HO CHI MINH UNIVERSITY OF TECHNOLOGY

ELECTRICAL ELECTRONIC ENGINEERING

Electronic Department

----------o0o----------

LOGIC DESIGN / LOGIC SYNTHESIS

LAB 3

ASYNCHRONOUS FIFO

INSTRUCTORS: Tran Hoang Linh

Nguyen Tuan Hung

STUDENT : Ha Gia Huy _ 2051117

Tran Minh Tri _ 2051022

HO CHI MINH CITY, 2024

Table of Content

List of Figure

I. OVERVIEW

The objective of this lab is an asynchronous FIFO module and conduct its verification using assertions.

The FIFO controller assumes responsibility for managing the memories by executing data writes and reads. Concurrently, the FIFO controller is capable of performing read and write operations simultaneously. We have previously acquired familiarity with utilizing the Altera IP Catalog Memory through Lab 2, wherein we employed it for memory implementation and incorporated megafunction files for simulation purposes. The memory module supports the storage of 24-bit data and can accommodate a maximum of 32 entries. When the memory reaches full capacity, it must cease accepting additional data. Conversely, when empty, it should prevent data read operations. Following the design of the FIFO Controller and its associated memories, a testbench is created to validate its functionality. The testbench is engineered to simulate all conceivable scenarios and ascertain the correctness of the FIFO controller's operations within each. A successful outcome confirms testbench functionality, while any deviation represents a failure.

In our comprehensive test plan, structured with three columns titled "Test number | Case name | Case description | Pass Condition | Fail Condition", we outline twelve distinct test cases. We're going to be implementing a FIFO (First-In-First-Out) structure on the DE10 Kit FPGA board. This implementation will involve using a component known as a "Linear Feedback Shift Register" (LFSR) to generate random numbers. Additionally, we will create a top-level module named top_lab3 to interconnect the LFSR, FIFO controller, Memory, BCD7SEG, and a clock divider.

II. THEORY

2.1. First-In-First-Out Structure

FIFOs are commonly used in electronic circuits for buffering and flow control between hardware and software. In its hardware form, a FIFO primarily consists of a set of read and write pointers, storage and control logic. Storage may be static random access memory (SRAM), flip-flops, latches or any other suitable form of storage. For FIFOs of non-trivial size, a dual-port SRAM is usually used, where one port is dedicated to writing and the other to reading.

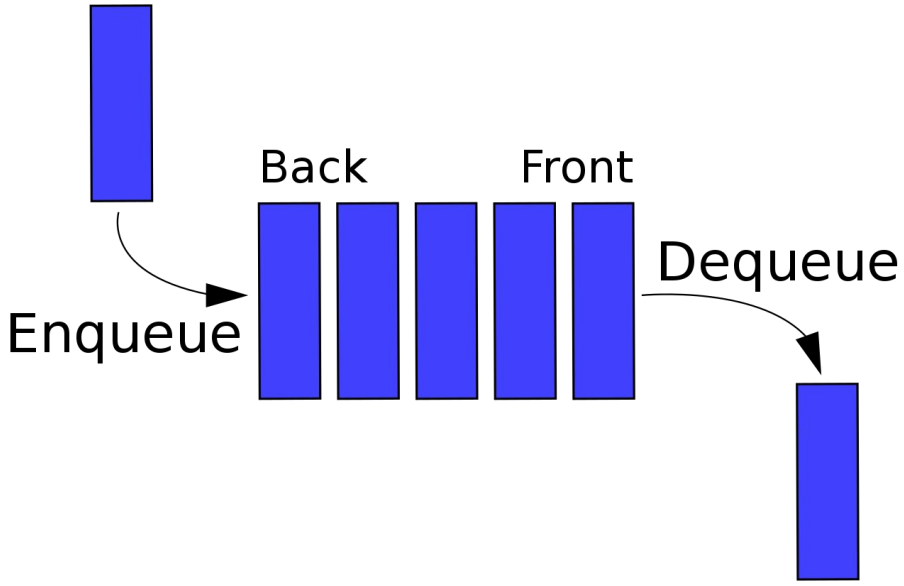


Figure 1. Representation of a FIFO queue

A synchronous FIFO is a FIFO where the same clock is used for both reading and writing. An asynchronous FIFO uses different clocks for reading and writing and they can introduce metastability issues. A common implementation of an asynchronous FIFO uses a Gray code (or any unit distance code) for the read and write pointers to ensure reliable flag generation. One further note concerning flag generation is that one must necessarily use pointer arithmetic to generate flags for asynchronous FIFO implementations. Conversely, one may use either a leaky bucket approach or pointer arithmetic to generate flags in synchronous FIFO implementations. A hardware FIFO is used for synchronization purposes. It is often implemented as a circular queue, and thus has two pointers: Read pointer/read address register & Write pointer/write address register.

