University of Windsor

Electrical and Computer Engineering

ELEC-4430: Embedded System Design

Data Buffering System for FPGA, Progress Report 1

Student ID: **103186504**	Student ID: **104593653**	Student ID: **104754540**
Last Name: **ILIEVSKI**	Last Name: **HUYNH**	Last Name: **KHAN**
First Name: **BRAJAN**	First Name: **KHANH**	First Name: **MAHWISH**
ilievsk1@uwindsor.ca	huynh116@uwindsor.ca	khan1cz@uwindsor.ca

February 22, 2021

Submitted To

Instructor: Dr M Khalid

GA: Rico, Md Maksud-Ul-Kabir

Design Building Blocks

The design of our data buffering system began with efforts focused on the FIFO memory implementation. Given that the design specifications require that the system be driven by two independent clock domains with different frequencies (asynchronous), we initially sought to design a synchronous FIFO memory implementation where a single clock would be used for reading and writing to memory. This was critical in order to understand the logic for three fundamental ideas in FIFO implementation: 1) the read and write pointers required for keeping track of memory addressing, 2) asserting the full and empty flags, and 3) limitations of adapting this design to an asynchronous system.

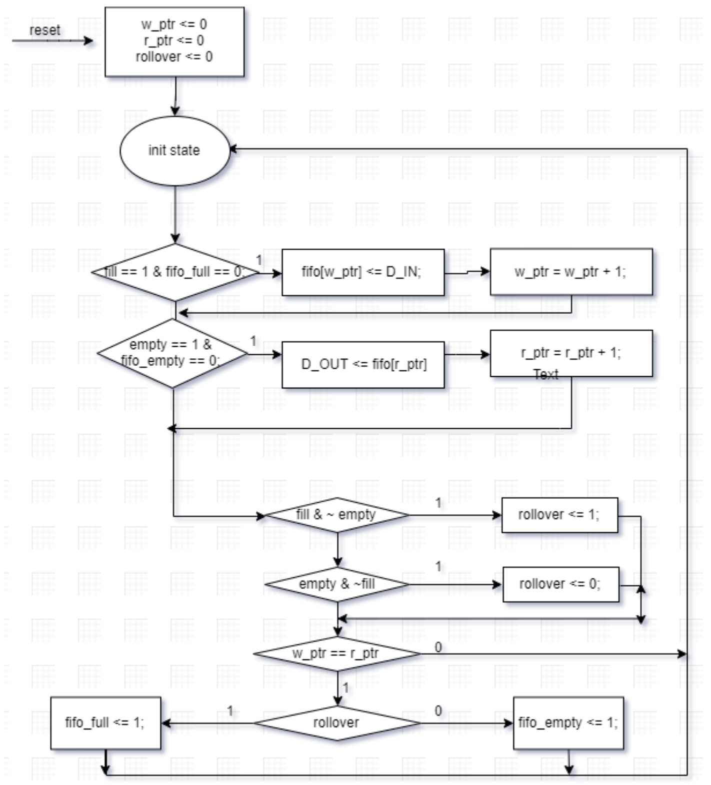


Figure 1: ASM chart for a synchronous FIFO memory design driven by a single clock for read and write operations.

