EXERCISE 2 (Asynchronous FIFO)

Daniel Duggan

March 21th 2025

Contents

1 Introduction

The purpose of this exercise is to design an asynchronous First In First Out (FIFO) buffer as shown in figure 1. Such asynchronous FIFO's are typically used in designs that have a CDC i.e. were communication is required between sections of a design that run on different clock periods. This report details the implementation of an asynchronous FIFO that can be used in such circumstances as outlined in the exercise sheet found here (task 3). Furthermore, it answers the questions asked in task 2 and task 1. All code can be found on github at url: `https://github.com/duggan9265/FPGA-Design-For-Communication-Systems/tree/master/Course_work/Ex_2_Async_Fifo` Note: Updates have been made to correct the functionality of the async FIFO, which did not operate correctly originally. These updates are written in blue. Design changes are given via images from GitHub Desktop.

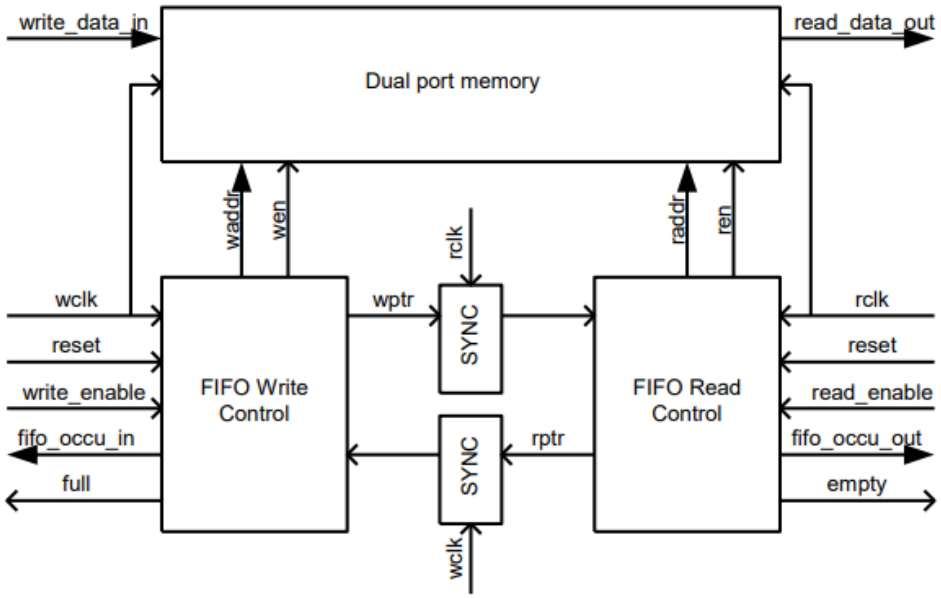


Figure 1: Block diagram of the asynchronous FIFO.

2 Task 1

Task 1 enquires as to why the circuit in figure 2 is insufficient for use in an asynchronous design. This circuitry is insufficient as the write pointer (wrptr), which is a multi-bit signal, can have propagation delays. As we have asynchronous sampling, a metastable state is more likely, as is partial or invalid data capture. The latter is shown in figure 3. It can be seen from this figure that the 3 bit wrptr has initial value (1 1 1). When it's clock goes high, this data changes to (0 0 0) i.e. 3 bits change. There is a propagation delay. When read clock (rd_clk) goes high, this propagation delay causes incorrect data to be captured. Instead of (0 0 0), (0 0 1) is captured instead. Thus data from an incorrect memory address may be read out. A metastable state would occur if the data change occurs on the same rising edge as the rd_clk i.e. violating setup or hold time requirements. Adding more flip-flops will not solve the fundamental issue of incorrect multi-bit sampling. Furthermore, it is only useful for reducing

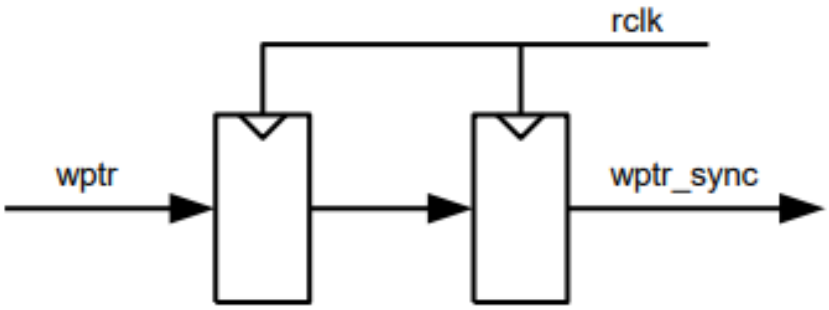


Figure 2: Circuitry that is insufficient to handle correct synchronisation in an asynchronous FIFO design.

the probability of having metastable states when transferring *single* bits between clock domains, as opposed to multiple bits as is the case here. It is required to use a circuitry that involves a binary to grey code mapping as seen in figure 4.

3 Task 2

The correct synchronisation circuitry is given in figure 4. Here, binary to grey code and grey code to binary code is used. The reason this circuitry is a better design is the binary to grey code ensures that only one bit changes at a time when the pointer is incremented or decremented which reduces the likelihood of incorrect data capture and/or metastable states. This circuitry introduces a delay however, which is very important. The longer the delay, the larger ones FIFO must be in order to not fill the buffer too early and cause data loss. Typically, one would impose a timing constraint on this delay in order for the tools to sufficiently implement the design. However, due to the small size of the FIFO, this delay should not be an issue here.

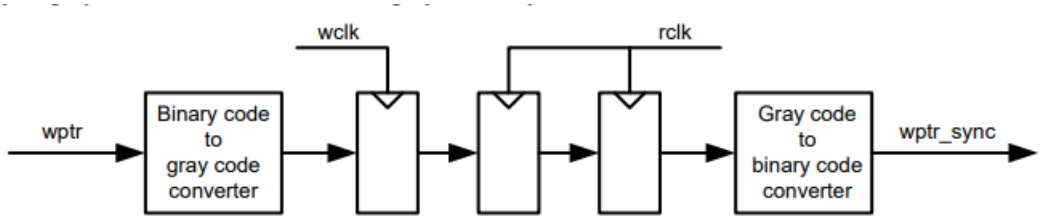


Figure 4: Synchronisation circuit for the write pointer.

3.1 Logical Equations for Binary Code to Grey Code Conversion and Grey Code to Binary Code Conversion

The logical equation for binary to grey code conversion is

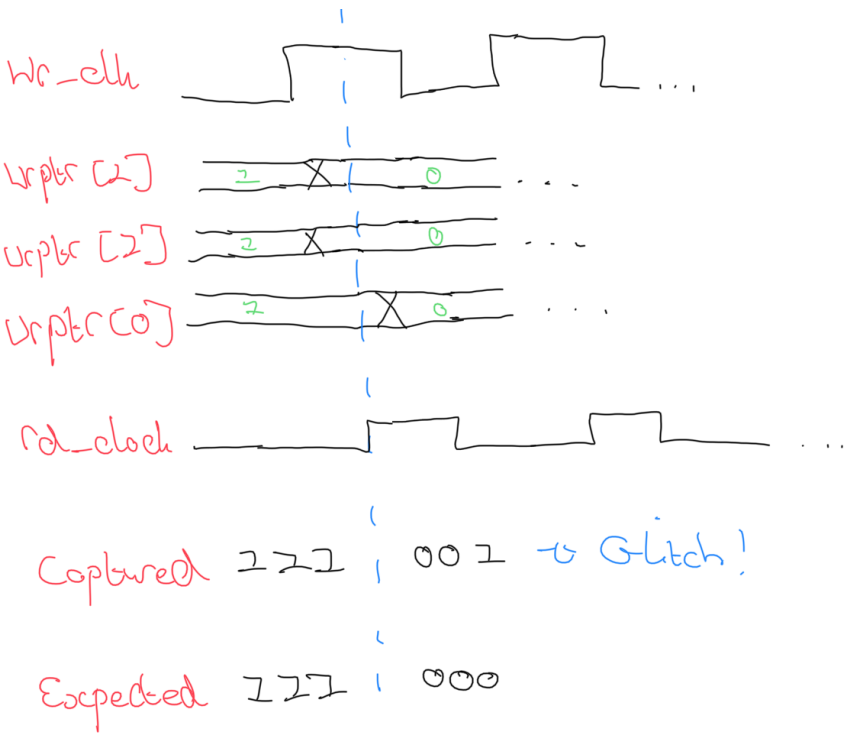


Figure 3: Example as to how circuitry in figure 2 can lead to incorrect output

$$G_{n-1} = B_{n-1}$$
$$G_i = B_{i+1} \oplus B_i$$

Here, the top equation states that the most significant bit is the same. The lower equation states that an XOR operation is conducted between bit i and bit i+1 up to n where n is the number of bits. For n=5 one has

