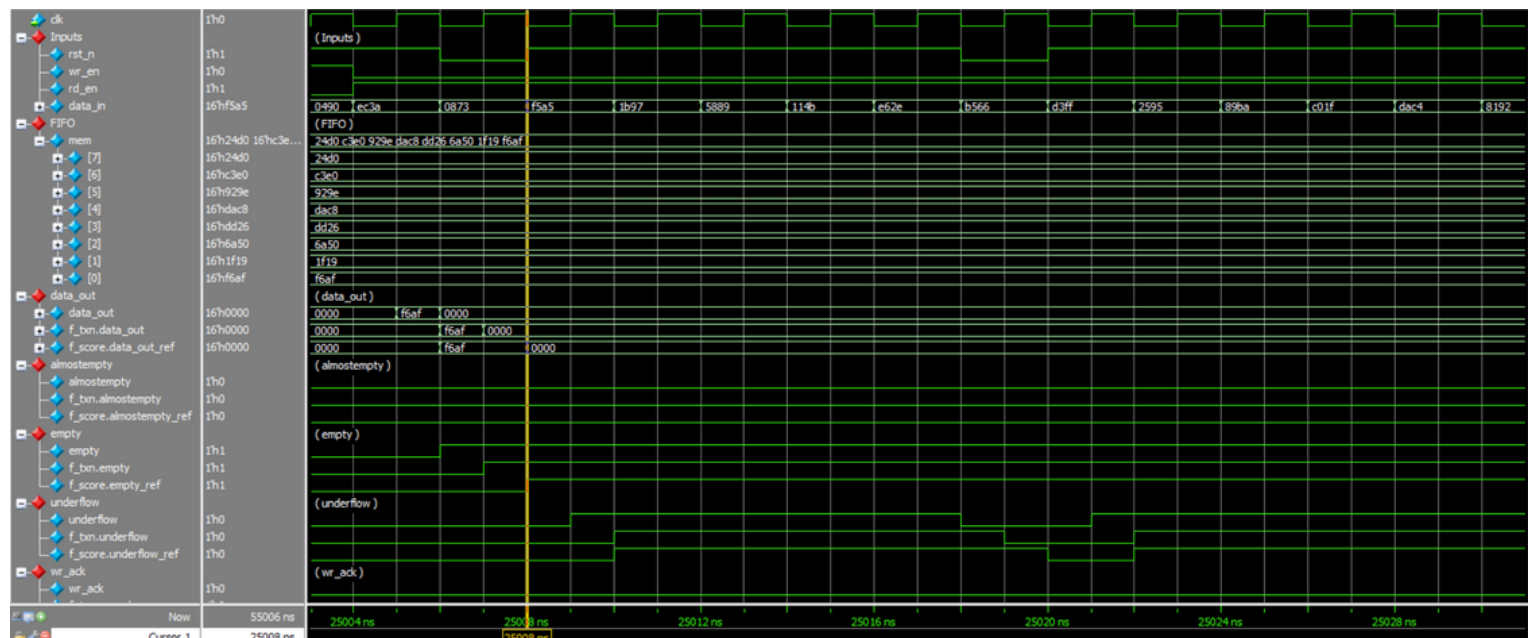


WAVEFORM SNIPPETS

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



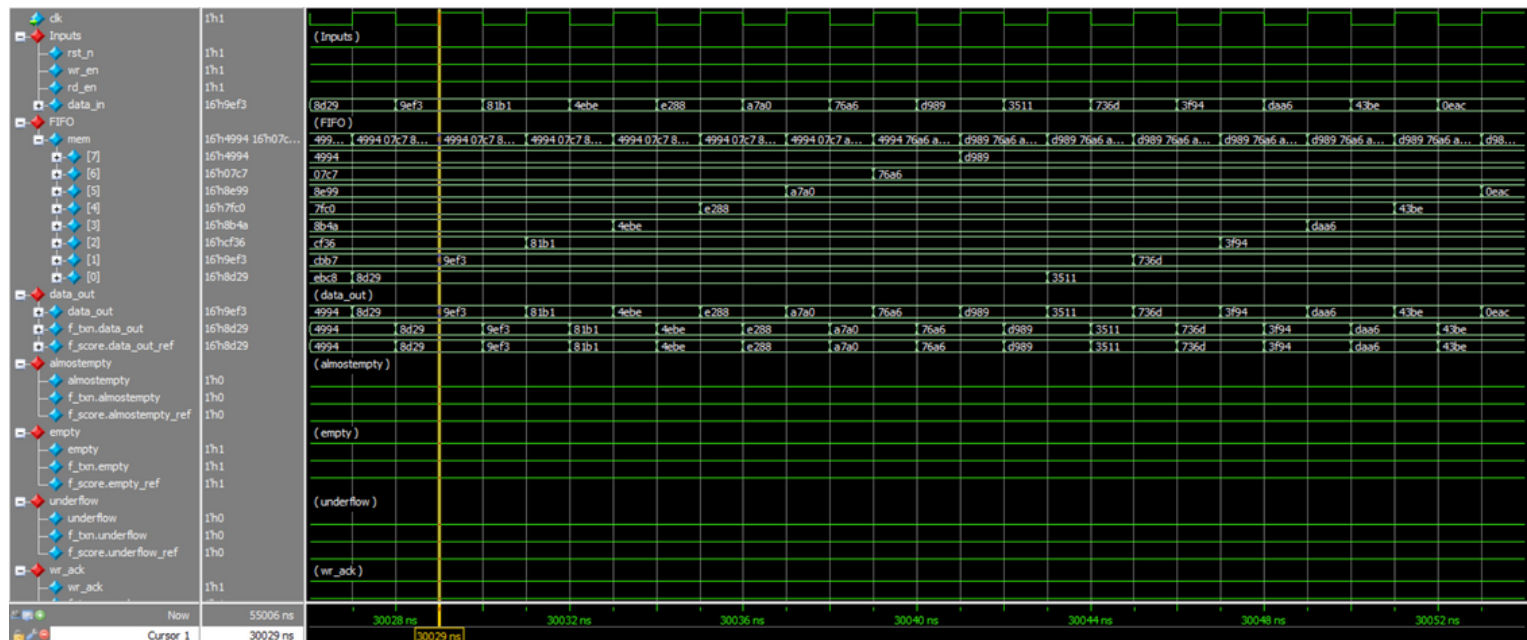
rd_en = 1, wr_en = 0: Read Op until FIFO is empty then underflow is high if rd_en is still high