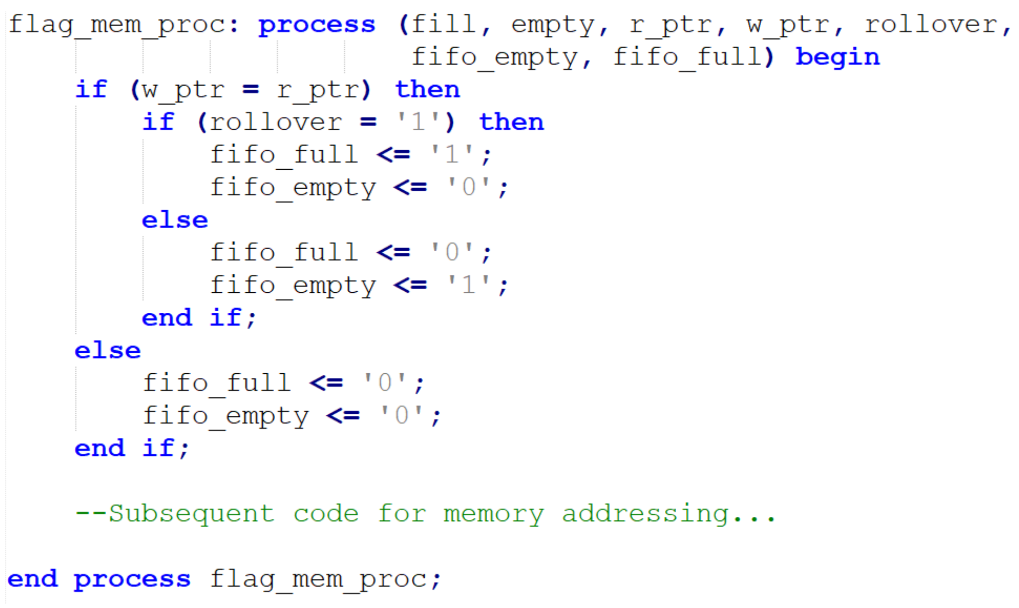
The ASM chart of our synchronous FIFO design is given above in Figure 1. Asserting the *fill* input will initiate writing to the memory location specified by the write pointer *w_ptr*, followed by the subsequent incrementation of the pointer. Given that the FIFO memory module must be first filled then emptied, *empty* will not be asserted, allowing the algorithm to enter the four sequential decision boxes that will check the state of the inputs and whether the pointers are equal. In the case when *fill* is asserted (and *empty* is not), the internal *rollover* signal will be asserted. The *rollover* signal is critical to determine when *w_ptr* has reached the end of memory addressing and has rolled over back to 0. In our design example, this *rollover* will be asserted when *w_ptr* has reached address “111” and will roll over to “000” on the next increment. When this case has occurred, the write pointer will equal the read pointer (“000” == “000”), *rollover* will be set to '1', and the full flag will be asserted.

When the FIFO is full, the *empty* input can be asserted and the memory can be read starting at the address of *r_ptr* (which will be "000"), and the pointer subsequently incremented. Following the ASM further, the *empty & ~fill* condition will be met, *rollover* will be reset to '0', and since the read pointer is no longer equal to the write pointer, the state machine will return back to *init state.* Subsequent reading will occur at the next memory locations and *r_ptr* will be incremented until the read pointer itself has rolled over and becomes equal to the write pointer once again ("000" == "000"). In this case the *fifo_empty* flag will be asserted and the memory will be ready for a new write cycle.

To illustrate the implementation of this ASM in a synthesizable design, consider the VHDL code for the synchronous FIFO memory implementation at the end of this report. At the core of our synchronous FIFO memory design are two concurrently running processes that handle the assignment of either the read/write signal pointers or the full/empty flags and memory addressing: *ptr_proc* and *flag_mem_proc,* respectively. The *ptr_proc* process (driven solely by the clock) is responsible for initializing the pointers and the *rollover* bit upon reset. When the *empty* input is given and the FIFO is empty, the write pointer is incremented, and *rollover* is set to zero. The process for the *fill* command is similar whereby the read pointer is incremented but *rollover* is set to 1.

```
ptr_proc: process (clk) begin
    if(rising_edge(clk)) then
        if (reset = '1') then
            w_ptr <= "000";
            r_ptr <= "000";
            rollover <= '0';
        elsif (empty = '1' and fifo_empty = '0') then
            r_ptr <= r_ptr + 1;
            rollover <= '0';
        elsif (fill = '1' and fifo_full = '0') then
            w_ptr <= w_ptr + 1;
            rollover <= '1';
        end if;
    end if;
end process;
```