$$G_4 = B_4$$

$$G_3 = B_4 \oplus B_3$$

$$G_2 = B_3 \oplus B_2$$

$$G_1 = B_2 \oplus B_1$$

$$G_0 = B_1 \oplus B_0$$

To do the reverse, and go from grey code to binary code, one uses:

$$B_{n-1} = G_{n-1}$$
$$B_i = G_i \oplus B_{i+1}$$

where the top equation states that the most significant bit is the same. For n=5, one has To do the reverse, and go from grey code to binary code, one uses:

$$B_4 = G_4$$

$$B_3 = G_3 \oplus B_4$$

$$B_2 = G_2 \oplus B_3$$

$$B_1 = G_1 \oplus B_2$$

$$B_0 = G_0 \oplus B_1$$

4 Task 3

For task 3, the block diagram in figure 1 is designed. A top level entity called Async_FIFO.vhd is created. This contains the external inputs RST, WCLK, RCLK, READ_ENABLE, WRITE_ENABLE, WRITE_DATA_IN and the external output READ_DATA_OUT. Instantiated within the top level are the entites for the read and write control (FIFO_read_control.vhd and FIFO_write_control.vhd respectively), the dual-port memory (blk_mem_gen_0 IP from AMD) and the read and write pointer synchronisation entities (read_pointer_sync.vhd and write_pointer_sync.vhd respectively.) Within the FIFO_write_control (FIFO_read_control) entity, the data is written (read) to the memory. Part of the synchronisation circuit in figure 4, namely that shown in figure 6 is coded within this entity, namely the binary code to grey code converter and first flip-flop.

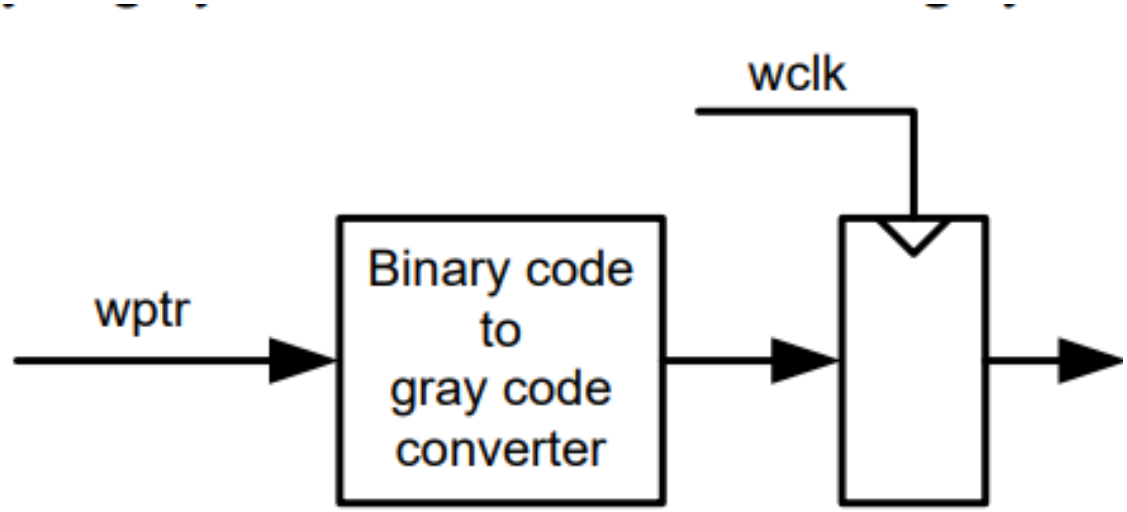


Figure 5: Partial synchronisation circuit carried out inside FIFO_write_control. The corresponding circuit for the read control is carried out inside FIFO_read_control.

```
35
36          elsif rising_edge(WCLK) then
37
38              wr_ptr_grey_code(4) <= wr_ptr_sig(4); -- Binary code to grey code
39              wr_ptr_grey_code(3) <= wr_ptr_sig(4) xor (wr_ptr_sig(3));
40              wr_ptr_grey_code(2) <= wr_ptr_sig(3) xor (wr_ptr_sig(2));
41              wr_ptr_grey_code(1) <= wr_ptr_sig(2) xor (wr_ptr_sig(1));
42              wr_ptr_grey_code(0) <= wr_ptr_sig(1) xor (wr_ptr_sig(0));
43
44              if WRITE_ENABLE(0) = '1' and full_sig = '0' then --don't write to full memory
45                  wr_ptr_sig <= (wr_ptr_sig + 1); --unsigned so naturally wraps to 0
46                  write_enable_sig <= (others => '1');
47              else
48                  write_enable_sig <= (others => '0');
49
50             end if;
51             WPTR <= wr_ptr_grey_code; -- WPTR is now in grey code. Sent to write_pointer_sync for sync
52          end if;
53      end process;
54
```

Figure 6: Code to create the partial synchronisation circuit carried out inside FIFO_write_control. The corresponding code for the read control is inside FIFO_read_control which can be viewed in the appendix.

The rest of the synchronisation circuit is contained within write_pointer_sync (read_pointer_sync). It's architecture is shown in figure 7

```
17  architecture rtl of write_pointer_sync is
18      signal wptr_ff_1 : unsigned(4 downto 0);
19      signal wptr_ff_2 : unsigned(4 downto 0);
20      signal grey2binary : unsigned(4 downto 0);
21
22  begin
23
24      second_FF_process : process (RCLK, RST)
25      begin
26          if rst = '0' then
27              WRITE_POINTER_SYNC <= (others => '0');
28              wptr_ff_1 <= (others => '0');
29              wptr_ff_2 <= (others => '0');
30
31
32          elsif rising_edge(RCLK) then
33             -- wptr_ff_0 <= WPTR;
34              wptr_ff_1 <= WPTR;
35              wptr_ff_2 <= wptr_ff_1;
36
37              -- Grey code to binary code
38              grey2binary(4) <= wptr_ff_2(4);
39              grey2binary(3) <=  wptr_ff_2(3) xor grey2binary(4);
40              grey2binary(2) <= wptr_ff_2(2) xor grey2binary(3);
41              grey2binary(1) <= wptr_ff_2(1) xor grey2binary(2);
42              grey2binary(0) <= wptr_ff_2(0) xor grey2binary(1);
43          end if;
44         WRITE_POINTER_SYNC <= grey2binary;
45      end process;
46      --WRITE_POINTER_SYNC <= grey2binary;
47  end architecture rtl;
```

Figure 7: Architecture of write_pointer_sync that completes the block diagram of figure 4 i.e. the synchronisation of the write pointer between clock domains. The corresponding code for the read pointer synchronisation is inside read_pointer_sync which can be viewed in the appendix. This design has been updated. Please see section 4.1.