FIFO status flags include: full, empty, noempty. A FIFO is empty when the read address register reaches the write address register. A FIFO is full when the write address register reaches the read address register. Read and write addresses are initially both at the first memory location and the FIFO queue is empty.

In both cases, the read and write addresses end up being equal. To distinguish between the two situations, a simple and robust solution is to add one extra bit for each read and write address which is inverted each time the address wraps. With this set up, the disambiguation conditions, when the read address register equals the write address register, the FIFO is empty. When the read and write address registers differ only in the extra most significant bit and the rest are equal, the FIFO is full.

2.2. Linear Feedback Shift Register

A linear-feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state. The most commonly used linear function of single bits is exclusive-or (XOR). Thus, an LFSR is most often a shift register whose input bit is driven by the XOR of some bits of the overall shift register value. The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and has a very long cycle.

Applications of LFSRs include generating pseudo-random numbers, pseudo-noise sequences, fast digital counters, and whitening sequences. Both hardware and software implementations of LFSRs are common. The mathematics of a cyclic redundancy check, used to provide a quick check against transmission errors, are closely related to those of an LFSR.[1] In general, the arithmetics behind LFSRs makes them very elegant as an object to study and implement. One can produce relatively complex logics with simple building blocks. However, other methods, that are less elegant but perform better, should be considered as well.

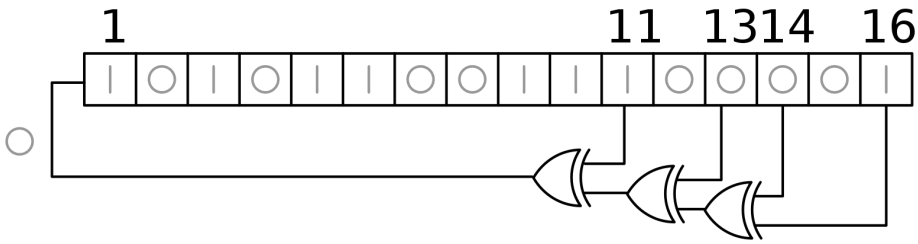


Figure 2. A 16-bit Fibonacci LFSR

The bit positions that affect the next state are called the taps. In the diagram the taps are [16,14,13,11]. The rightmost bit of the LFSR is called the output bit, which is always also a tap. To obtain the next state, the tap bits are XOR-ed sequentially; then, all bits are shifted one place to the right, with the rightmost bit being discarded, and that result of XOR-ing the tap bits is fed back into the now-vacant leftmost bit. To obtain the pseudorandom output stream, read the rightmost bit after each state transition. The arrangement of taps for feedback in an LFSR can be expressed in finite field arithmetic as a polynomial mod 2. This means that the coefficients of the polynomial must be 1s or 0s. This is called the feedback polynomial or reciprocal characteristic polynomial. For example, if the taps are at the 16th, 14th, 13th and 11th bits (as shown), the feedback polynomial is $x^{16} + x^{14} + x^{13} + x^{11} + 1$

2.3. Frequency Divider

A frequency divider, also called a clock divider or scaler or prescaler, is a circuit that takes an input signal of a frequency, f_{in}, and generates an output signal of a frequency: $f_{out} = \frac{f_{in}}{N}$ where N is an integer. Phase-locked loop frequency synthesizers make use of frequency dividers to generate a frequency that is a multiple of a reference frequency. Frequency dividers can be implemented for analog / digital applications.