The concurrently running *flag_mem_proc* is independent of the clock and driven by changes in the inputs, pointers, and rollover bit. After initialization of the FIFO, the read and write pointers are equal (both initialized to zero), the *rollover* bit is set to zero, and the *if* condition asserts the *fifo_empty* flag. When writing operations have begun, the write pointer begins to increment, rollover is set to 1, and when the FIFO memory has been written to eight times, *w_ptr* transitions from "111" to "000", and the read and write pointers are once again equal. However, in this case, the *rollover* bit has been set and the *if* condition asserts the *fifo_full* flag. It is observed that the *rollover* bit is critical to differentiate between the two equal conditions that occur when the FIFO is either empty (rollover = 0) or full (rollover = 1).



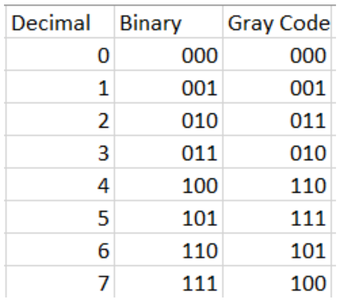
```
flag_mem_proc: process (fill, empty, r_ptr, w_ptr, rollover,
                        fifo_empty, fifo_full) begin
    if (w_ptr = r_ptr) then
        if (rollover = '1') then
            fifo_full <= '1';
            fifo_empty <= '0';
        else
            fifo_full <= '0';
            fifo_empty <= '1';
        end if;
    else
        fifo_full <= '0';
        fifo_empty <= '0';
    end if;

    --Subsequent code for memory addressing...

end process flag_mem_proc;
```

Challenges with an Asynchronous Design

Although this is a sound design for a synchronous FIFO operating in one clock domain, we encountered limitations when attempting to adapt this design for asynchronous dual-clock operation. First, the asynchronous design requires that the read and write pointers be driven by their respective *oclk* and *iclk* signals. Although this can be achieved by separating the initial *ptr_proc* process into two individual processes, we encountered challenges with the *rollover* bit, which cannot be driven by two separate clock signals. This mandates that we abandon the idea of the *rollover* bit and instead opt for a new approach that is safe for an asynchronous design in order to differentiate between the equal conditions when the FIFO is full or empty.

Another limitation with extending this design into asynchronous operation is related to timing and signal instability. In the asynchronous FIFO, the output clock operates at one fourth the frequency of the input clock, which creates challenges when attempting to compare the binary read and write pointers while they are being driven on the positive edge of their respective clocks. This establishes the challenge of metastable states, whereby changing signals have not settled to their final states before the pointer values are to be compared [1, 2]. Further, there exists inherent timing issues when attempting to compare the binary pointers in different clock domains when multiple bits change simultaneously. This can result in inaccurate comparison of the pointers, especially during roll over instances where the pointers transition from “111” to “000” and any intermediate combination of simultaneous bit changes may be captured before the final value settles. Therefore, additional approaches such as Gray Code counters and Synchronizers are implemented to resolve the timing issue and establish a stable comparison between the *iclk* driven write pointer and the *oclk* driven read pointer [1, 2].

Gray Code or reflected binary code: is a form of binary that only change one bit from one position to another. Gray Code is applicable in many different areas of digital processing, particularly, in asynchronous FIFO pointer.



Decimal	Binary	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Figure 3: a sample 3-bit binary to gray code conversation

In the process of implementing asynchronous FIFO, Gray Code is utilized to limit the change of read and write pointer to one bit every clock cycle. Consequently, Gray Code can prevent the capture of transient states of the pointer bits when being transferred to the opposite sides (clock domain crossing). Therefore, Gray Code can resolve the in which not only minimized the amount of switching but also improve the reliability of the switching systems.

Synchronizers: Clock synchronization flip flops resolve the metastability between two different clock-driven signals, particularly, write and read pointers. Fundamentally, synchronizers consist of flip flops to store the changing signal. In the design of asynchronous FIFO, a two-stage D flip-flop synchronizer is inserted to retain the signal behavior during crossing clock domains.