4.1 Test Results from Simulation

The created entity Async_FIFO is tested in simulation using the Async_FIFO_tb.sv script. This script can be viewed in the appendix. The write operation and full flags are tested first. There are 19 data inputs for this test: 8'h11 (0), 8'h22 (1), 8'h33 (2), 8'h44 (3), 8'h55 (4), 8'h66 (5), 8'h77 (6), 8'h88 (7), 8'h99 (8), 8'haa (9), 8'hbb (10), 8'hcc (11), 8'hdd (12), 8'hee (13), 8'hff (14), 8'h01 (15), 8'h03 (16), 8'h05 (17), 8'h06 (18). As such, the last bit of data that should enter is 8'h01 (the 16th piece of data, as there are 16 memory locations in the memory. It is seen from figure 8 that this is the case. Data is read in to the correct memory address. When all 15 memory locations have been written to, and no data has been read, the FULL flag goes high as required. No more data is written to the memory.

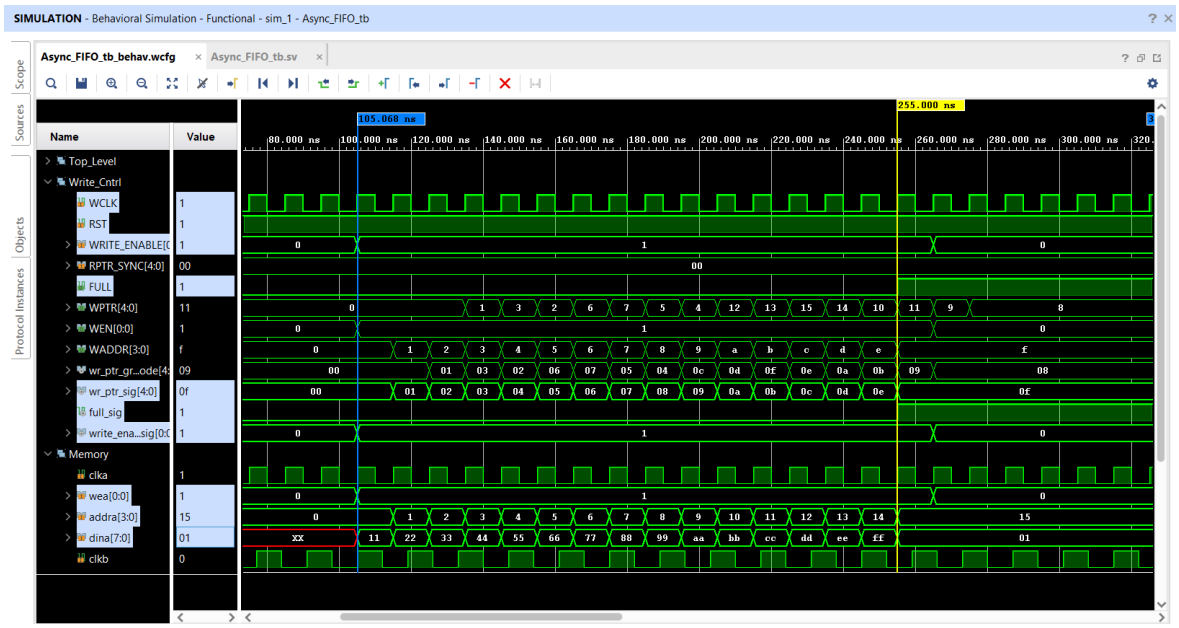


Figure 8: Simulation showing data is read in correctly, FULL flag operates correctly, and data is not read in once FULL flag is set.

Update FIFO_read_control.vhd: The design for the write enable output(i.e. WEN) has been corrected. Before, the WEN signal was delayed by 1 clock cycle, due to it being dependent on an internal signal write_enable_sig. This caused data to be read a clock cycle too late, such that the first data input was read to address 1 and not 0. To correct for this, the write pointer was delayed by 1 clock cycle by instantiating another flip flop. Therefore, data was written to the correct address. However, a better design is now used whereby write enable out (WEN) is directly dependent on the enable in port (WRITE_ENABLE(0)) and the full_sig i.e. we do not use any internal signals. Therefore the extra flip-flop is not needed. These changes can be seen in figure 9.

Figure 9: Screenshot from Git Desktop showing changes made to the design of the FIFO_write_control entity, particularly with respect to the WEN output signal.

The read side however does not work as expected (figure 10). Firstly, although the read_pointer works as required (and the RD_ADDR is thus correct), the output from the dual-port memory is delayed by 2 clock cycles. It is seen that 0x11 is read out at address 0x2 instead of 0x0. The author is unable to explain the cause of this. Furthermore, the synchronised write pointer WPTR_SYNC also has an incorrect value. This means the design inside write_pointer_sync.vhd is incorrect. Its simulation output is shown in figure 11. As WPTR_SYNC is incorrect, the empty flag is thus incorrect. The RDPTR_SYNC signal also increments incorrectly. The author did not have enough time to fully debug either of these issues. This will be done, but not in time for this report. It is not immediately clear as to the reasons for any of these issues, thus likely causes are held in reserve.

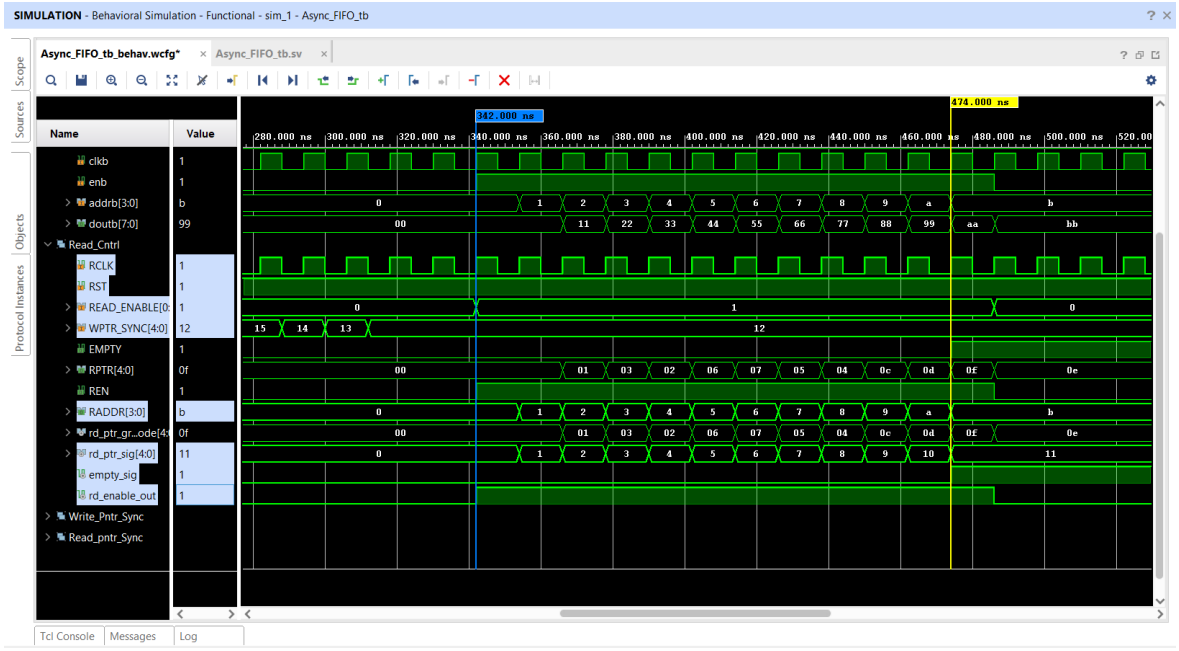


Figure 10: Simulation showing the read control side of the design does not operate as required.

