University of Windsor

Electrical and Computer Engineering

ELEC-4430: Embedded System Design

Data Buffering System for FPGA, Progress Report 2

Student ID: **103186504**
Last Name: **ILIEVSKI**
First Name: **BRAJAN**
ilievsk1@uwindsor.ca

Student ID: **104593653**
Last Name: **HUYNH**
First Name: **KHANH**
huynh116@uwindsor.ca

Student ID: **104754540**
Last Name: **KHAN**
First Name: **MAHWISH**
khan1cz@uwindsor.ca

March 1, 2021

Submitted To

Instructor: Dr M Khalid

GA: Rico, Md Maksud-Ul-Kabir

Code Progression:

In the previous progress report, the group successfully demonstrated the limitations of extending a synchronous design to work in an asynchronous two-clock domain. These limitations included the challenges with driving control signals with two different clocks and the need for synchronizers and grey code counters to overcome the challenges with comparing pointers existing in two different frequency domains.

In this report, we begin with a description and walkthrough of our asynchronous FIFO design and its associated code. Unlike our previous iteration, the async FIFO design separates the read and write pointer control paths into two separate processes driven by *oclk* and *iclk,* respectively. For both read side and write side processes, there exists a respective grey-to-binary converter that remains outside the processes to enable comparison of the read or write pointer (in its native clock domain) to a synchronized read or write pointer originating from the other clock domain. To more clearly illustrate the asynchronous design, consider the code snippet for the write side process (W is initialized to 3):

```
write_side : process(iclk)
begin
    if rising_edge(iclk) then
        if resetW = '1' then
            writePtr <= (others => '0');
            writePtrGray <= (others => '0');
            syncReadPtrBin <= (others => '0');
            readPtrGraySync0 <= (others => '0');
            readPtrGraySync1 <= (others => '0');
        else
            if fill = '1' and not full = '1' then
                writePtr <= writePtr + '1';
            end if;
            --Binary-to-Grey Conversion for the write pointer
            writePtrGray <= writePtr xor ('0' & writePtr(W-1 downto 1));
            --Synchroniser
            readPtrGraySync0 <= readPtrGray;
            readPtrGraySync1 <= readPtrGraySync0;
            --Read Pointer Register
            syncReadPtrBin <= readPtrBin;
        end if;
    end if;
end process;

--Grey-to-Binary Conversion for the read pointer
readPtrBin(W-1) <= readPtrGraySync1(W-1);
for i in W-2 downto 0 generate
    readPtrBin(i) <= readPtrBin(i+1) xor readPtrGraySync1(i);
end generate;

full <= '1' when writePtr + '1' = syncReadPtrBin else '0';
fifo_full <= full;
```

When *reset* is not asserted, the write pointer is incremented when the *fill* input is asserted and the FIFO is not full. This is sequentially followed by the conversion of the current binary write pointer into grey code by XORing the write pointer with a left-shifted version of itself. The Grey-code write pointer (*writePtrGray)* is not used in this write-side process but will instead be sent and synchronized to the read-side process for comparison against the native read pointer (more on this later). Similar to how the native *writePtrGray* is sent to the read side, this write-side process will receive a Gray-code read pointer (*readPtrGray*) from the read side, and this read pointer will be sent to the two-stage synchronizer. Outside of the read-side process, the synchronized Gray-code read pointer (*readPtrGraySync1)* is converted back to binary using a *for* loop to incrementally XOR the grey-code pointer with the binary pointer starting from MSB to LSB.

Having successfully synchronized and converted the read pointer into the write-clock domain, we next carry out a comparison between the native write pointer and synchronized read pointer in binary form. We bring special attention to the second last line of this code snippet given that this is where the *full* flag is asserted. When the FIFO has been written to eight times and the write pointer is incremented to "111" in the process, we check whether a roll over condition *will* occur in the next increment by adding '1' to the read pointer and comparing it against the synchronized read pointer. In this case, "111" + '1' returns "000" which will be equal to the read pointer. This will assert the *full* flag and will disable any more write pointer incrementation in the above process, and thus the write pointer will remain at "111".

