EXERCISE 2 (Asynchronous FIFO)

Daniel Duggan

March 12th 2025

Contents

1 Introduction

The purpose of this exercise is to design an asynchronous First In First Out (FIFO) 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`

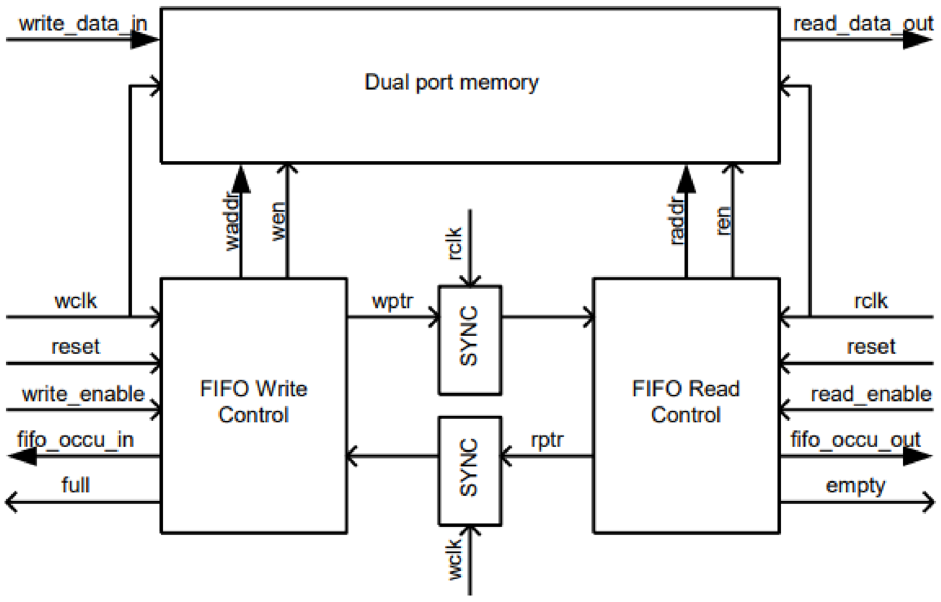


Figure 1: Block diagram of the asynchronous FIFO.

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 the probability of having metastable states when transferring *single* bits between clock domains, as opposed to

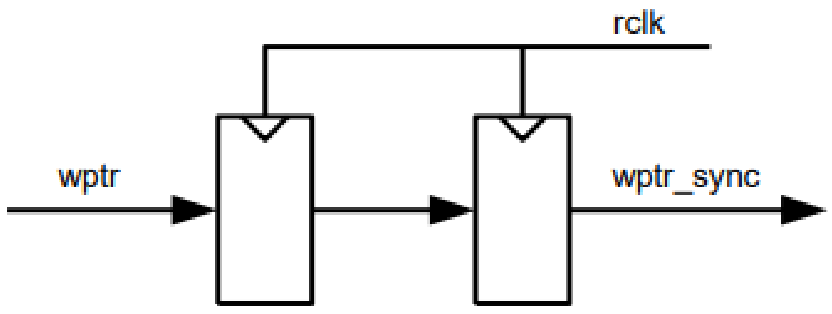


Figure 2: Circuitry that is insufficient to handle correct synchronisation in an asynchronous FIFO design.

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.

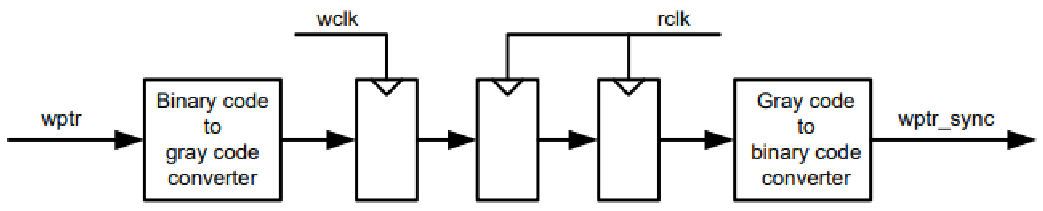

wclk
rclk
wptr
Binary code to gray code converter
Gray code to binary code converter
wptr_sync