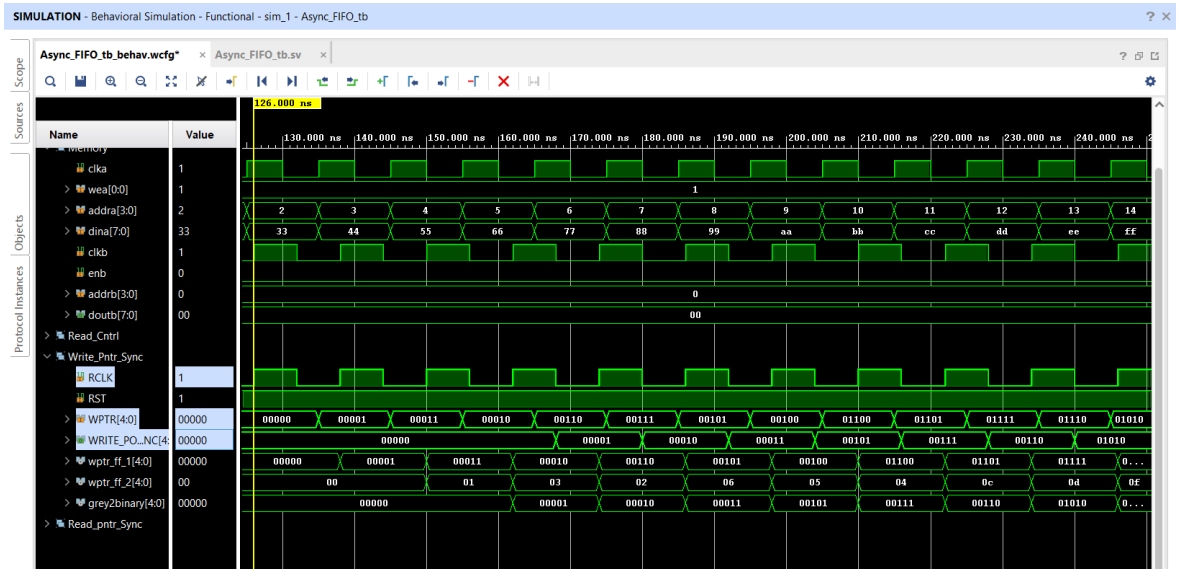


Figure 11: Simulation showing the synchronised write pointer circuitry does not operate as expected. WPTR is the input, which has undergone binary code to grey code conversion. WRITE_PO...NC = WRITE_POINTER_SYNC is the write pointer that has gone through two flip flops with the read side clock as the clock input, and through grey code to binary code conversion. It's output in decimal is 1, 2, 3, 5, 7, 6... which is clearly incorrect. The WRITE_POINTER_SYNC shows similar behaviour.

Update FIFO_write_control.vhd: The read side has been corrected. Similar to as in the FIFO_write_control, in FIFO_read_control, we remove any dependency that the read enable output REN has on internal signals. Further, within the block memory generator, the option of selecting registers on the output of the simple dual-port memory

has been selected. This is so the Async FIFO can be used for large dual port memories. This has the affect of delaying the output of the block memory (doutb) by 1 additional clock cycle (see figure 17 in appendix). The corrected testbench waveforms are shown in figure 12.

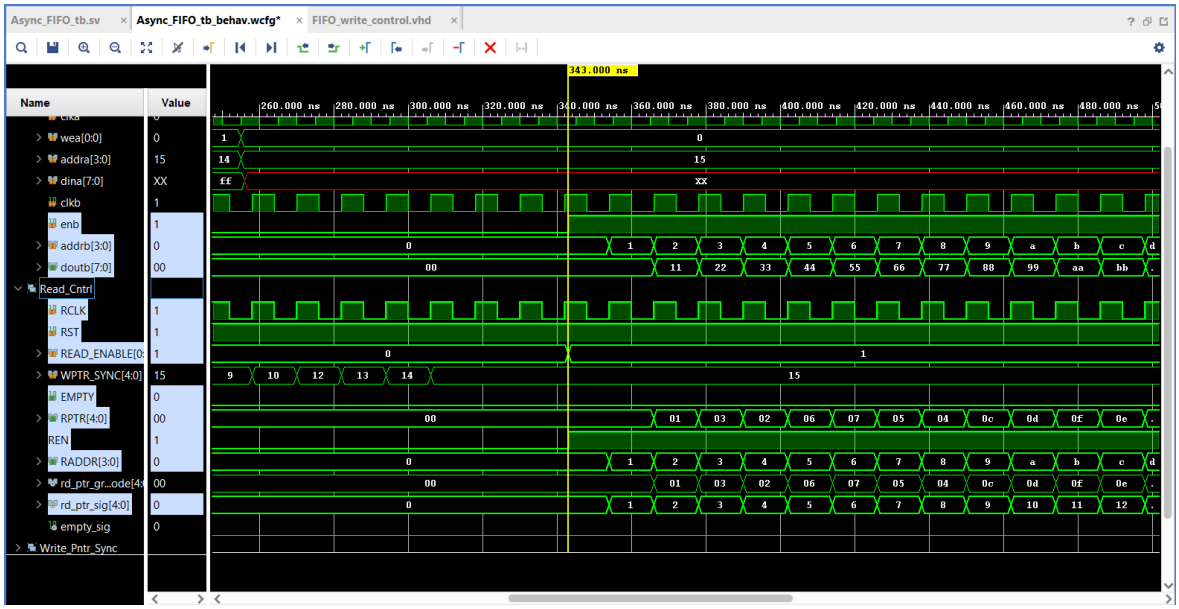
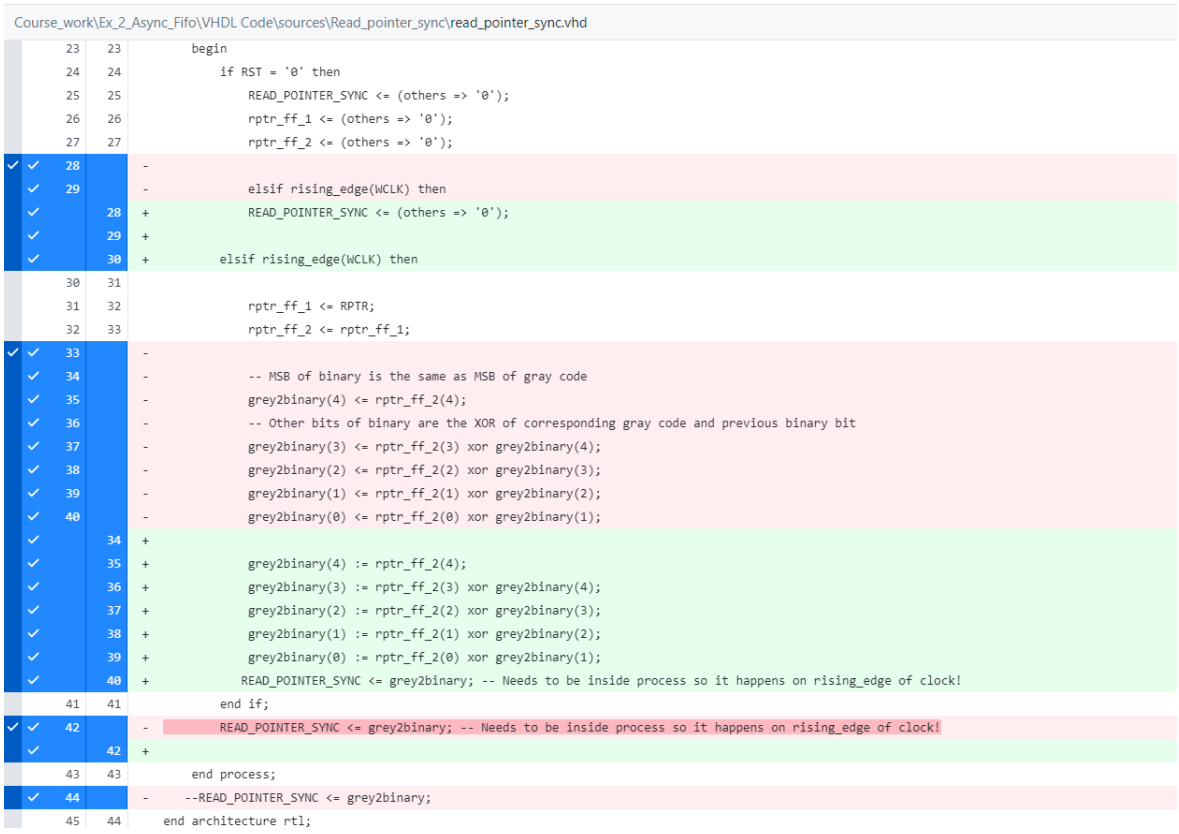