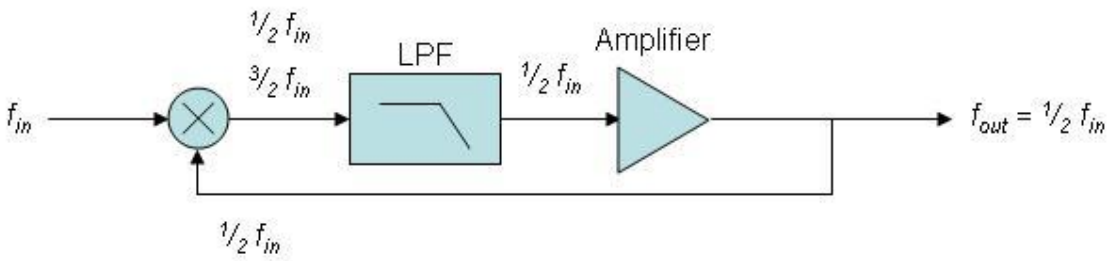


Figure 3. A regenerative frequency divider (Miller frequency divider)

The feedback signal is fin/2. This produces sum and difference frequencies fin/2, 3fin/2 at the output of the mixer. A low pass filter removes the higher frequency, and the fin/2 frequency is amplified and fed back into the mixer.

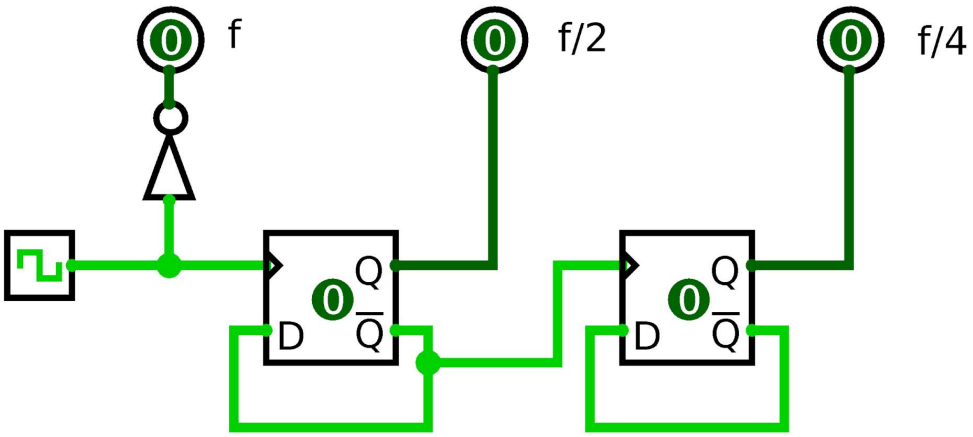


Figure 4. A frequency divider implemented with D flip-flops

For power-of-2 integer division, a simple binary counter can be used, clocked by the input signal. The least-significant output bit alternates at 1/2 the rate of the input clock, the next bit at 1/4 the rate, the third bit at 1/8 the rate, etc. An arrangement of flipflops is a classic method for integer-n division. Such division is frequency and phase coherent to the source over environmental variations, including temperature. The easiest configuration is a series where each flip-flop is a divide-by-2. For a series of three of these, such a system would be a divide-by-8. By adding additional logic gates to the chain of flip-flops, other division ratios can be obtained. Integrated circuit logic families can provide a single-chip solution for some common division ratios.

Another popular circuit to divide a digital signal by an even integer multiple is a Johnson counter. This is a type of shift register network that is clocked by the input signal. The last register's complemented output is fed back to the first register's input. The output signal is derived from one or more of the register outputs. For example, a divide-by-6 divider can be constructed with a 3-register Johnson counter. The six valid values of the counter are 000, 100, 110, 111, 011, and 001. This pattern repeats each time the input signal clocks the network. The output of each register is an f/6 square wave with 120° of phase shift between registers. Additional registers can be added to provide additional integer divisors.

2.4. Altera IP Catalog Memory

A memory block contains two address ports (port A and port B) with their respective output data ports, and you can use them for read and write operations depending on the memory mode you choose. The input and output ports shown in the block diagrams refer to the ports of the wrapper that contains the memory IP core instantiated in it. The ports of the wrapper are mapped to the ports of either the ALTSYNCRAM or the ALTDPRAM IP core depending on your memory configuration, and the port name reflects the memory features you create. For example, the name of the wrapper port clockena maps to the clock_enable_input_a port of the ALTSYNCRAM IP core, which relates to the clock enable feature.