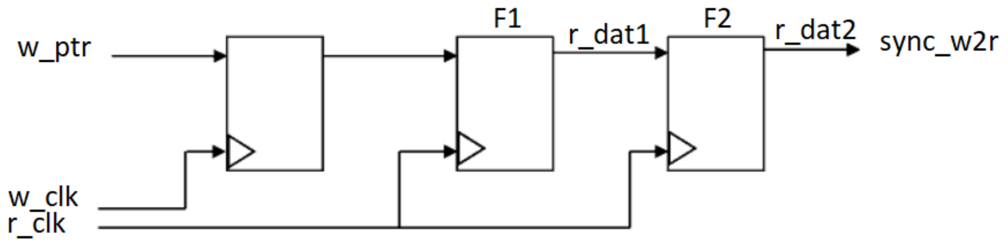


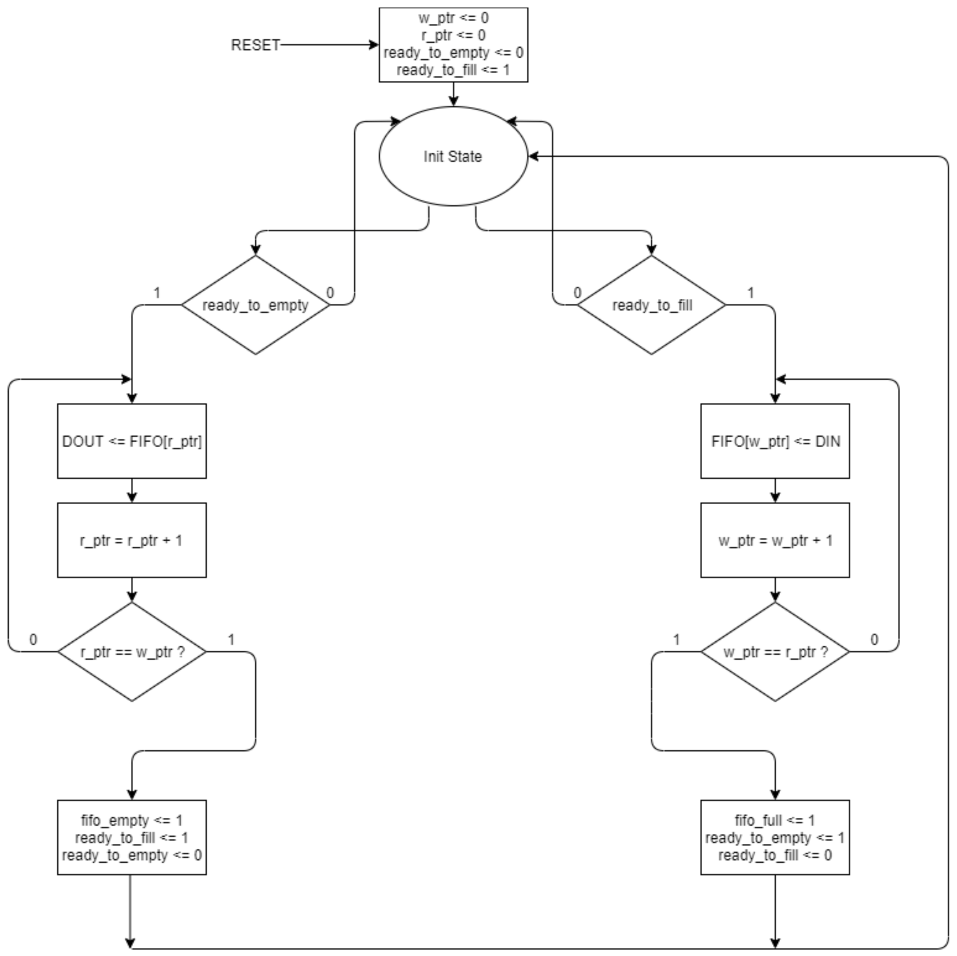
Figure 4: a sample 2 flip flops synchronizers