Figure 12: Waveform showing correct operation of the read control portion of the Async FIFO.

Figure 13: GitHub Desktop image showing main changes to fix the FIFO_read_control entity.

Update write_pointer_sync, read_pointer_sync : The synchronisation between the read and write point didn't was as required. This was due to an incorrect design inside the write_pointer_sync and read_pointer_sync entities. Before, the binary code output (grey2binary) was updated via signal assignments. This resulted in incorrect output

as previous values of wptr_ff_2 (the grey code after the 2nd flip-flop) were being used to do the binary code to grey code conversion as opposed to the current values. Therefore the output did not increment as required. Instead, grey2binary was redesigned as a variable to allow for it's immediate update and correct incrementation of WRITE_POINTER_SYNC and WRITE_POINTER_SYNC.



Course_work\Ex_2_Async_Fifo\VHDL Code\sources\Read_pointer_sync\read_pointer_sync.vhd

```
23 23       begin
24 24           if RST = '0' then
25 25               READ_POINTER_SYNC <= (others => '0');
26 26               rptr_ff_1 <= (others => '0');
27 27               rptr_ff_2 <= (others => '0');
28 -
29 -               elsif rising_edge(WCLK) then
   28 +            READ_POINTER_SYNC <= (others => '0');
   29 +
   30 +        elsif rising_edge(WCLK) then
30 31
31 32               rptr_ff_1 <= RPTR;
32 33               rptr_ff_2 <= rptr_ff_1;
33 -
34 -               -- MSB of binary is the same as MSB of gray code
35 -               grey2binary(4) <= rptr_ff_2(4);
36 -               -- Other bits of binary are the XOR of corresponding gray code and previous binary bit
37 -               grey2binary(3) <= rptr_ff_2(3) xor grey2binary(4);
38 -               grey2binary(2) <= rptr_ff_2(2) xor grey2binary(3);
39 -               grey2binary(1) <= rptr_ff_2(1) xor grey2binary(2);
40 -               grey2binary(0) <= rptr_ff_2(0) xor grey2binary(1);
   34 +
   35 +            grey2binary(4) := rptr_ff_2(4);
   36 +            grey2binary(3) := rptr_ff_2(3) xor grey2binary(4);
   37 +            grey2binary(2) := rptr_ff_2(2) xor grey2binary(3);
   38 +            grey2binary(1) := rptr_ff_2(1) xor grey2binary(2);
   39 +            grey2binary(0) := rptr_ff_2(0) xor grey2binary(1);
   40 +           READ_POINTER_SYNC <= grey2binary; -- Needs to be inside process so it happens on rising_edge of clock!
41 41           end if;
42 -           READ_POINTER_SYNC <= grey2binary; -- Needs to be inside process so it happens on rising_edge of clock!
   42 +
43 43       end process;
44 -    --READ_POINTER_SYNC <= grey2binary;
45 44   end architecture rtl;
```

Figure 14: Updates to the read_point_sync entity. Similar changes are made in the write_point_sync entity.

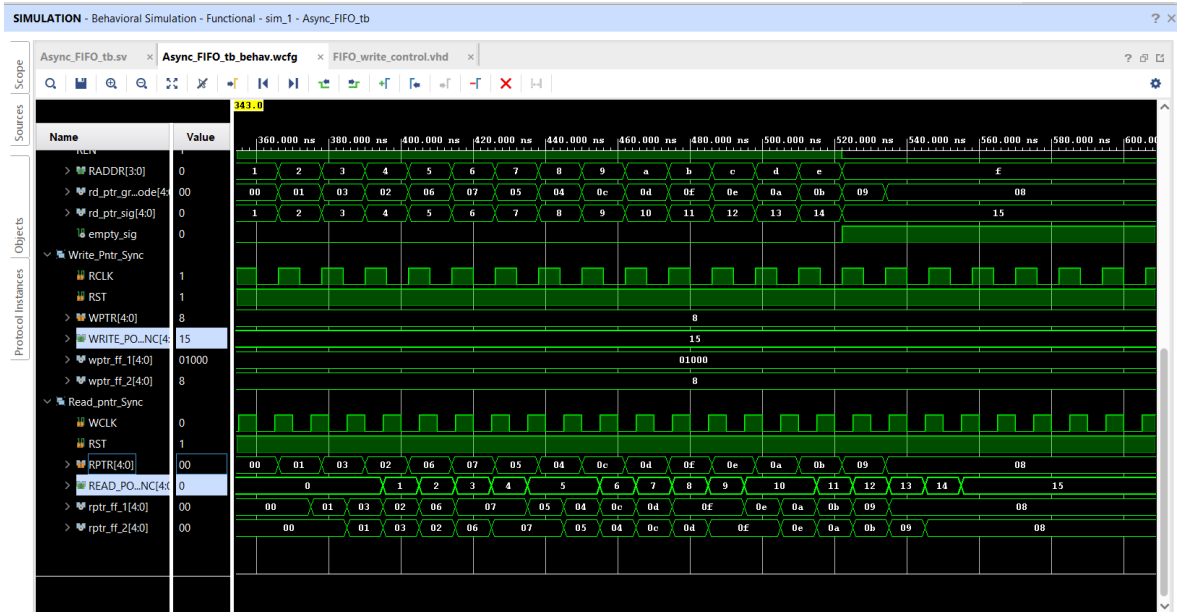


Figure 15: Waveform showing correct operation of the synchronised read and write pointers..

Update to testbench: Lastly, some slight changes were made to the testbench. The original testbench caused delta cycles when inputting data to the DUT. Thus, a slight delay has been added before the if statement to write data, and before the if statement for reading data. Further, when data is not being written, X's i.e. unknowns are read out. This allows for easier viewing of incorrect operation if it exists.