Memory IP	Supported Memory Mode	Features
RAM: 1-PORT	Single-port RAM	• Non-simultaneous read and write operations from a single address. • Read enable port to specify the behavior of the RAM output ports during a write operation, to overwrite or retain existing value. • Supports freeze logic feature.
RAM: 2-PORT	Simple dual-port RAM	• Simultaneous one read and one write operations to different locations. • Supports error correction code (ECC). • Supports freeze logic feature.
	True dual-port RAM	• Simultaneous two reads. • Simultaneous two writes. • Simulatenous one read and one write at two different clock frequencies. • Supports freeze logic feature.
ROM: 1-PORT	Single-port ROM	• One port for read-only operations. • Initialization using a **.mif** or **.hex** file.
ROM: 2-PORT	Dual-port ROM	• Two ports for read-only operations. • Initialization using a **.mif** or **.hex** file.

Figure 5. Memory IP Cores and Their Features

In simple dual-port RAM mode, a dedicated address port is available for each read and write operation (one read port and one write port). A write operation uses write address from port A while read operation uses read address and output from port.

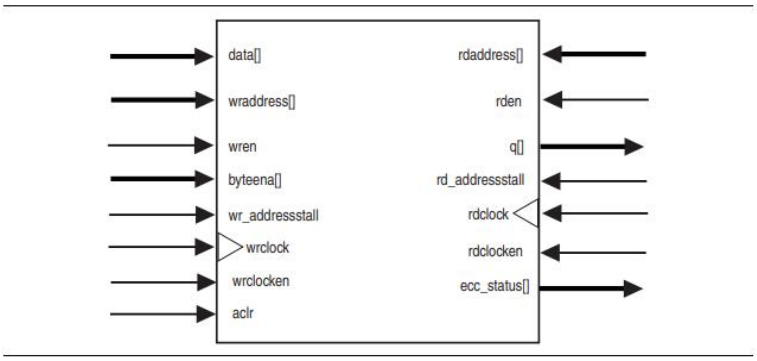


Figure 6. Block diagram of a simple dual-port RAM

III. DESIGN STRATEGY

Here are the specific connections and functionalities for the top_lab3 module:

3.1. Clock Connections

- Use the CLOCK_2SEC (0.5Hz clock) to drive the LFSR and the write FIFO controller write clock (clkw).
- Use the CLOCK_1SEC (1 Hz clock) to drive the FIFO controller read clock (clkr).

3.2. Output Connections

Use SW0 to toggle between displaying the memory data and the FIFO length to the HEX (HEX5 to HEX0).

- SW0 == 0: The HEX displays the output data from the Memory (24-bit)
- SW0 == 1: The HEX displays the fifo_len signal.

3.3. Control Inputs

- Use KEY0 to enable writing into the FIFO controller.
- Use KEY1 to enable reading from the FIFO controller.
- Use KEY2 to reset the FIFO controller.

3.4. Status Indicators

Utilize three LEDs (LEDR0, LEDR1, LEDR2) for status indication:

- LEDR0: Turns on when the FIFO is full.
- LEDR1: Turns on when the FIFO is not empty.
- LEDR2: Turns on when the FIFO is empty.

Ensure that the top_lab3 module correctly manages the control signals for writing to and reading from the FIFO, utilizes appropriate clocks for timing, and updates the status indicators and display based on the current state of the FIFO and memory.