Next, let us analyze the condition required to assert the *empty* flag from the read side. In the code snippet below, it is observed that the read pointer does not have '1' added to it in the comparison. Instead, we directly compare the value of the read pointer to that of the synchronized write pointer. After writing and reading from the FIFO a total of eight times, the read pointer will be incremented to "111" during the last read operation within the process. When this occurs, the native read pointer now equals the synchronized binary write pointer (originally transmitted as *writePtrGray* from the write side) and the *empty* flag is asserted.

```
--Subsequent to read-side process
empty <= '1' when readPtr = syncWritePtrBin else '0';
fifo_empty <= empty;
```

From the description given above, it is observed that the *full* and *empty* flags are asserted under two different equal pointer conditions without the need to track a *rollover* bit under two different clocks. This is achieved by raising the *full* flag when a rollover of the native *write* pointer <u>will</u> occur in the next increment process but the write pointer is maintained at "111". In this way, we are able to compare the read and write pointers at the condition "000" = "000" without incrementing the write pointer. Leveraging the situation that the write pointer remains at "111", we can then evaluate when the read pointer catches up to the write pointer at the condition "111" = "111". As such, we successfully evaluate two different equal conditions without an asynchronously driven rollover bit.

The asynchronous code implemented is present below in the appendix, and the logic exists within the *asynchron_FIFO* entity that begins on line 99.

Upon completing the design expansion from synchronous FIFO to asynchronous FIFO, the next step in the data buffering system is to build the top entity which is comprised of 3 FIFO's along with supplementary logic that connects everything together and drives the control signals. As is presented in the appendix below, our current code demonstrates the *data_buff* entity declaration along with the ports specified in the spec, as well as the *asynchron_FIFO* entity which is instantiated thrice in *data_buff.* Currently, some connections are overlapping, as the additional logic has not been inserted by way of parallel processes.

It can be noted that the way to define the physical connections within the entity is by utilizing a port map which as its name indicates, directly maps ports from the component to ports of the top-most entity. Figure 1 below displays how this is written in VHDL.

```
,...
FIFO1: asynchronFIFO
      generic map (8, 8, 8)
      port map (resetR => reset, resetW => reset, oclk => OCLK, iclk => ICLK,
                enEmpty => empty_fifo, enFill => fill_fifo,
                fifo_empty => isEmpty1, fifo_full => isFull1,
                FIFO_OUT => DOUT, FIFO_IN => DIN); -- note, DOUT will need to be mapped to
                                                   -- auxiliary signal as we cannot map all FIFO outputs to DOUT
```

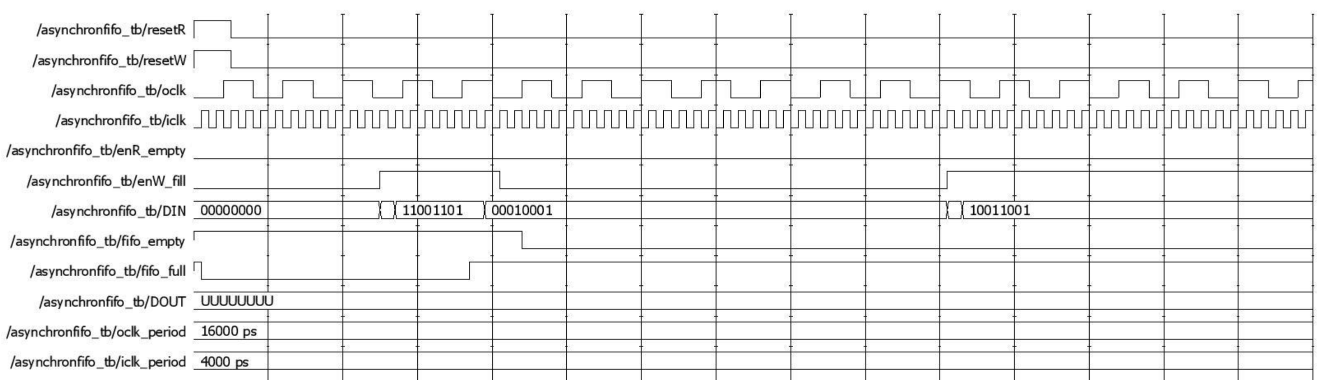
Figure 1: FIFO1 instantiation and port mapping