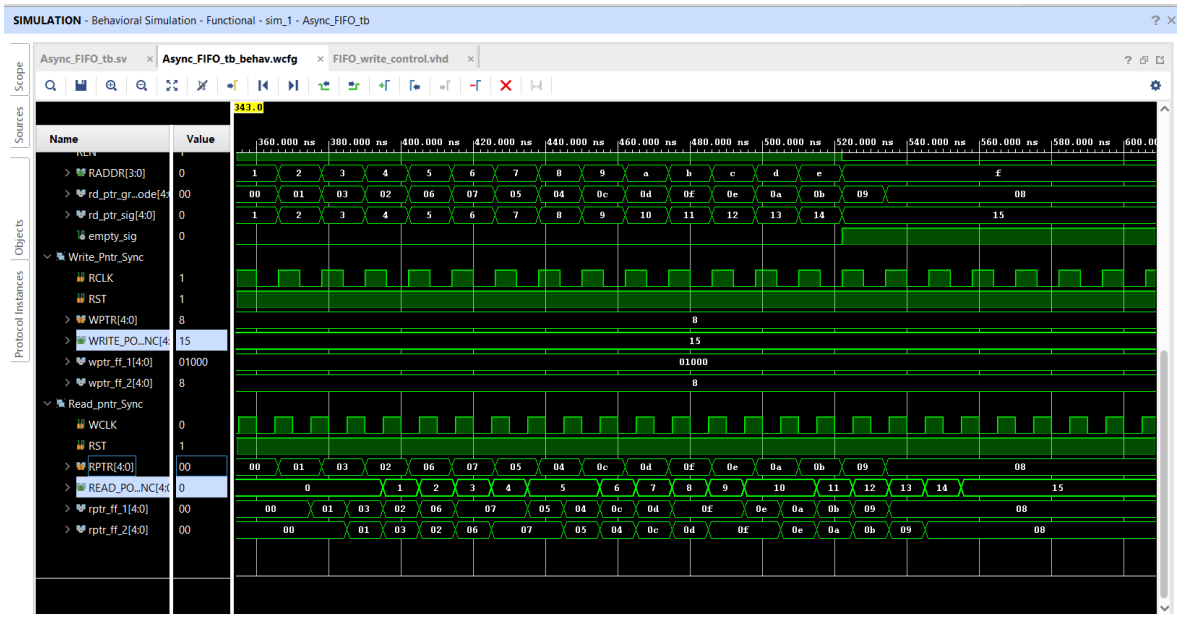


Figure 16: Changes made to the test bench.

5 Appendix

5.1 Async_FIFO.vhd

```
1  -- vhdl-linter-disable type-resolved component. component
2  -- Author Daniel Duggan
3  library IEEE;
4  use ieee.std_logic_1164.all;
5  use ieee.numeric_std.all;
6  -- LIBRARY blk_mem_gen_v8_4_7;
7  -- USE blk_mem_gen_v8_4_7.blk_mem_gen_v8_4_7;
8
9  entity Async_FIFO is
10     port (
11         RST_top : in std_logic;
12         WCLK_top : in std_logic;
13         RCLK_top : in std_logic;
14         WRITE_ENABLE_TOP : in std_logic_vector(0 downto 0);
15         READ_ENABLE_TOP : in std_logic_vector(0 downto 0);
16         FULL_top : out std_logic; --output if memory is full
17         EMPTY_top : out std_logic; -- output if memory is empty
18         WRITE_DATA_IN_top : in std_logic_vector(7 downto 0);
19         WRITE_DATA_OUT_top : out std_logic_vector(7 downto 0)
20     );
21 end entity;
22
23 architecture rtl of Async_FIFO is
24
25     -- Declare the component
26     COMPONENT blk_mem_gen_0
27         PORT (
28             clka   : IN std_logic;
29             wea    : IN std_logic_vector(0 DOWNTO 0);
30             addra  : IN std_logic_vector(3 DOWNTO 0);
31             dina   : IN std_logic_vector(7 DOWNTO 0);
32             clkb   : IN std_logic;
33             enb    : IN std_logic;
34             addrb  : IN std_logic_vector(3 DOWNTO 0);
35             doutb  : OUT std_logic_vector(7 DOWNTO 0)
36         );
37     END COMPONENT;
38
39     -- FIFO_WRITE_CONTROL SIGNALS.
40     signal wen_sig : std_logic_vector(0 downto 0); -- write enable from
       FIFO_WRITE_Control
```

```
41       --write address from FIFO_WRITE_Control
42      signal wr_pointer_sig : unsigned(4 downto 0); --write address from
            FIFO_WRITE_Control are bits (3 downto 0)
43      signal wr_addr_a : unsigned(3 downto 0);
44      signal wr_pointer_sync : unsigned(4 downto 0);
45
46      --FIFO_READ_CONTROL SIGNALS.
47      signal ren_sig : std_logic; -- write enable from FIFO_READ_CONTROL
48      signal rd_pointer_sig : unsigned(4 downto 0);
49      signal rd_addr_b : unsigned(3 downto 0);
50      signal rd_pointer_sync : unsigned(4 downto 0);
51
52  begin
53
54      Fifo_write_control_inst : entity work.FIFO_write_control
55      port map(
56          WCLK => WCLK_top,
57          RST => RST_top,
58          WRITE_ENABLE => WRITE_ENABLE_TOP,
59          RPTR_SYNC => rd_pointer_sync,  --Input from Read_pointer_sync.
                Synchronised read pointer.
60          FULL => FULL_top,
61          WPTR => wr_pointer_sig, --Write pointer goes to the sync
62          WEN => wen_sig, -- goes to block_mem via sig wen_sig in async_FIFO
63          WADDR => wr_addr_a -- Output to addra of Memory (Write Address).
64
65      );
66
67      Dual_port_memory_inst : blk_mem_gen_0 -- vhdl-linter-disable-line not-
            declared
68
69      port map(
70          clka => WCLK_top,
71          wea => wen_sig, -- write enable from FIFO_WRITE_Control
72          addra => std_logic_vector(wr_addr_a), -- write address from
                FIFO_WRITE_Control
73          dina => WRITE_DATA_IN_top,
74          clkb => RCLK_top,
75          enb => ren_sig, -- read enable from FIFO_READ_CONTROL
76          addrb => std_logic_vector(rd_addr_b), --read address from
                FIFO_READ_CONTROL
77          doutb => WRITE_DATA_OUT_top
78      );
79
```

```
80      Fifo_read_control_inst : entity work.FIFO_read_control -- vhdl-linter-
            disable-line not-declared
81      port map(
82          RCLK => RCLK_top,
83          RST => RST_top,
84          READ_ENABLE => READ_ENABLE_TOP, -- read enable from TOP
85          REN => ren_sig, -- write enable to BLOCK_MEM
86          WPTR_SYNC => wr_pointer_sync, -- Input sig from WRITE_POINTER_SYNC.
                To determine occupancy and empty/full.
87          EMPTY => EMPTY_top, -- Output sig. Goes to TOP
88          RPTR => rd_pointer_sig, -- output sig to Read_pointer_sync entity.
                Signal to be synchronised with wr_pointer_sig
89          RADDR => rd_addr_b  -- Output sig to Dual port memory. Read address.
90
91      );
92
93      Write_pointer_sync_inst : entity work.write_pointer_sync
94      port map(
95              RCLK => RCLK_top,
96              --WCLK => WCLK_top,
97              RST => RST_top,
98              WPTR => wr_pointer_sig,
99              WRITE_POINTER_SYNC => wr_pointer_sync
100     );
101
102     Read_pointer_sync_inst : entity work.read_pointer_sync
103     port map(
104             --RCLK => RCLK_top,
105             WCLK => WCLK_top,
106             RST => RST_top,
107             RPTR => rd_pointer_sig, -- input from FIFO_read_control.
108             READ_POINTER_SYNC => rd_pointer_sync -- Output to
                    FIFO_write_control. Synchronised read_pointer.
109     );
110
111 end architecture rtl;
```