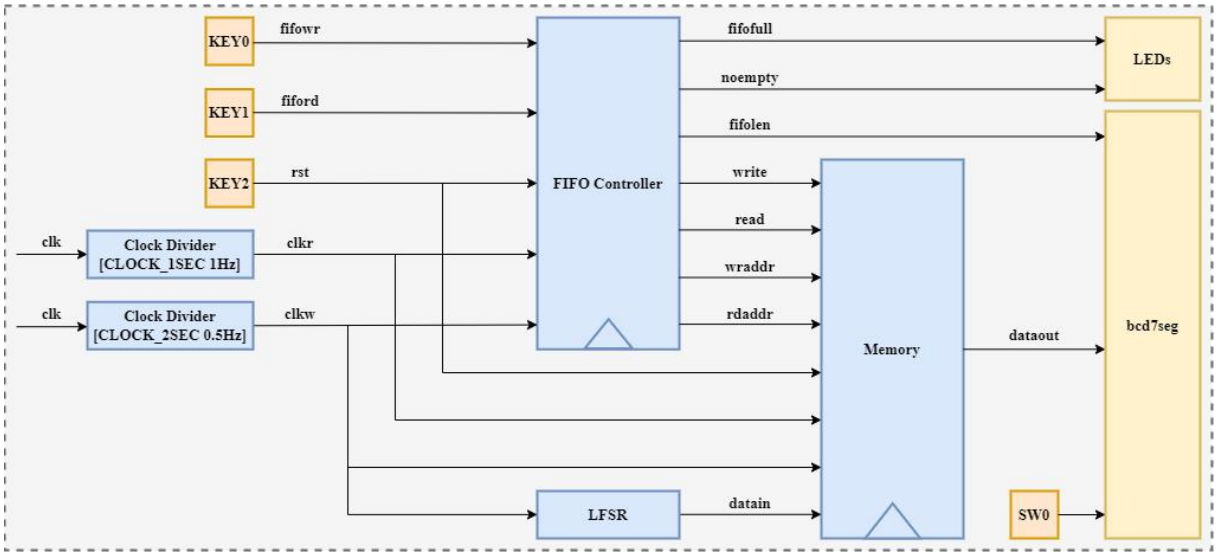


Figure 7. Block Diagram of *the top_lab3 module*

Our FPGA design carefully integrate these components and functionalities to achieve the desired FIFO behavior on the DE10 Kit board. Each module (LFSR, FIFO controller, Memory, BCD7SEG, clock divider) should be instantiated and interconnected within the top_lab3 top-level module to ensure proper functionality and communication between components.

IV. VERIFICATION STRATEGY *(Test Plan)*

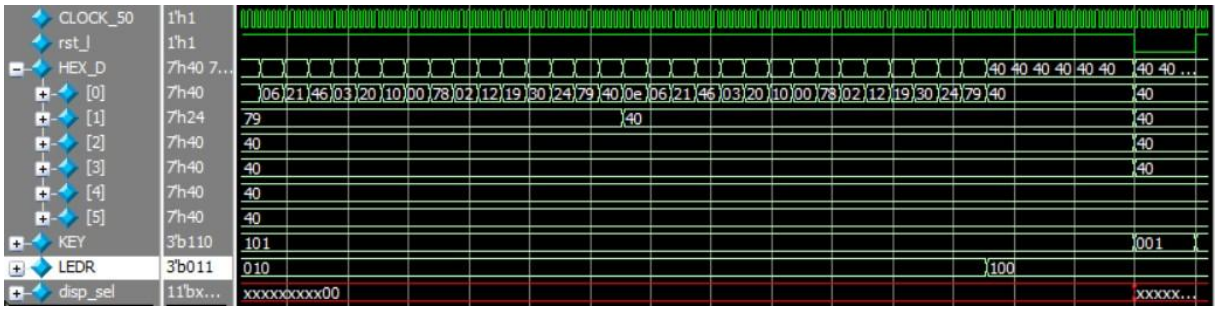
Test Number	Case Name	Case Description	Pass Condition	Fail Condition
1	write_operation	Check operation of write_pointer /write_address	write_addr: increment by 1 when success. write_addr: current value is 0, previous value is 32.	else
2	read_operation	Check operation of read_pointer /read_address	read_addr: increment by 1 when success. read_addr: current value is 0, previous value is 32.	else
3	fifo_len_write	Check fifo_len after each write_operation	fifo_len: increment by 1 when success. fifo_len: stable when 1 successful read previously.	else

			fifo_len: decrement by 1 when 2 successful read previously	
4	fifo_len_read	Check fifo_len after each read_operation	fifo_len: decrement by 1 when success. fifo_len: stable when 1 successful write previously. fifo_len: decrement by 1.	else
5	reset	Check control signals of fifo (full, wptr,…) after reset	control_signals: return the initial value when reset is ON	else
6	dont_write_if_ful	rule_1:"Never write to fifo when full"	write_addr: stable when fifo is FULL	else
7	dont_read_if_empty	rule_2:"Never read from fifo when empty"	read_addr: stable when fifo is EMPTY	else
8	full_empty_not_both_high	Check full and empty signals not high at the same time	write_en | read_en are high, full and empty is not both high at the all time	else
9	check_data_fifo	Check data written and data read is same at the same location	data_in && write_addr when write available is same data_out && read_addr when read_available.	else
10	reset_mem	Check the data output from memory when reset is ON	data_out equal 0 when reset is ON	else
11	mem_write	Check the data input of memory when being available write	data_in is not X and Z when availabe write	else
12	mem_read	Check the data out from memory when being available read	data_out is not X and Z when available read	else