The first flip flop samples the asynchronous input signal into the other clock domain. The first flip flop will store and wait any metastability of the signal to decay; this issue might occur when the input signal changes value during the setup time of the flip flop. Therefore, in the second clock cycle r_clk, the output from the second flip flop is now stable and synchronized to the destinated clock domain. The combination of Grey Code and D flip-flop synchronizers will enable the stable comparison of Grey Code pointers in the following manner: 1) the native write pointer in the *iclk* domain against a synchronized read pointer, and 2) the native read pointer in the *oclk* domain against a synchronized write pointer. It is important to keep in mind that the comparison of Grey Code pointers will require additional logic for binary-to-Gray and Gray-to-binary conversion logic, which will be undertaken in the next progress report.

<u>ASM Chart Development for an Asynchronous System</u>

As the group builds upon our understanding of a synchronous, single FIFO design to an asynchronous, multiple FIFO data buffering system, there are iterations of the ASM chart that are being updated to reflect our revised comprehension.

Figure 5 below is the second iteration of an ASM chart, focusing on the algorithm for synchronously reading and writing to a FIFO memory block. As can be seen, the reading and writing processes have been separated despite their similarities and this is done as it allows for an organized manner of viewing the logic flow that takes place for each situation. Additionally, when the time comes to separate the driving clocks, these flows can be separated into two processes in VHDL, with their respective clocks present in the sensitivity lists.



<u>Figure 5</u>: ASM chart in progress for an asynchronous FIFO memory design, with a focus on read/write operation logic

This ASM chart depicts a continuous, synchronous fill and refill of a single FIFO element and can be traced quite simply and quickly. In the event of an asynchronous reset, all pointers will be set to 0 as well as the *ready_to_empty* since there is no current data stored for the user. The *ready_to_fill* flag will be raised, indicating that information can be inserted for the time being. After the init state, when the FIFO is being filled it will take in each word from the *DIN* bus and increment the *w_ptr* until it reaches back to the original point. In a general-purpose FIFO, when these two pointers are equal, it would suggest that the FIFO is either empty or full. To distinguish between those two cases, an additional flag (for example, one dubbed "*rollover*") could be implemented. When the FIFO has been filled to its depth, this *rollover* flag would be set to true, indicating that as the write pointer returns to its initial position and is now equal to the read pointer, the FIFO is in fact full. In contrast, if *rollover* is false, that would mean that the FIFO depth has not yet been filled and if the pointers are equal, then it must in fact be empty. For the data buffering system to be designed, the spec indicates that any time a FIFO is being written to, it will be filled with 8 words in 8 clock cycles and will not ever be partially filled. As such, the rollover flag is not imperative to this asynchronous design because anytime a write process is triggered, the FIFO will be filled, the *ready_to_empty* flag will be set to true and the *ready_to_*fill flag will be set to false, and these can be used as an indication that the FIFO is in fact full. With the same logic, anytime a FIFO is being read from, all 8 words will be subsequently read and the *ready_to_empty* flag will become false and the *ready_to_fill* flag will become true, again clearly providing a way to check whether the FIFO is full or empty. That is why the *rollover* flag has been eliminated in Figure 5.

The next steps moving forward are to expand the ASM to incorporate the combinatorial logic that allows for selecting which FIFO of the three will be written or read to. From the lab spec, an important note is that the reading and writing will always occur in the order of FIFO1, FIFO2, FIFO3. This logic will occur after the *init* state, because the logic presented in terms of the pointers will remain valid. Additionally, as mentioned earlier in the report, a critical design aspect to consider is the synchronization to resolve metastability issues due to the different driving clocks. When adding this logic, there will be more crossover between the read and write column logic as pointer values are converted to gray code and relayed to each process, allowing for comparison once they are converted back to binary.