5.2 FIFO_write_control.vhd

```
1  -- vhdl-linter-disable type-resolved
2  -- Author Daniel Duggan
3  library ieee;
4  use ieee.std_logic_1164.all;
5  use ieee.numeric_std.all;
```

```
6
7  entity FIFO_WRITE_CONTROL is
8      port (
9          WCLK : in std_logic;
10         RST : in std_logic;
11         WRITE_ENABLE : in std_logic_vector(0 downto 0);
12         RPTR_SYNC : in unsigned(4 downto 0); --rd pointer that comes from
               the sync
13         FULL : out std_logic;
14         WPTR : out unsigned(4 downto 0); --Write pointer goes to the sync i.
               e. grey-coded
15         WEN : out std_logic_vector(0 downto 0); -- goes to block_mem via sig
               wen_sig in async_FIFO
16         WADDR : out unsigned(3 downto 0) -- write address. Goes to block_mem
               via signal waddr_sig in async_FIFO
17     );
18 end entity;
19
20 architecture rtl of FIFO_WRITE_CONTROL is
21
22     signal wr_ptr_grey_code : unsigned(4 downto 0);
23     signal wr_ptr_sig : unsigned(4 downto 0);
24     signal full_sig : std_logic;
25
26
27 begin
28
29     write_control_process : process (WCLK,RST) --asyn reset
30     begin
31             if RST = '0' then --reset active low
32                 wr_ptr_grey_code <= (others => '0');
33                 wr_ptr_sig <= (others => '0');
34
35
36
37
38             elsif rising_edge(WCLK) then
39
40                 wr_ptr_grey_code(4) <= wr_ptr_sig(4); -- Binary code to grey
                       code
41                 wr_ptr_grey_code(3) <= wr_ptr_sig(4) xor (wr_ptr_sig(3));
42                 wr_ptr_grey_code(2) <= wr_ptr_sig(3) xor (wr_ptr_sig(2));
43                 wr_ptr_grey_code(1) <= wr_ptr_sig(2) xor (wr_ptr_sig(1));
44                 wr_ptr_grey_code(0) <= wr_ptr_sig(1) xor (wr_ptr_sig(0));
45
```

```
46                  if WRITE_ENABLE(0) = '1' and full_sig = '0' then --don't
                        write to full memory
47                      wr_ptr_sig <= (wr_ptr_sig + 1);
48                  end if;
49                  WPTR <= wr_ptr_grey_code; -- WPTR is now in grey code. Sent
                        to write_pointer_sync for sync
50              end if;
51      end process;
52
53      full_sig <= '1' when (wr_ptr_sig - RPTR_SYNC = 15) else '0';
54
55
56      --WPTR <= wr_ptr_grey_code; -- WPTR is now in grey code. Sent to
            write_pointer_sync for sync
57      FULL <= full_sig;
58      WADDR <= (wr_ptr_sig(3 downto 0)); -- sent to the Dual-port memory
59      WEN <=  (others =>'1') when (WRITE_ENABLE(0)='1' AND full_sig = '0')
            else (others => '0'); --sent to the Dual-port memory '1' when (
            rd_ptr_sig_delay = WPTR_SYNC) else '0';
60  end architecture rtl;
```

5.3 FIFO_read_control.vhd

```
1   -- vhdl-linter-disable type-resolved
2   -- Author Daniel Duggan
3   library ieee;
4   use ieee.std_logic_1164.all;
5   use ieee.numeric_std.all;
6
7   entity FIFO_READ_CONTROL is
8       port (
9           RCLK : in std_logic;
10          RST : in std_logic;
11          READ_ENABLE : in std_logic_vector(0 downto 0);
12          WPTR_SYNC : in unsigned(4 downto 0); --wp pointer that comes from
                the sync
13          EMPTY : out std_logic;
14          RPTR : out unsigned(4 downto 0); --Write pointer goes to the sync i.
                e. grey-coded
15          REN : inout std_logic; -- goes to block_mem via sig wen_sig in
                async_FIFO
16          RADDR : out unsigned(3 downto 0) -- write address. Goes to block_mem
                 via signal waddr_sig in async_FIFO
17      );
```

```
18 end entity;
19
20 architecture rtl of FIFO_READ_CONTROL is
21
22     signal rd_ptr_grey_code : unsigned(4 downto 0);
23     signal rd_ptr_sig : unsigned(4 downto 0);
24     signal empty_sig : std_logic;
25
26
27 begin
28
29     FIFO_read_control_process : process (RCLK, RST) --asyn reset
30     begin
31         if RST = '0' then --reset active low
32             --RPTR <= (others => '0');
33             rd_ptr_grey_code <= (others => '0');-- this gives multiple load
                   error
34             rd_ptr_sig <= (others => '0'); --this gives multiple load error
35
36
37
38         elsif rising_edge(RCLK) then
39
40             rd_ptr_grey_code(4) <= rd_ptr_sig(4); -- Binary code to grey
                   code
41             rd_ptr_grey_code(3) <= rd_ptr_sig(4) xor (rd_ptr_sig(3));
42             rd_ptr_grey_code(2) <= rd_ptr_sig(3) xor (rd_ptr_sig(2));
43             rd_ptr_grey_code(1) <= rd_ptr_sig(2) xor (rd_ptr_sig(1));
44             rd_ptr_grey_code(0) <= rd_ptr_sig(1) xor (rd_ptr_sig(0));
45
46             if READ_ENABLE(0) = '1' and empty_sig = '0' then --don't read
                   from empty memory
47                 rd_ptr_sig <= (rd_ptr_sig + 1); --unsigned so naturally
                       wraps to 0
48             end if;
49         end if;
50
51     end process;
52
53     empty_sig <= '1' when (rd_ptr_sig = WPTR_SYNC) else '0'; --Check if
           empty or not. Uses synced wr_ptr which has 5 cc delay
54     RADDR <= (rd_ptr_sig(3 downto 0));
55     RPTR <= rd_ptr_grey_code; -- RPTR is now in grey code.
56     EMPTY <= empty_sig;
```

```
57      REN <= '1' when (READ_ENABLE(0)='1' AND empty_sig ='0') else '0'; --use
            rd_eable
58  end architecture rtl; --all 5 bits and equal, empty, all 5 bits and
        difference is 16 is full.
```

5.4 write_pointer_sync.vhd

```
1   -- vhdl-linter-disable type-resolved
2   -- Author Daniel Duggan
3   library IEEE;
4   use ieee.std_logic_1164.all;
5   use ieee.numeric_std.all;
6
7   entity write_pointer_sync is
8       port (
9           RCLK : in std_logic;
10          --WCLK : in std_logic;
11          RST : in std_logic;
12          WPTR : in unsigned(4 downto 0);
13          WRITE_POINTER_SYNC : out unsigned(4 downto 0)
14
15      );
16  end entity;
17
18  architecture rtl of write_pointer_sync is
19      signal wptr_ff_1 : unsigned(4 downto 0);
20      signal wptr_ff_2 : unsigned(4 downto 0);
21  begin
22
23      second_FF_process : process (RCLK, RST)
24          variable grey2binary : unsigned(4 downto 0);
25      begin
26          if rst = '0' then
27              WRITE_POINTER_SYNC <= (others => '0');
28              wptr_ff_1 <= (others => '0');
29              wptr_ff_2 <= (others => '0');
30          elsif rising_edge(RCLK) then
31              -- wptr_ff_0 <= WPTR;
32              wptr_ff_1 <= WPTR;
33              wptr_ff_2 <= wptr_ff_1;
34
35              -- Grey code to binary code
36              grey2binary(4) := wptr_ff_2(4);
37              grey2binary(3) := wptr_ff_2(3) xor grey2binary(4);
```

```
38              grey2binary(2) := wptr_ff_2(2) xor grey2binary(3);
39              grey2binary(1) := wptr_ff_2(1) xor grey2binary(2);
40              grey2binary(0) := wptr_ff_2(0) xor grey2binary(1);
41              WRITE_POINTER_SYNC <= grey2binary; -- Needs to be inside process
                    so it happens on rising_edge of clock!
42          end if;
43      end process;
44  end architecture rtl;
```