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

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

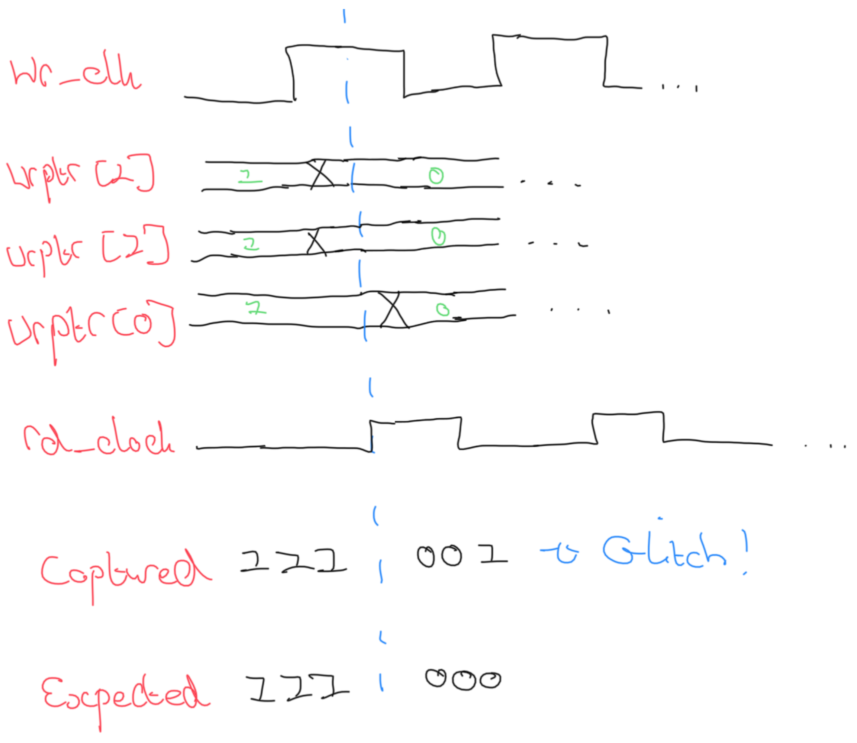


Figure 3: Example as to how circuitry in figure 2 can lead to incorrect output

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.

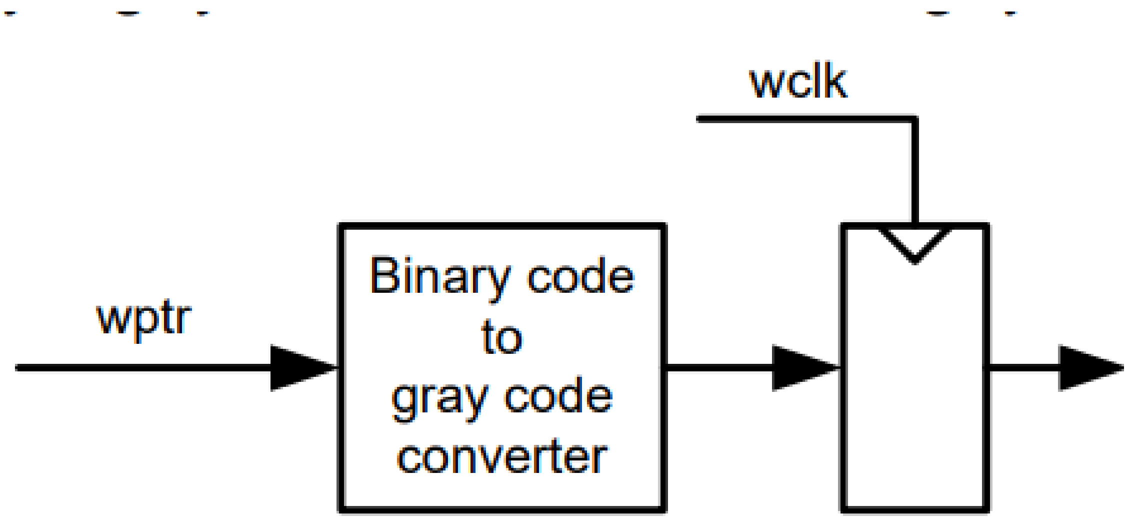
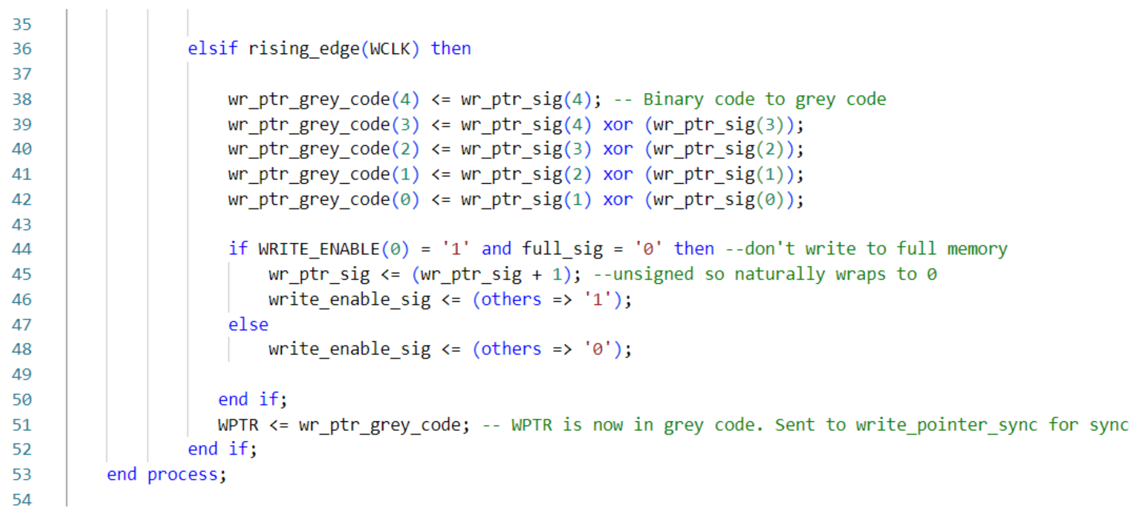