This is repeated for FIFO2 and FIFO3, although certain ports will be mapped differently. Moving forward, the parallel processes that drive the control logic, such as indicating an empty of full FIFO, determining which FIFO to read from provided the order requirement and etc.

<u>Testbench Development:</u>

Although a testbench will be provided for the data buffering system, to actively monitor the performance of the data buffering system, it was crucial to put together a simple testbench for basic testing and verification. This testbench was developed upon completing the asynchronous FIFO code and is currently limited to testing just one FIFO as opposed to the entire system. The stimuli generated includes the input and output clocks, the reset signals, the enabling fill/empty and etc. The main focus of the waveform generated below is to analyze the clocks and the writing ability.

The output of the FIFO data buffer and the testbench is presented as below:



From the waveform output, it is recognizable that the data buffer in running on two different clocks simultaneously, (*oclk* and *iclk).* The input clock signal oversees the write or fill operation is 4 times faster than output clock signal since *oclk* and *iclk* are set to be 16000ps and 4000ps, respectively. With more rising clock edge, the write operation is updated more frequent than read operation. Consequently, the input transferred into the data buffer system can be more valid and plausible (avoid the transient state input). As a result, the data buffer system is expected to conduct more write operation than read as indicated in the *oclk* and *iclk* waveform.

The output of FIFO data buffer above demonstrates the behavior of the asynchronous FIFO in write operation. At the beginning, two reset signals (resetR and resetW) are set to active-high to normalize the system. To write an input into FIFO, the enable write (enW_fill) must be set to high. In the testbench, write enables is set to be high every 30 rising-edge of input clock. As soon as the write enable is active-high, the data system begins to load memory FIFO with DIN value every rising edge clock cycle. The write enable is set to 8 consecutive rising edge input clocks to fully load 8 words into data FIFO. After writing 8 words in FIFO buffer, the fifo_empty flag is set to active-low since the FIFO is no longer empty. However, there is a small delay of two input clock cycles, which can be explained by the two flip-flops synchronizers. Although synchronizers help to ensure the valid state for time-domain crossing, signals are held back and delayed for the equal number of implemented flip-flops. However, the fifo_full flag is active-high after 7 words, indicating the FIFO is fully loaded with words in only 7 rising edges. Therefore, the FIFO logic and testbench must be revised before the completed data buffer implementation.

The main purpose of the second progress report is to update the VHDL code for the asynchronous functionality and present the testbench output of the data buffer system. There are some few unexpected flags behaviors; particularly, the fifo_full. However, the system waveform behavior can mostly model the asynchronous data buffer system. In the last progress report, it is anticipated that the FIFO VHDL logic and testbench produce the complete behaviors of asynchronous FIFO. Moreover, the combinational logic between three FIFOs will also be included and fully explained to indicate the hand-shaking process within the data buffer system.