5.5 read_pointer_sync.vhd

```
1   -- vhdl-linter-disable type-resolved
2   -- Author Daniel Duggan
3   library IEEE;
4   use ieee.std_logic_1164.all;
5   use ieee.numeric_std.all;
6
7   entity read_pointer_sync is
8       port (
9           --RCLK : in std_logic;
10          WCLK : in std_logic;
11          RST : in std_logic;
12          RPTR : in unsigned(4 downto 0);
13          READ_POINTER_SYNC : out unsigned(4 downto 0)
14      );
15  end entity;
16
17  architecture rtl of read_pointer_sync is
18      signal rptr_ff_1 : unsigned(4 downto 0);
19      signal rptr_ff_2 : unsigned(4 downto 0);
20
21  begin
22      second_FF_process : process (WCLK, RST)
23          variable grey2binary : unsigned(4 downto 0);
24      begin
25          if RST = '0' then
26              READ_POINTER_SYNC <= (others => '0');
27              rptr_ff_1 <= (others => '0');
28              rptr_ff_2 <= (others => '0');
29              READ_POINTER_SYNC <= (others => '0');
30
31          elsif rising_edge(WCLK) then
32
33              rptr_ff_1 <= RPTR;
```

```
34              rptr_ff_2 <= rptr_ff_1;
35
36              grey2binary(4) := rptr_ff_2(4);
37              grey2binary(3) := rptr_ff_2(3) xor grey2binary(4);
38              grey2binary(2) := rptr_ff_2(2) xor grey2binary(3);
39              grey2binary(1) := rptr_ff_2(1) xor grey2binary(2);
40              grey2binary(0) := rptr_ff_2(0) xor grey2binary(1);
41             READ_POINTER_SYNC <= grey2binary; -- Needs to be inside process
                  so it happens on rising_edge of clock!
42          end if;
43
44      end process;
45  end architecture rtl;
```

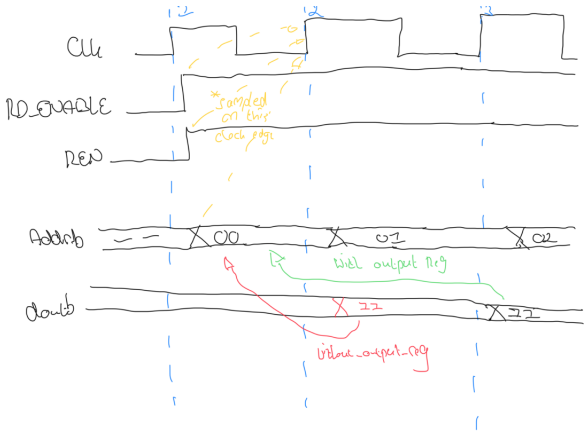
5.6 Async_FIFO_tb.sv

```
1  // Testbench for Async_FIFO
2  // Author Daniel Duggan
3  module Async_FIFO_tb;
4
5  // Signals
6  logic RST_top;
7  logic WCLK_top;
8  logic RCLK_top;
9  logic [0:0] WRITE_ENABLE_TOP;
10 logic [0:0] READ_ENABLE_TOP;
11 logic FULL_top;
12 logic EMPTY_top;
13 logic [7:0] WRITE_DATA_IN_top;
14 logic [7:0] WRITE_DATA_OUT_top;
15
16 // Clock periods
17 parameter CLK_PERIOD_WR = 10; // Write clock period (e.g., 100MHz)
18 parameter CLK_PERIOD_RD = 12; // Read clock period (e.g., 66.6MHz)
19
20 // FIFO instance
21 Async_FIFO uut (
22     .RST_top(RST_top),
23     .WCLK_top(WCLK_top),
24     .RCLK_top(RCLK_top),
25     .WRITE_ENABLE_TOP(WRITE_ENABLE_TOP),
26     .READ_ENABLE_TOP(READ_ENABLE_TOP),
27     .FULL_top(FULL_top),
28     .EMPTY_top(EMPTY_top),
```

```
29      .WRITE_DATA_IN_top(WRITE_DATA_IN_top),
30      .WRITE_DATA_OUT_top(WRITE_DATA_OUT_top)
31  );
32
33  // Clock Generation
34  always #(CLK_PERIOD_WR/2) WCLK_top = ~WCLK_top;
35  always #(CLK_PERIOD_RD/2) RCLK_top = ~RCLK_top;
36
37  // Predefined data array
38  logic [7:0] predefined_data [0:18] = '{
39      8'h11, 8'h22, 8'h33, 8'h44, 8'h55, 8'h66, 8'h77, 8'h88,
40      8'h99, 8'haa, 8'hbb, 8'hcc, 8'hdd, 8'hee, 8'hff, 8'h01,
41      8'h03, 8'h05, 8'h06
42  };
43
44  // Queue for verification
45  logic [7:0] fifo_queue [$];
46
47  initial begin
48      // Initialize
49      WCLK_top = 0;
50      RCLK_top = 0;
51      RST_top = 0; //reset active low
52      WRITE_ENABLE_TOP = 0;
53      READ_ENABLE_TOP = 0;
54      //WRITE_DATA_IN_top = 8'h00;
55
56      // Apply Reset
57      #(5*CLK_PERIOD_WR);
58      RST_top = 1;
59      #(5*CLK_PERIOD_WR);
60
61      // Write to FIFO
62      for (int i = 0; i < 19; i++) begin
63          @(posedge WCLK_top);
64          #1;
65          if (!FULL_top) begin
66              WRITE_ENABLE_TOP = 1;
67              WRITE_DATA_IN_top = predefined_data[i];
68              fifo_queue.push_back(predefined_data[i]); // Store for
                    verification
69              $display("Written: %h", predefined_data[i]);
70          end else begin
71              WRITE_ENABLE_TOP = 0;
72              WRITE_DATA_IN_top = 8'hx;
```

```
73          end
74      end
75      WRITE_ENABLE_TOP = 0;
76      WRITE_DATA_IN_top = 8'hx;
77
78      // Small delay before reading
79      #(5*CLK_PERIOD_WR);
80
81      // Read from FIFO
82      for (int i = 0; i < 20; i++) begin
83          @(posedge RCLK_top);
84          #1;
85          if (!EMPTY_top) begin
86              READ_ENABLE_TOP = 1;
87              //#7;
88              //@(posedge RCLK_top); // Wait for data to be valid
89              if (fifo_queue.size() > 0) begin
90                  automatic logic [7:0] expected_value = fifo_queue.pop_front
                        ();
91                  $display("Read␣Data:␣%h,␣Expected:␣%h,␣Match:␣%s",
92                           WRITE_DATA_OUT_top, expected_value,
93                           (WRITE_DATA_OUT_top == expected_value) ? "YES" : "
                             NO");
94              end
95          end else begin
96              READ_ENABLE_TOP = 0;
97          end
98      end
99      READ_ENABLE_TOP = 0;
100
101     // End Simulation
102     #(20*CLK_PERIOD_WR);
103     $stop;
104 end
105
106 endmodule
```

5.6.1 Output Delayed by Extra Clock due to Output Registers



- first clock edge, read enable go high
  - Addrb & rd_pointer at 0
- 2nd clock edge, addrb is incremented to 0x1
  - Data from address 0x00 is read out [ZZ]. This is when there are no output registers
  - When there are output registers, the data [ZZ] is put into registers
- 3rd clock edge: addr is incremented to 0x2
  - Data from address 0x00 [ZZ] is read out (assum. Lav. output registers)
    Data from address 0x01 [21] is put into the registers (assum. Lav. output registers)

* on this clock edge, Both rd_enable are seen to be high. This means data can be read. The address is seen to be 0x00. Thus data at 0x00 [0x17] is output [if no output registers].
  If output registers data is output on next clock edge.

Figure 17: Hand drawn waveforms displaying where doutb would first output data if no output registers were enabled v were it does when output registers are enabled