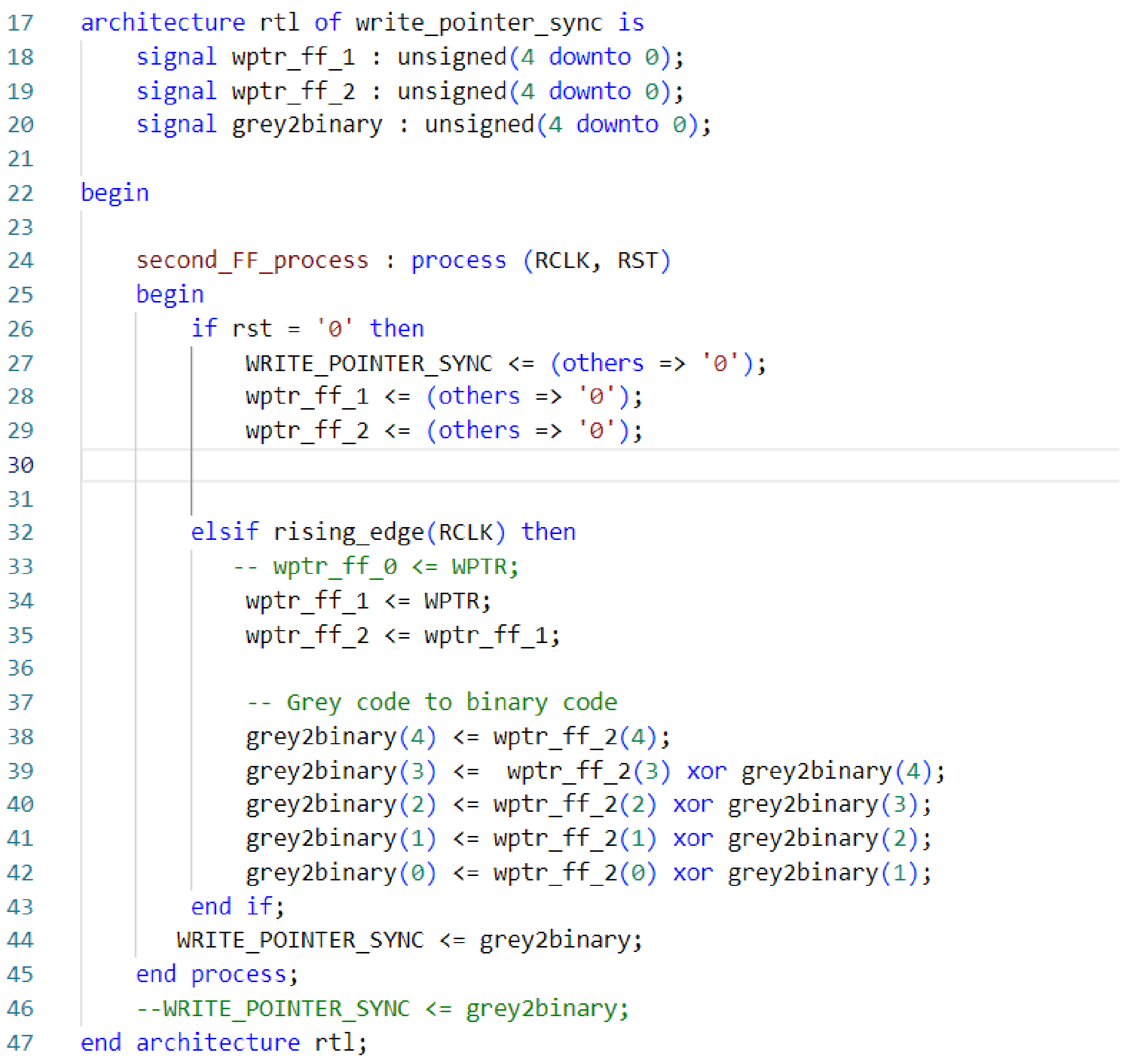
Figure 5: Partial synchronisation circuit carried out inside FIFO_write_control. The corresponding circuit for the read control is carried out inside FIFO_read_control.



```
            elsif rising_edge(WCLK) then

                wr_ptr_grey_code(4) <= wr_ptr_sig(4); -- Binary code to grey code
                wr_ptr_grey_code(3) <= wr_ptr_sig(4) xor (wr_ptr_sig(3));
                wr_ptr_grey_code(2) <= wr_ptr_sig(3) xor (wr_ptr_sig(2));
                wr_ptr_grey_code(1) <= wr_ptr_sig(2) xor (wr_ptr_sig(1));
                wr_ptr_grey_code(0) <= wr_ptr_sig(1) xor (wr_ptr_sig(0));

                if WRITE_ENABLE(0) = '1' and full_sig = '0' then --don't write to full memory
                    wr_ptr_sig <= (wr_ptr_sig + 1); --unsigned so naturally wraps to 0
                    write_enable_sig <= (others => '1');
                else
                    write_enable_sig <= (others => '0');

               end if;
               WPTR <= wr_ptr_grey_code; -- WPTR is now in grey code. Sent to write_pointer_sync for sync
            end if;
    end process;
```