In the upcoming progress report, it is expected that the next iteration of the ASM chart will reflect the full data buffering system. As code is being developed for the synchronous design, it is providing insight on limitations that are present which allows us to update the ASM chart to handle these issues. Finally, the updated ASM chart will lead to writing synthesizable code for the asynchronous design.

References

[1] C. E. Cummings, “Simulation and Synthesis Techniques for Asynchronous FIFO Design,” 2002. [Online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf. [Accessed: 22-Feb-2021].

[2] R. Donohue, “Synchronization in Digital Logic Circuits,” Standford Education. [Online]. Available: https://web.stanford.edu/class/ee183/handouts/synchronization_pres.pdf. [Accessed: 22-Feb-2021].

[3] Clock Domain Crossing. [Online]. Available: https://filebox.ece.vt.edu/~athanas/4514/ledadoc/html/pol_cdc.html. [Accessed: 23-Feb-2021].

[4] “ASM Methodology for RTL Designs,” ASM : a modern Algorithmic State Machine methodology for RTL designs. [Online]. Available: http://www.deeper.uva.es/asm /index.php?part=5. [Accessed: 19-Feb-2021].

```vhdl
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
USE IEEE.NUMERIC_STD.ALL;

entity FIFO is
    port(
            clk     : in  std_logic;
            empty       : in std_logic;
            fill        : in std_logic;
            reset       : in std_logic;
            DIN         : in std_logic_vector(7 downto 0);
            fifo_empty  : buffer std_logic;
            fifo_full   : buffer std_logic;
            DOUT        : out std_logic_vector(7 downto 0) := "ZZZZZZZZ");
end FIFO;

architecture FIFO_arch of FIFO is

    --Memory Signals
    subtype word is std_logic_vector (7 downto 0);
    type memory is array (0 to 7) of word;
    signal RAM : memory;
    signal addr: std_logic_vector (2 downto 0);
    signal write_en: std_logic;

    --FIFO Control Signals
    signal r_ptr: std_logic_vector(2 downto 0);
    signal w_ptr: std_logic_vector (2 downto 0);
    signal rollover: std_logic;

begin

ptr_proc: process (clk) begin
    if(rising_edge(clk)) then
        if (reset = '1') then
            w_ptr <= "000";
            r_ptr <= "000";
            rollover <= '0';
        elsif (empty = '1' and fifo_empty = '0') then
            r_ptr <= r_ptr + 1;
            rollover <= '0';
        elsif (fill = '1' and fifo_full = '0') then
            w_ptr <= w_ptr + 1;
            rollover <= '1';
        end if;
    end if;
end process ptr_proc;


flag_mem_proc: process (fill, empty, r_ptr, w_ptr, rollover,
                        fifo_empty, fifo_full) begin
    if (w_ptr = r_ptr) then
        if (rollover = '1') then
            fifo_full <= '1';
            fifo_empty <= '0';
        else
            fifo_full <= '0';
            fifo_empty <= '1';
        end if;
    else
        fifo_full <= '0';
        fifo_empty <= '0';
    end if;

    if (fill = '0' and empty = '0') then
        addr <= r_ptr;
        write_en <= '0';
    elsif (fill = '1' and empty = '0') then
```

```vhdl
            addr <= w_ptr;
            if (fifo_full = '0') then
                write_en <= '1';
            else
                write_en <= '0';
            end if;
        elsif (fill = '0' and empty = '1') then
            addr <= r_ptr;
            write_en <= '0';
        else
            if (fifo_empty = '0') then
                addr <= r_ptr;
                write_en <= '0';
            else
                addr <= w_ptr;
                write_en <= '1';
            end if;
        end if;
    end process flag_mem_proc;

    --MEMORY
    process (write_en)
        begin
                if (write_en = '0') then
                    DOUT <= RAM(to_integer(unsigned(addr)));
                elsif (write_en = '1') then
                    RAM(to_integer(unsigned(addr))) <= DIN;
                    DOUT <= "ZZZZZZZZ";
                end if;

    end process;

    end FIFO_arch;
```