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

-- added logic *PROCESS* for checking empty/full
-- DIN does not require temp signals cause it can drive all FIFO's but DOUT requires some temp signals


entity data_buff is

 port( ICLK:              in std_logic;
         OCLK:                in std_logic;
         reset:           in std_logic;
         fill_fifo:       in std_logic;
         empty_fifo:     in std_logic;
         DIN:        in std_logic_vector(7 downto 0);
         DOUT:       out std_logic_vector(7 downto 0);
         ready_to_fill:  out std_logic;
         fill_done:      out std_logic;
         ready_to_empty: out std_logic;
         start_empty:    out std_logic;
         empty_done:     out std_logic);
end data_buff;

architecture hier of data_buff is

component asynchronFIFO is
  Generic ( W : natural := 3;   -- RAM address Width in bits
                  D : natural := 8; -- RAM Depth in lines, equals to 2^W - 256 depth Maximum Frequency: 145.773MHz
                  B : natural := 8   -- Input/Output bus width
     );
    Port ( resetR : in  STD_LOGIC;
           resetW : in  STD_LOGIC;
           oclk : in  STD_LOGIC;
           iclk : in  STD_LOGIC;
           enEmpty : in  STD_LOGIC;
           enFill : in  STD_LOGIC;
           fifo_empty : out  STD_LOGIC;
           fifo_full : out  STD_LOGIC;
           FIFO_OUT : out  STD_LOGIC_VECTOR (B-1 downto 0);
           FIFO_IN : in  STD_LOGIC_VECTOR (B-1 downto 0));
end component asynchronFIFO;


signal isEmpty1, isEmpty2, isEmpty3: std_logic;
signal isFull1, isFull2, isFull3: std_logic;
signal raise_ready_to_fill, raise_ready_to_empty: std_logic;
signal temp_DOUT2, temp_DOUT3: std_logic_vector(7 downto 0);

begin
 FIFO1: asynchronFIFO
         generic map (3, 8, 8)
         port map (resetR => reset, resetW => reset, oclk => OCLK, iclk => ICLK,
```

```
                         enEmpty => empty_fifo, enFill => fill_fifo,
                         fifo_empty => isEmpty1, fifo_full => isFull1,
                         FIFO_OUT => DOUT, FIFO_IN => DIN);

 FIFO2: asynchronFIFO
         generic map (3, 8, 8)
         port map (resetR => reset, resetW => reset, oclk => OCLK, iclk => ICLK,
                         enEmpty => empty_fifo, enFill => fill_fifo,
                         fifo_empty => isEmpty2, fifo_full => isFull2,
                         FIFO_OUT => temp_DOUT2, FIFO_IN => DIN); -- change DOUT later

 FIFO3: asynchronFIFO
         generic map (3, 8, 8)
         port map (resetR => reset, resetW => reset, oclk => OCLK, iclk => ICLK,
                         enEmpty => empty_fifo, enFill => fill_fifo,
                         fifo_empty => isEmpty3, fifo_full => isFull3,
                         FIFO_OUT => temp_DOUT3, FIFO_IN => DIN); -- change DOUT later




 full_empty_logic: process(isEmpty1, isEmpty2, isEmpty3, isFull1, isFull2, isFull3)
     begin

         if isEmpty1 = '1' or isEmpty2 = '1' or isEmpty3 = '1' then
             raise_ready_to_fill <= '1';
         else
             raise_ready_to_fill <= '0';
         end if;

         if isFull1 = '1' or isFull2 = '1' or isFull3 = '1' then
             raise_ready_to_empty <= '1';
         else
             raise_ready_to_empty <= '0';
         end if;

     end process;


end architecture hier;



library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;


entity asynchronFIFO is
     Generic ( W : natural := 3;   -- RAM address Width in bits
```