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



```vhdl
architecture rtl of write_pointer_sync is
    signal wptr_ff_1 : unsigned(4 downto 0);
    signal wptr_ff_2 : unsigned(4 downto 0);
    signal grey2binary : unsigned(4 downto 0);

begin

    second_FF_process : process (RCLK, RST)
    begin
        if rst = '0' then
            WRITE_POINTER_SYNC <= (others => '0');
            wptr_ff_1 <= (others => '0');
            wptr_ff_2 <= (others => '0');


        elsif rising_edge(RCLK) then
           -- wptr_ff_0 <= WPTR;
            wptr_ff_1 <= WPTR;
            wptr_ff_2 <= wptr_ff_1;

            -- Grey code to binary code
            grey2binary(4) <= wptr_ff_2(4);
            grey2binary(3) <=  wptr_ff_2(3) xor grey2binary(4);
            grey2binary(2) <= wptr_ff_2(2) xor grey2binary(3);
            grey2binary(1) <= wptr_ff_2(1) xor grey2binary(2);
            grey2binary(0) <= wptr_ff_2(0) xor grey2binary(1);
        end if;
       WRITE_POINTER_SYNC <= grey2binary;
    end process;
    --WRITE_POINTER_SYNC <= grey2binary;
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.

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.

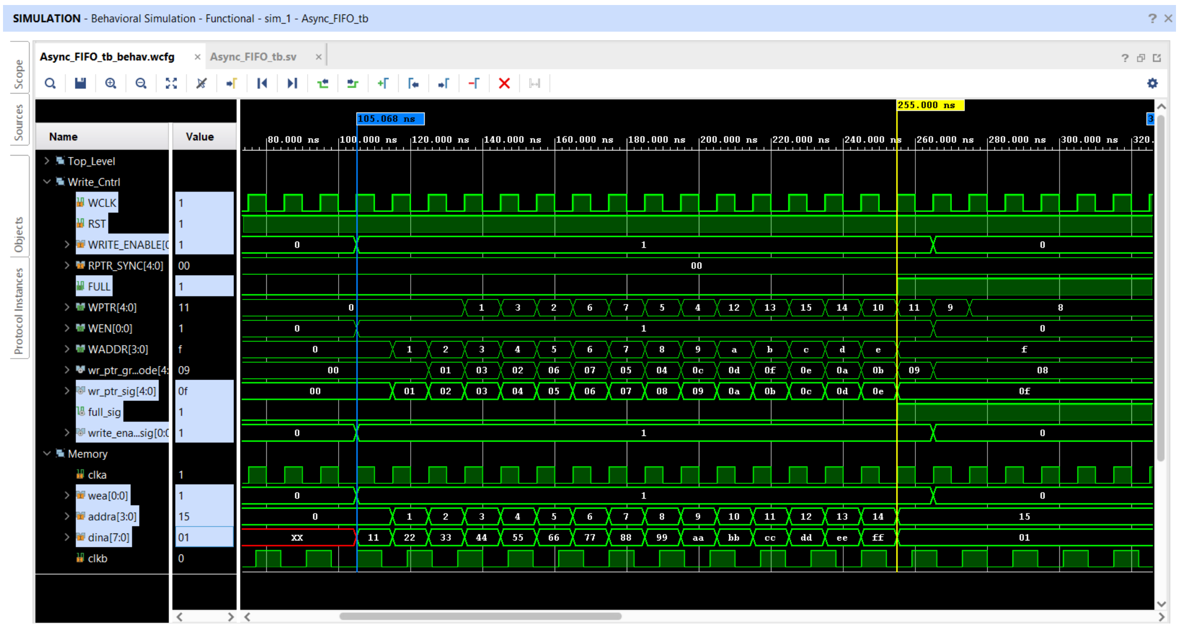


Figure 8: Simulation showing data is read in correctly, FULL flag operates correctly, and data is not read in once FULL flag is set.

The read side however does not work as expected (figure 9). 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 10. 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.

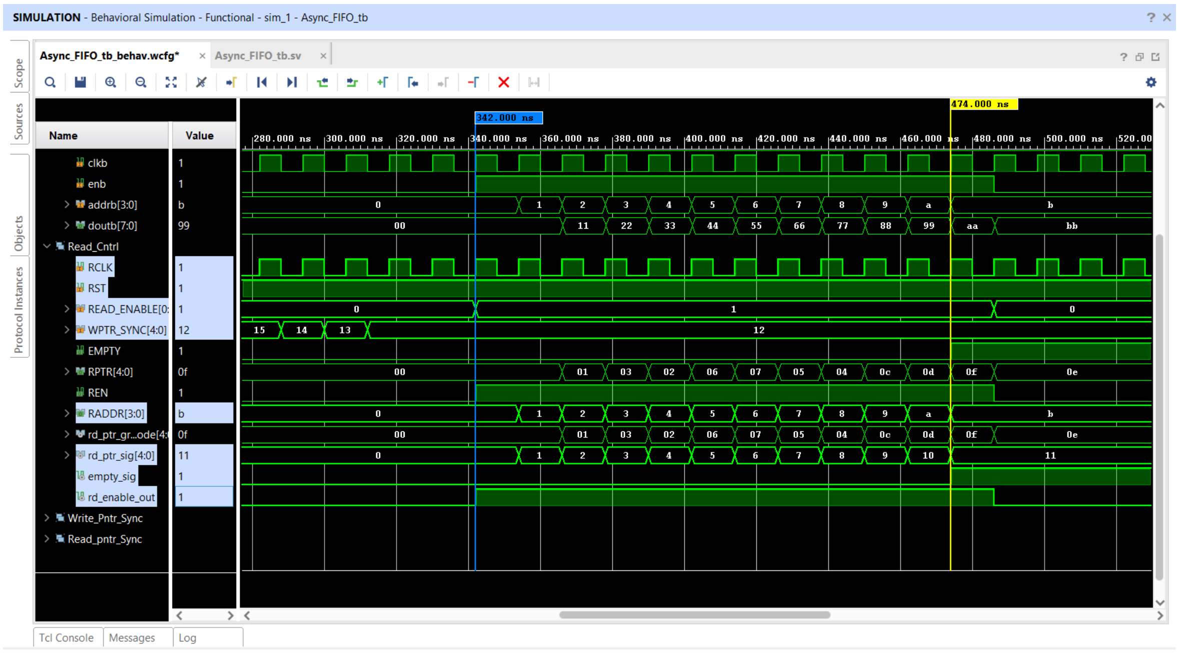


Figure 9: Simulation showing the read control side of the design does not operate as required.

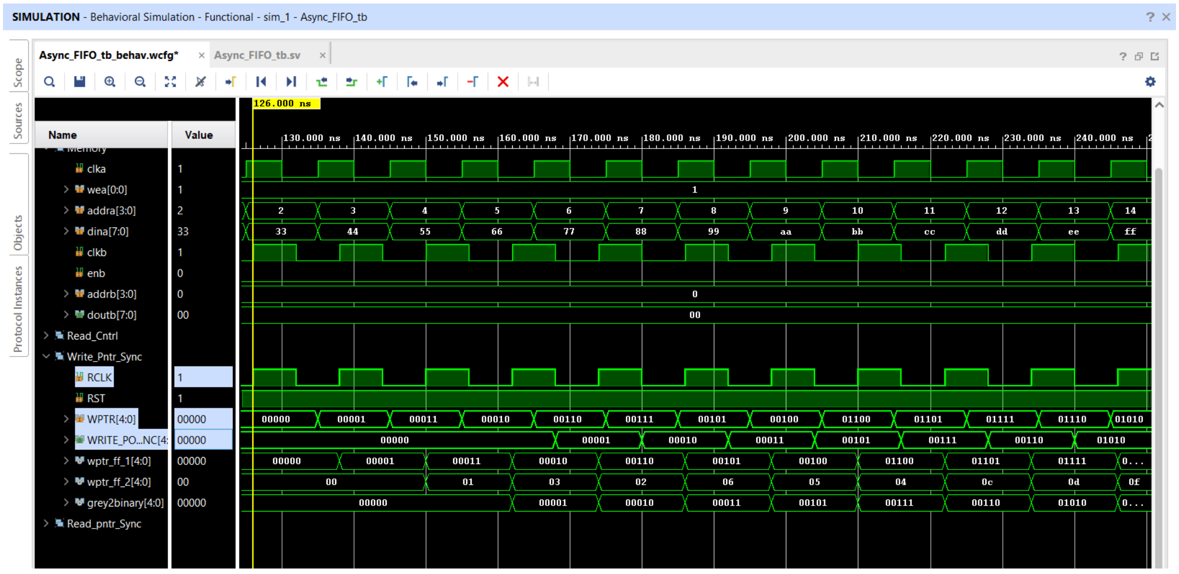


Figure 10: Simulation showing the syncronised write pointer circuitry does not operate as expected. WPTR is the input, which has undergone binary code to grey code conversion. WRITE_PO...NC = WRITE_POINTER_SYNC is the write pointer that has gone through two flip flops with the read side clock as the 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.

5 Appendix

5.1 Async_FIFO.vhd

