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.

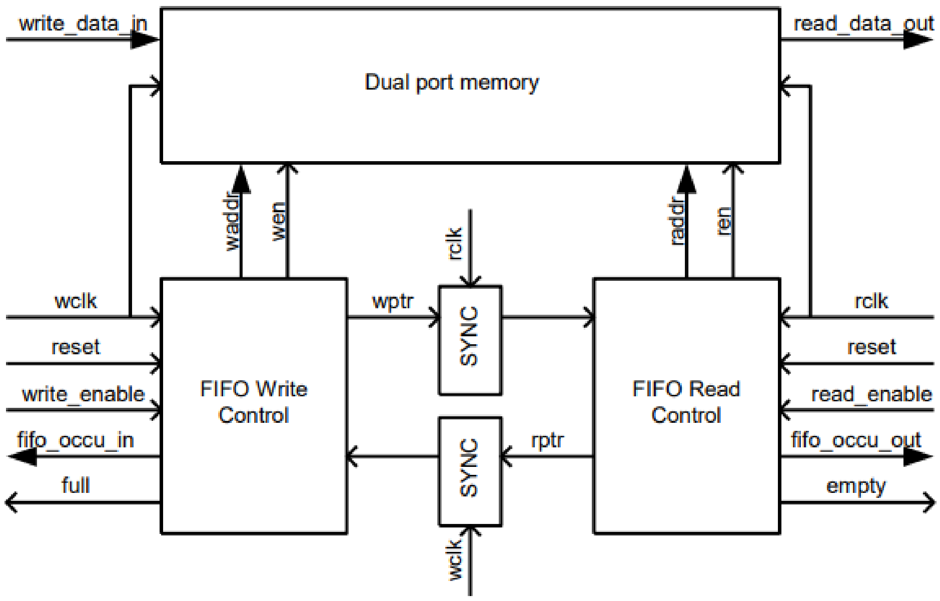

Figure 1: Block diagram of the asynchronous FIFO.

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 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.

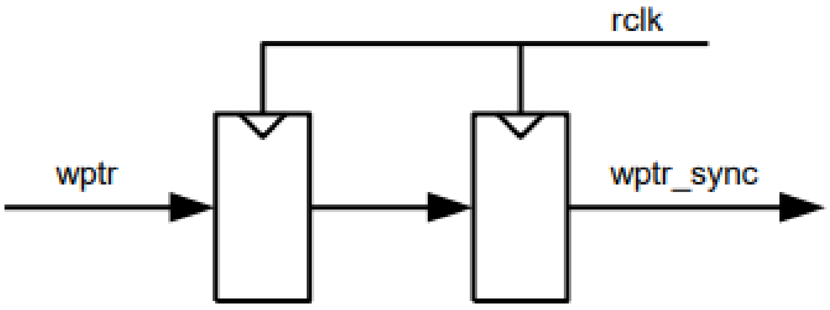

Figure 2: Circuitry that is insufficient to handle correct synchronisation in an asynchronous FIFO design.