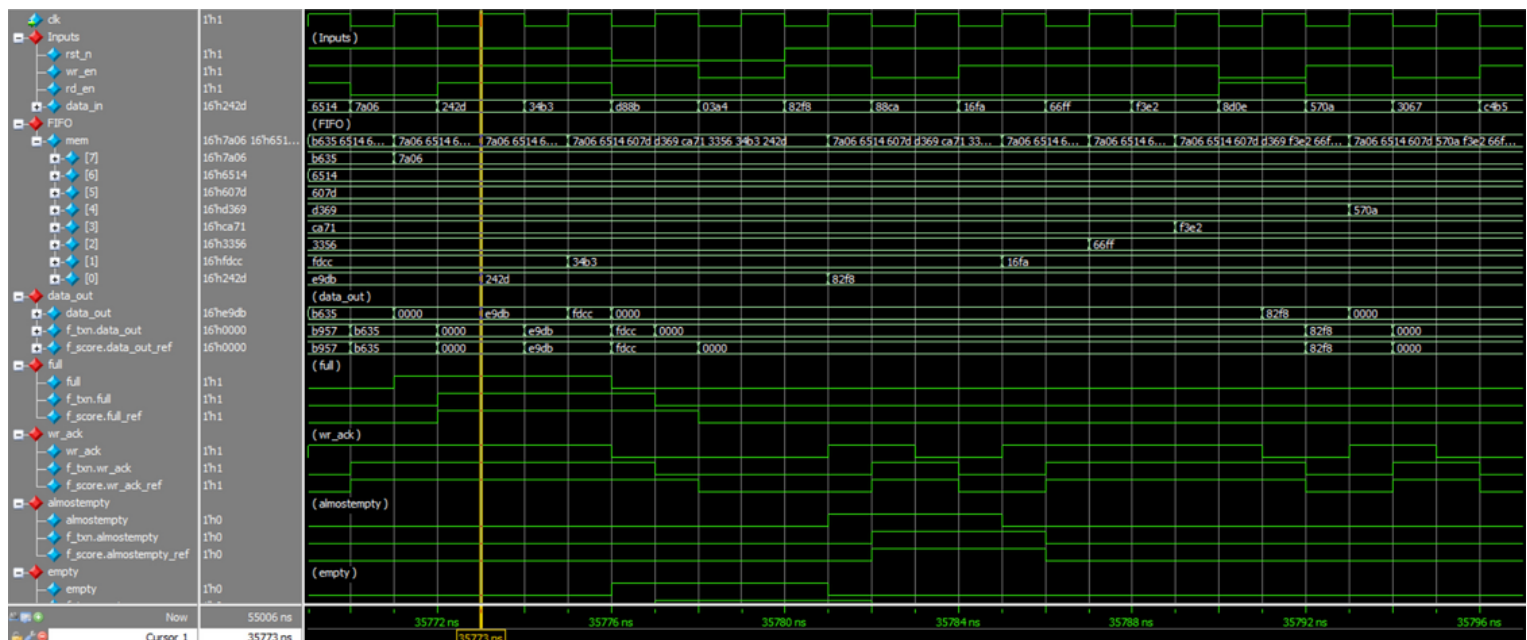
FIFO is empty on reset assertion, so data has to be written in the FIFO for a read op to occur

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

43	Assertion Coverage:				
44	Assertions	11	11	0	100.00%
367	Directive Coverage:				
368	Directives	11	11	0	100.00%
369					
71	Branch Coverage:				
72	Enabled Coverage	Bins	Hits	Misses	Coverage
73	-----	----	----	-----	-----
74	Branches	29	29	0	100.00%
75					
195	Condition Coverage:				
196	Enabled Coverage	Bins	Covered	Misses	Coverage
197	-----	----	----	-----	-----
198	Conditions	20	20	0	100.00%
388	Statement Coverage:				
389	Enabled Coverage	Bins	Hits	Misses	Coverage
390	-----	----	----	-----	-----
391	Statements	31	31	0	100.00%
553	Toggle Coverage:				
554	Enabled Coverage	Bins	Hits	Misses	Coverage
555	-----	----	----	-----	-----
556	Toggles	20	20	0	100.00%
557					
9222	Covergroup Coverage:				
9223	Covergroups	1	na	na	100.00%
9224	Coverpoints/Crosses	28	na	na	na
9225	Covergroup Bins	98	98	0	100.00%
9226					