```vhdl
-- vhdl-linter-disable type-resolved component. component
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-- LIBRARY blk_mem_gen_v8_4_7;
-- USE blk_mem_gen_v8_4_7.blk_mem_gen_v8_4_7;

entity Async_FIFO is
    port (
        RST_top : in std_logic;
        WCLK_top : in std_logic;
        RCLK_top : in std_logic;
        WRITE_ENABLE_TOP : in std_logic_vector(0 downto 0);
        READ_ENABLE_TOP : in std_logic_vector(0 downto 0);
        FULL_top : out std_logic; --output if memory is full
        EMPTY_top : out std_logic; -- output if memory is empty
        WRITE_DATA_IN_top : in std_logic_vector(7 downto 0);
        WRITE_DATA_OUT_top : out std_logic_vector(7 downto 0)
    );
end entity;

architecture rtl of Async_FIFO is

    -- Declare the component
    COMPONENT blk_mem_gen_0
        PORT (
            clka   : IN std_logic;
            wea    : IN std_logic_vector(0 DOWNTO 0);
            addra  : IN std_logic_vector(3 DOWNTO 0);
            dina   : IN std_logic_vector(7 DOWNTO 0);
            clkb   : IN std_logic;
            enb    : IN std_logic;
            addrb  : IN std_logic_vector(3 DOWNTO 0);
            doutb  : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    -- FIFO_WRITE_CONTROL SIGNALS.
    signal wen_sig : std_logic_vector(0 downto 0); -- write enable from
        FIFO_WRITE_Control
     --write address from FIFO_WRITE_Control
    signal wr_pointer_sig : unsigned(4 downto 0); --write address from FIFO_WRITE_Control are bits (3 downto 0)
    signal wr_addr_a : unsigned(3 downto 0);
    signal wr_pointer_sync : unsigned(4 downto 0);

    --FIFO_READ_CONTROL SIGNALS.
    signal ren_sig : std_logic; -- write enable from FIFO_READ_CONTROL
    signal rd_pointer_sig : unsigned(4 downto 0);
    signal rd_addr_b : unsigned(3 downto 0);
    signal rd_pointer_sync : unsigned(4 downto 0);

begin

    Fifo_write_control_inst : entity work.FIFO_write_control
    port map(
        WCLK => WCLK_top,
        RST => RST_top,
        WRITE_ENABLE => WRITE_ENABLE_TOP,
        RPTR_SYNC => rd_pointer_sync,  --Input from Read_pointer_sync. Synchronised read pointer.
        FULL => FULL_top,
        WPTR => wr_pointer_sig, --Write pointer goes to the sync
        WEN => wen_sig, -- goes to block_mem via sig wen_sig in async_FIFO
        WADDR => wr_addr_a -- Output to addra of Memory (Write Address).