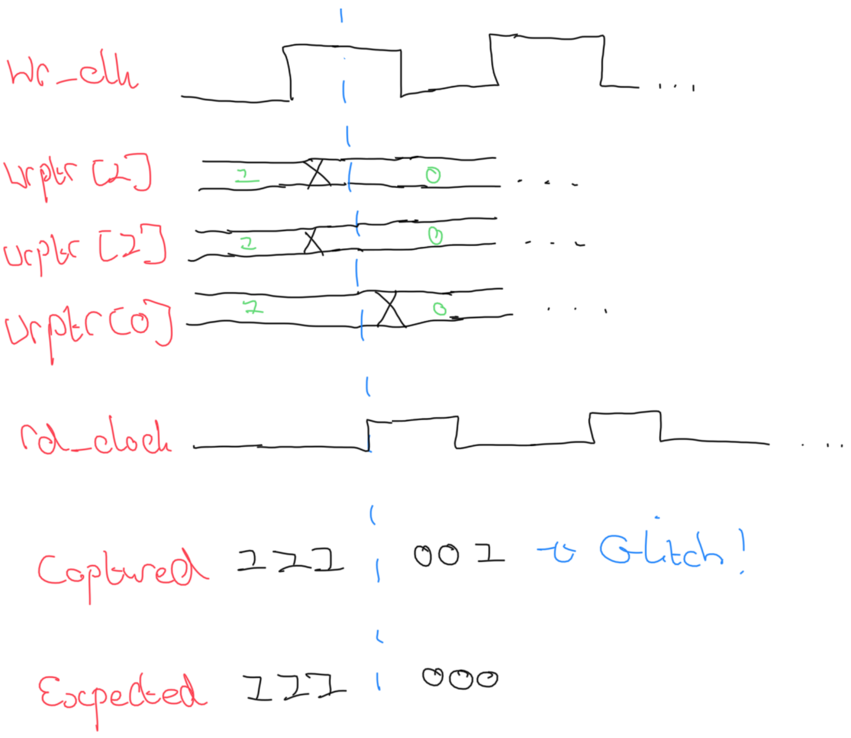

Figure 3: Example as to how circuitry in figure 2 can lead to incorrect output

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.

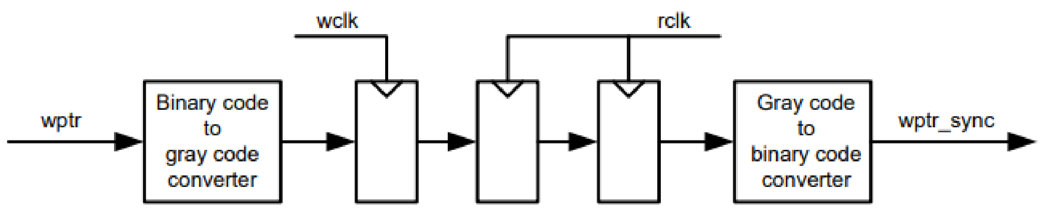
wclk
rclk
wptr
Binary code to gray code converter
Gray code to binary code converter
wptr_sync

Figure 4: Block diagram of the asynchronous FIFO.

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

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

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

$$G_5 = B_5$$

$$G_4 = B_5 \oplus B_4$$

$$G_3 = B_4 \oplus B_3$$

$$G_2 = B_3 \oplus B_2$$

$$G_1 = B_2 \oplus B_1$$

$$G_0 = B_1 \oplus B_0$$

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:

$$B_5 = G_5$$

$$B_4 = G_4 \oplus B_5$$

$$B_3 = G_3 \oplus B_4$$

$$B_2 = G_2 \oplus B_3$$

$$B_1 = G_1 \oplus B_2$$

$$B_0 = G_0 \oplus B_1$$

References

4 Appendix

4.1 Serial Implementation - System Top (Sys_top.vhd)

```
-- Author Daniel Duggan
-- vhdl-linter-disable type-resolved
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

USE IEEE.NUMERIC_STD.ALL;

LIBRARY UNISIM;
USE UNISIM.VCOMPONENTS.ALL; -- Required for IBUFDS -- vhdl-linter-disable-
    line not-declared

ENTITY sys_top IS
    PORT(
        SYS_CLOCK_P : IN STD_LOGIC;  -- LVDS clock positive
        SYS_CLOCK_N : IN STD_LOGIC;  -- LVDS clock negative
        RST : IN STD_LOGIC;
        START_OF_FRAME_top : IN STD_LOGIC;
        END_OF_FRAME_top : IN STD_LOGIC;
        DATA_IN_top : IN STD_LOGIC;
        FCS_ERROR_top : OUT STD_LOGIC -- vhdl-linter-disable-line type-
            resolved
    );
END sys_top;

ARCHITECTURE rtl OF sys_top IS
    -- Single-ended clock signal after differential conversion
    SIGNAL sysclk_single : STD_LOGIC; -- vhdl-linter-disable-line type-
        resolved

BEGIN
    -- Convert the differential clock to single-ended using IBUFDS
    clk_buffer: IBUFDS -- don't use work as this is not a user defined
        entity -- vhdl-linter-disable-line not-declared
    PORT MAP (
        I  => SYS_CLOCK_P,  -- Positive clock input
        IB => SYS_CLOCK_N,  -- Negative clock input
        O  => sysclk_single -- Output single-ended clock
    );

    -- Instance of fcs_check_serial using the single-ended clock
    fcs_serial_FSM_inst: entity work.fcs_check_serial
    PORT MAP (
        CLK => sysclk_single,  -- Send the converted clock
        RST => RST,
        START_OF_FRAME => START_OF_FRAME_top,
        END_OF_FRAME => END_OF_FRAME_top,
        DATA_IN => DATA_IN_top,
        FCS_ERROR => FCS_ERROR_top
    );

END rtl;
```