```vhdl
                D : natural := 8; -- RAM Depth in lines, equals to 2^W - 256 depth Maximum Frequency: 145.773MHz
                B : natural := 8   -- Input/Output bus width
     );
    Port ( resetR : in  STD_LOGIC;
           resetW : in  STD_LOGIC;
           oclk : in  STD_LOGIC;
           iclk : in  STD_LOGIC;
           enEmpty : in  STD_LOGIC;
           enFill : in  STD_LOGIC;
           fifo_empty : out  STD_LOGIC;
           fifo_full : out  STD_LOGIC;
           FIFO_OUT : out  STD_LOGIC_VECTOR (B-1 downto 0);
           FIFO_IN : in  STD_LOGIC_VECTOR (B-1 downto 0));
end asynchronFIFO;

architecture Behavioral of asynchronFIFO is

    signal
        full,
        empty
            : std_logic;
    signal
        writePtr,
        syncReadPtrBin,
        readPtrGraySync0,
        readPtrGraySync1,
        writePtrGray,
        readPtrBin,
        readPtr,
        syncWritePtrBin,
        writePtrGraySync0,
        writePtrGraySync1,
        readPtrGray,
        writePtrBin
            : std_logic_vector(W-1 downto 0);
    type
        ramT
            is array (D-1 downto 0) of std_logic_vector(B-1 downto 0);
    signal
        ram
            : ramT;

begin
    write_side : process(iclk)
    begin
        if rising_edge(iclk) then
            if resetW = '1' then
                writePtr <= (others => '0'); --allows to change all bits despite the size
                writePtrGray <= (others => '0');
                syncReadPtrBin <= (others => '0');
                readPtrGraySync0 <= (others => '0');
                readPtrGraySync1 <= (others => '0');
            else
                -- write pointer handling
                if enFill = '1' and not full = '1' then
```

```
                    writePtr <= writePtr + '1';
                end if;
                --write pointer to gray code conversion
                writePtrGray <= writePtr xor ('0' & writePtr(W-1 downto 1));
                --gray coded read pointer synchronisation
                readPtrGraySync0 <= readPtrGray;
                readPtrGraySync1 <= readPtrGraySync0;
                --register read pointer in order to be resetable
                syncReadPtrBin <= readPtrBin;
            end if;
        end if;
    end process;


    --read pointer to binary conversion
    readPtrBin(W-1) <= readPtrGraySync1(W-1);
    gray2binW : for i in W-2 downto 0 generate
        readPtrBin(i) <= readPtrBin(i+1) xor readPtrGraySync1(i);
    end generate;
    --set full flag
    full <= '1' when writePtr + '1' = syncReadPtrBin else '0';
    fifo_full <= full;


    read_side : process(oclk)
    begin
        if rising_edge(oclk) then
            if resetR = '1' then
                readPtr <= (others => '0');
                readPtrGray <= (others => '0');
                syncWritePtrBin <= (others => '0');
                writePtrGraySync0 <= (others => '0');
                writePtrGraySync1 <= (others => '0');
            else
                -- read pointer handling
                if enEmpty = '1' and not empty = '1' then
                    readPtr <= readPtr + '1';
                end if;
                --read pointer to gray code conversion
                readPtrGray <= readPtr xor ('0' & readPtr(W-1 downto 1));
                --gray coded write pointer synchronisation
                writePtrGraySync0 <= writePtrGray;
                writePtrGraySync1 <= writePtrGraySync0;
                --register write pointer in order to be resetable
                syncWritePtrBin <= writePtrBin;
            end if;
        end if;
    end process;
    --write pointer to binary conversion
    writePtrBin(W-1) <= writePtrGraySync1(W-1);
    gray2binR : for i in W-2 downto 0 generate
        writePtrBin(i) <= writePtrBin(i+1) xor writePtrGraySync1(i);
    end generate;
    --set empty flag
    empty <= '1' when readPtr = syncWritePtrBin else '0';
    fifo_empty <= empty;
```

```
    write_in_process : process(iclk)
    begin
        if rising_edge(iclk) then
            if enFill = '1' and not full = '1' then
                ram(conv_integer(writePtr)) <= FIFO_IN;
            end if;
      end if;
    end process;

  read_out_process : process(oclk)
    begin
        if rising_edge(oclk) then
            if enEmpty = '1' and not empty = '1' then
                FIFO_OUT <= ram(conv_integer(readPtr));
            end if;
      end if;
    end process;


end Behavioral;
```