    );

    Dual_port_memory_inst : blk_mem_gen_0 -- vhdl-linter-disable-line not-declared

    port map(
        clka => WCLK_top,
        wea => wen_sig, -- write enable from FIFO_WRITE_Control
        addra => std_logic_vector(wr_addr_a), -- write address from FIFO_WRITE_Control
        dina => WRITE_DATA_IN_top,
        clkb => RCLK_top,
        enb => ren_sig, -- write enable from FIFO_READ_CONTROL
        addrb => std_logic_vector(rd_addr_b), --read address from FIFO_READ_CONTROL
        doutb => WRITE_DATA_OUT_top
    );

    Fifo_read_control_inst : entity work.FIFO_read_control -- vhdl-linter-disable-line not-declared
    port map(
        RCLK => RCLK_top,
        RST => RST_top,
        READ_ENABLE => READ_ENABLE_TOP, -- read enable from TOP
        REN => ren_sig, -- write enable to BLOCK_MEM
        WPTR_SYNC => wr_pointer_sync, -- Input sig from WRITE_POINTER_SYNC. To determine occupancy and empty/full.
        EMPTY => EMPTY_top, -- Output sig. Goes to TOP
        RPTR => rd_pointer_sig, -- output sig to Read_pointer_sync entity. Signal to be synchronised with wr_pointer_sig
        RADDR => rd_addr_b  -- Output sig to Dual port memory. Read address.

    );

    Write_pointer_sync_inst : entity work.write_pointer_sync
    port map(
            RCLK => RCLK_top,
            --WCLK => WCLK_top,
            RST => RST_top,
            WPTR => wr_pointer_sig,
            WRITE_POINTER_SYNC => wr_pointer_sync
    );

    Read_pointer_sync_inst : entity work.read_pointer_sync
    port map(
            --RCLK => RCLK_top,
            WCLK => WCLK_top,
            RST => RST_top,
            RPTR => rd_pointer_sig, -- input from FIFO_read_control.
            READ_POINTER_SYNC => rd_pointer_sync -- Output to FIFO_write_control. Synchronised read_pointer.
    );

end architecture rtl;
```

5.2 FIFO_write_control.vhd

```vhdl
-- vhdl-linter-disable type-resolved
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity FIFO_WRITE_CONTROL is
    port (
        WCLK : in std_logic;
        RST : in std_logic;
        WRITE_ENABLE : in std_logic_vector(0 downto 0);
        RPTR_SYNC : in unsigned(4 downto 0); --rd pointer that comes from the sync
        FULL : out std_logic;
        WPTR : out unsigned(4 downto 0); --Write pointer goes to the sync i.e. grey-coded
        WEN : out std_logic_vector(0 downto 0); -- goes to block_mem via sig wen_sig in async_FIFO
        WADDR : out unsigned(3 downto 0) -- write address. Goes to block_mem via signal waddr_sig in async_FIFO
    );
end entity;

architecture rtl of FIFO_WRITE_CONTROL is

    signal wr_ptr_grey_code : unsigned(4 downto 0);
    signal wr_ptr_sig : unsigned(4 downto 0);
    signal full_sig : std_logic;
    signal write_enable_sig : std_logic_vector(0 downto 0);
    signal wr_ptr_sig_delay : unsigned(4 downto 0);

begin

    write_control_process : process (WCLK,RST) --asyn reset
    begin
            if RST = '0' then --reset active low
                wr_ptr_grey_code <= (others => '0');
                write_enable_sig <= (others => '0');
                wr_ptr_sig <= (others => '0');
                wr_ptr_sig_delay <= (others => '0');



            elsif rising_edge(WCLK) then

                wr_ptr_grey_code(4) <= wr_ptr_sig(4); -- Binary code to grey code
                wr_ptr_grey_code(3) <= wr_ptr_sig(4) xor (wr_ptr_sig(3));
                wr_ptr_grey_code(2) <= wr_ptr_sig(3) xor (wr_ptr_sig(2));
                wr_ptr_grey_code(1) <= wr_ptr_sig(2) xor (wr_ptr_sig(1));
                wr_ptr_grey_code(0) <= wr_ptr_sig(1) xor (wr_ptr_sig(0));

```

```vhdl
                if WRITE_ENABLE(0) = '1' and full_sig = '0' then --don't write to full memory
                    wr_ptr_sig_delay <= (wr_ptr_sig_delay + 1);
                    wr_ptr_sig <= wr_ptr_sig_delay; --unsigned so naturally wraps to 0
                    write_enable_sig <= (others => '1');
                else
                    write_enable_sig <= (others => '0');

               end if;
               WPTR <= wr_ptr_grey_code; -- WPTR is now in grey code. Sent to write_pointer_sync for sync
            end if;
    end process;

    full_sig <= '1' when (wr_ptr_sig - RPTR_SYNC = 15) else '0';


    --WPTR <= wr_ptr_grey_code; -- WPTR is now in grey code. Sent to write_pointer_sync for sync
    FULL <= full_sig;
    WADDR <= (wr_ptr_sig(3 downto 0)); -- sent to the Dual-port memory
    WEN <= write_enable_sig; --sent to the Dual-port memory
end architecture rtl;
```

5.3 FIFO_read_control.vhd

```vhdl
-- vhdl-linter-disable type-resolved
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity FIFO_READ_CONTROL is
    port (
        RCLK : in std_logic;
        RST : in std_logic;
        READ_ENABLE : in std_logic_vector(0 downto 0);
        WPTR_SYNC : in unsigned(4 downto 0); --wp pointer that comes from the sync
        EMPTY : out std_logic;
        RPTR : out unsigned(4 downto 0); --Write pointer goes to the sync i.e. grey-coded
        REN : out std_logic; -- goes to block_mem via sig wen_sig in async_FIFO
        RADDR : out unsigned(3 downto 0) -- write address. Goes to block_mem via signal waddr_sig in async_FIFO
    );
end entity;

architecture rtl of FIFO_READ_CONTROL is

    signal rd_ptr_grey_code : unsigned(4 downto 0);
    signal rd_ptr_sig : unsigned(4 downto 0);
    signal rd_ptr_sig_delay: unsigned(4 downto 0);
    signal rd_ptr_sig2 : unsigned(4 downto 0);
    signal empty_sig : std_logic;
    signal rd_enable_out : std_logic;

begin

    FIFO_read_control_process : process (RCLK, RST) --asyn reset
    begin
        if RST = '0' then --reset active low
            --RPTR <= (others => '0');
            rd_ptr_grey_code <= (others => '0');-- this gives multiple load error
            rd_enable_out <= '0';
            rd_ptr_sig <= (others => '0'); --this gives multiple load error
            rd_ptr_sig_delay <= (others => '0');
            rd_ptr_sig2 <= (others => '0');


        elsif rising_edge(RCLK) then

            rd_ptr_grey_code(4) <= rd_ptr_sig(4); -- Binary code to grey code
            rd_ptr_grey_code(3) <= rd_ptr_sig(4) xor (rd_ptr_sig(3));
            rd_ptr_grey_code(2) <= rd_ptr_sig(3) xor (rd_ptr_sig(2));
            rd_ptr_grey_code(1) <= rd_ptr_sig(2) xor (rd_ptr_sig(1));
            rd_ptr_grey_code(0) <= rd_ptr_sig(1) xor (rd_ptr_sig(0));

            if READ_ENABLE(0) = '1' and empty_sig = '0' then --don't read from empty memory
                rd_ptr_sig_delay <= (rd_ptr_sig_delay + 1); --unsigned so naturally wraps to 0
                rd_ptr_sig <= rd_ptr_sig_delay;
                --rd_ptr_sig <= rd_ptr_sig2;
                rd_enable_out <= '1';
            else
```

```
                rd_enable_out <= '0';
            end if;

        end if;

    end process;

    empty_sig <= '1' when (rd_ptr_sig_delay = WPTR_SYNC) else '0'; --Check
       if empty or not. Uses synced wr_ptr which has 5 cc delay
    RADDR <= (rd_ptr_sig(3 downto 0));
    RPTR <= rd_ptr_grey_code; -- RPTR is now in grey code.
    EMPTY <= empty_sig;
    REN <= rd_enable_out;
end architecture rtl;
```

5.4 write_pointer_sync.vhd

```
-- vhdl-linter-disable type-resolved
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity write_pointer_sync is
    port (
        RCLK : in std_logic;
        --WCLK : in std_logic;
        RST : in std_logic;
        WPTR : in unsigned(4 downto 0);
        WRITE_POINTER_SYNC : out unsigned(4 downto 0)

    );
end entity;

architecture rtl of write_pointer_sync is
    signal wptr_ff_1 : unsigned(4 downto 0);
    signal wptr_ff_2 : unsigned(4 downto 0);
    signal grey2binary : unsigned(4 downto 0);

begin

    second_FF_process : process (RCLK, RST)
    begin
        if rst = '0' then
            WRITE_POINTER_SYNC <= (others => '0');
            wptr_ff_1 <= (others => '0');
            wptr_ff_2 <= (others => '0');


        elsif rising_edge(RCLK) then
           -- wptr_ff_0 <= WPTR;
            wptr_ff_1 <= WPTR;
            wptr_ff_2 <= wptr_ff_1;

            -- Grey code to binary code
            grey2binary(4) <= wptr_ff_2(4);
            grey2binary(3) <=  wptr_ff_2(3) xor grey2binary(4);
            grey2binary(2) <= wptr_ff_2(2) xor grey2binary(3);
            grey2binary(1) <= wptr_ff_2(1) xor grey2binary(2);
            grey2binary(0) <= wptr_ff_2(0) xor grey2binary(1);
        end if;
       WRITE_POINTER_SYNC <= grey2binary;
    end process;
    --WRITE_POINTER_SYNC <= grey2binary;
end architecture rtl;
```

5.5 read_pointer_sync.vhd

```
-- vhdl-linter-disable type-resolved
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity read_pointer_sync is
    port (
        --RCLK : in std_logic;
        WCLK : in std_logic;
        RST : in std_logic;
        RPTR : in unsigned(4 downto 0);
        READ_POINTER_SYNC : out unsigned(4 downto 0)
    );
end entity;

architecture rtl of read_pointer_sync is
    signal rptr_ff_1 : unsigned(4 downto 0);
    signal rptr_ff_2 : unsigned(4 downto 0);
    signal grey2binary : unsigned(4 downto 0);

begin
```

```
    second_FF_process : process (WCLK,RST)
    begin
        if RST = '0' then
            READ_POINTER_SYNC <= (others => '0');
            rptr_ff_1 <= (others => '0');
            rptr_ff_2 <= (others => '0');

            elsif rising_edge(WCLK) then

            rptr_ff_1 <= RPTR;
            rptr_ff_2 <= rptr_ff_1;

            -- MSB of binary is the same as MSB of gray code
            grey2binary(4) <= rptr_ff_2(4);
            -- Other bits of binary are the XOR of corresponding gray code
                and previous binary bit
            grey2binary(3) <= rptr_ff_2(3) xor grey2binary(4);
            grey2binary(2) <= rptr_ff_2(2) xor grey2binary(3);
            grey2binary(1) <= rptr_ff_2(1) xor grey2binary(2);
            grey2binary(0) <= rptr_ff_2(0) xor grey2binary(1);
        end if;
        READ_POINTER_SYNC <= grey2binary; -- Needs to be inside process so
            it happens on rising_edge of clock!
    end process;
   --READ_POINTER_SYNC <= grey2binary;
end architecture rtl;
```

5.6 Async_FIFO_tb.sv

```
// Testbench for Async_FIFO
module Async_FIFO_tb;

// Signals
logic RST_top;
logic WCLK_top;
logic RCLK_top;
logic [0:0] WRITE_ENABLE_TOP;
logic [0:0] READ_ENABLE_TOP;
logic FULL_top;
logic EMPTY_top;
logic [7:0] WRITE_DATA_IN_top;
logic [7:0] WRITE_DATA_OUT_top;

// Clock periods
```

```
parameter CLK_PERIOD_WR = 10; // Write clock period (e.g., 100MHz)
parameter CLK_PERIOD_RD = 12; // Read clock period (e.g., 66.6MHz)

// FIFO instance
Async_FIFO uut (
    .RST_top(RST_top),
    .WCLK_top(WCLK_top),
    .RCLK_top(RCLK_top),
    .WRITE_ENABLE_TOP(WRITE_ENABLE_TOP),
    .READ_ENABLE_TOP(READ_ENABLE_TOP),
    .FULL_top(FULL_top),
    .EMPTY_top(EMPTY_top),
    .WRITE_DATA_IN_top(WRITE_DATA_IN_top),
    .WRITE_DATA_OUT_top(WRITE_DATA_OUT_top)
);

// Clock Generation
always #(CLK_PERIOD_WR/2) WCLK_top = ~WCLK_top;
always #(CLK_PERIOD_RD/2) RCLK_top = ~RCLK_top;

// Predefined data array
logic [7:0] predefined_data [0:18] = '{
    8'h11, 8'h22, 8'h33, 8'h44, 8'h55, 8'h66, 8'h77, 8'h88,
    8'h99, 8'haa, 8'hbb, 8'hcc, 8'hdd, 8'hee, 8'hff, 8'h01,
    8'h03, 8'h05, 8'h06
};

// Queue for verification
logic [7:0] fifo_queue [$];

initial begin
    // Initialize
    WCLK_top = 0;
    RCLK_top = 0;
    RST_top = 0; //reset active low
    WRITE_ENABLE_TOP = 0;
    READ_ENABLE_TOP = 0;
    //WRITE_DATA_IN_top = 8'h00;

    // Apply Reset
    #(5*CLK_PERIOD_WR);
    RST_top = 1;
    #(5*CLK_PERIOD_WR);

    // Write to FIFO
    for (int i = 0; i < 19; i++) begin
        @(posedge WCLK_top);
        if (!FULL_top) begin
            WRITE_ENABLE_TOP = 1;
            //#5;
            WRITE_DATA_IN_top = predefined_data[i];
            fifo_queue.push_back(predefined_data[i]); // Store for verification
            $display("Written:␣%h", predefined_data[i]);
        end else begin
            WRITE_ENABLE_TOP = 0;
        end
    end
    WRITE_ENABLE_TOP = 0;

    // Small delay before reading
    #(5*CLK_PERIOD_WR);

    // Read from FIFO
    for (int i = 0; i < 20; i++) begin
        @(posedge RCLK_top);
        if (!EMPTY_top) begin
            READ_ENABLE_TOP = 1;
            //#7;
            //@(posedge RCLK_top); // Wait for data to be valid
            if (fifo_queue.size() > 0) begin
                automatic logic [7:0] expected_value = fifo_queue.pop_front();
                $display("Read␣Data:␣%h,␣Expected:␣%h,␣Match:␣%s",
                         WRITE_DATA_OUT_top, expected_value,
                         (WRITE_DATA_OUT_top == expected_value) ? "YES" : "NO");
            end
        end else begin
            READ_ENABLE_TOP = 0;
        end
    end
    READ_ENABLE_TOP = 0;

    // End Simulation
    #(20*CLK_PERIOD_WR);
    $stop;
end

endmodule
```