V. SIMULATIONING ON QUESTA SIM

Case 1: reset is ON; write to FIFO until it is FULL; SW = 0 to display fifo_len

Case 2: read from FIFO until it is EMPTY; SW = 0 to display fifo_len



Case 3: read from FIFO until it is EMPTY; SW = 1 to display data_out


```
# ------------------------------------------------------------------------------------------
# TEST_NUMBER   |       TEST_CASE                |       PASS    |       FAIL    |
# ------------------------------------------------------------------------------------------
# 01            |       write_operation          |       214     |       0       |
# 02            |       read_operation           |       214     |       0       |
# 03            |       fifo_len_write           |       214     |       0       |
# 04            |       fifo_len_read            |       214     |       0       |
# 05            |       reset                    |       50      |       0
# 06            |       dont_write_if_full       |       35      |       0       |
# 07            |       dont_read_if_empty       |       84      |       0       |
# 08            |       full_empty_not_both_high |       2652    |       0       |
# 09            |       check_data_fifo          |       136     |       0       |
# 10            |       reset_mem                |       50      |       0       |
# 11            |       mem_write                |       211     |       0       |
# 12            |       mem_read                 |       210     |       0       |
# ------------------------------------------------------------------------------------------
#
# ** Note: $finish    : D:/ktsnc/Lab3/lab3new/testbench_SVA.sv(187)
```

VI. IMPLEMENTING ON HARDWARE

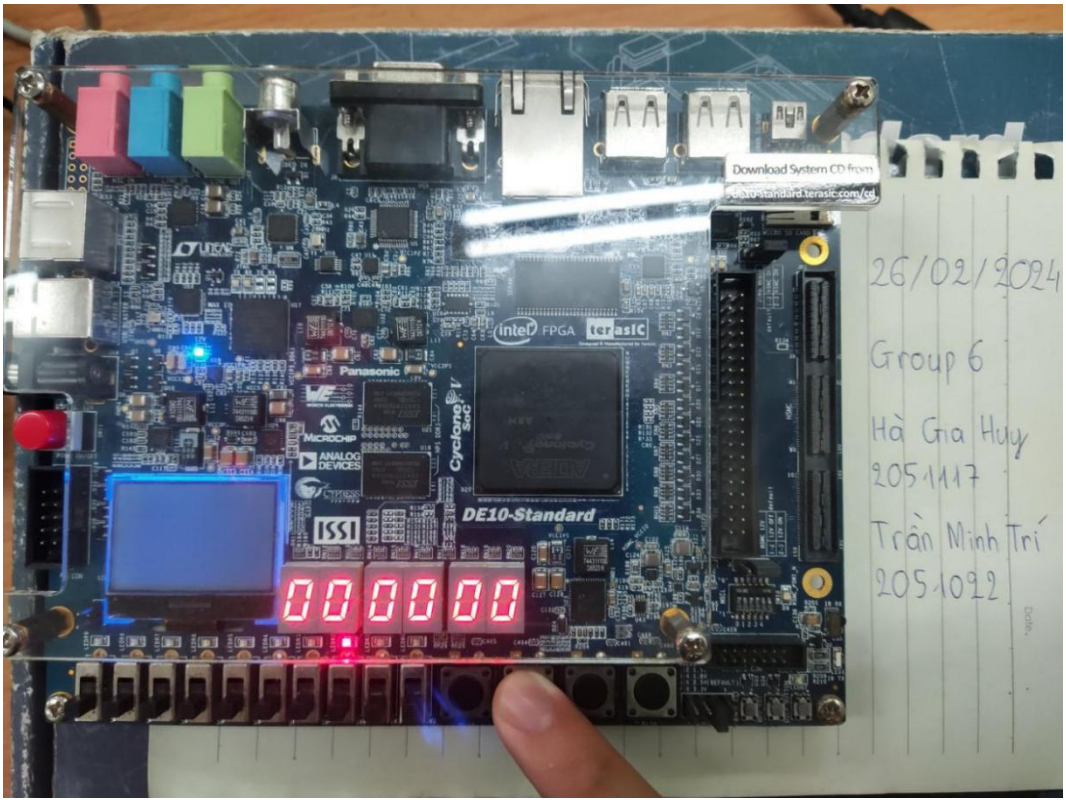


Figure 8. Reset Operation (FIFO_empty status)

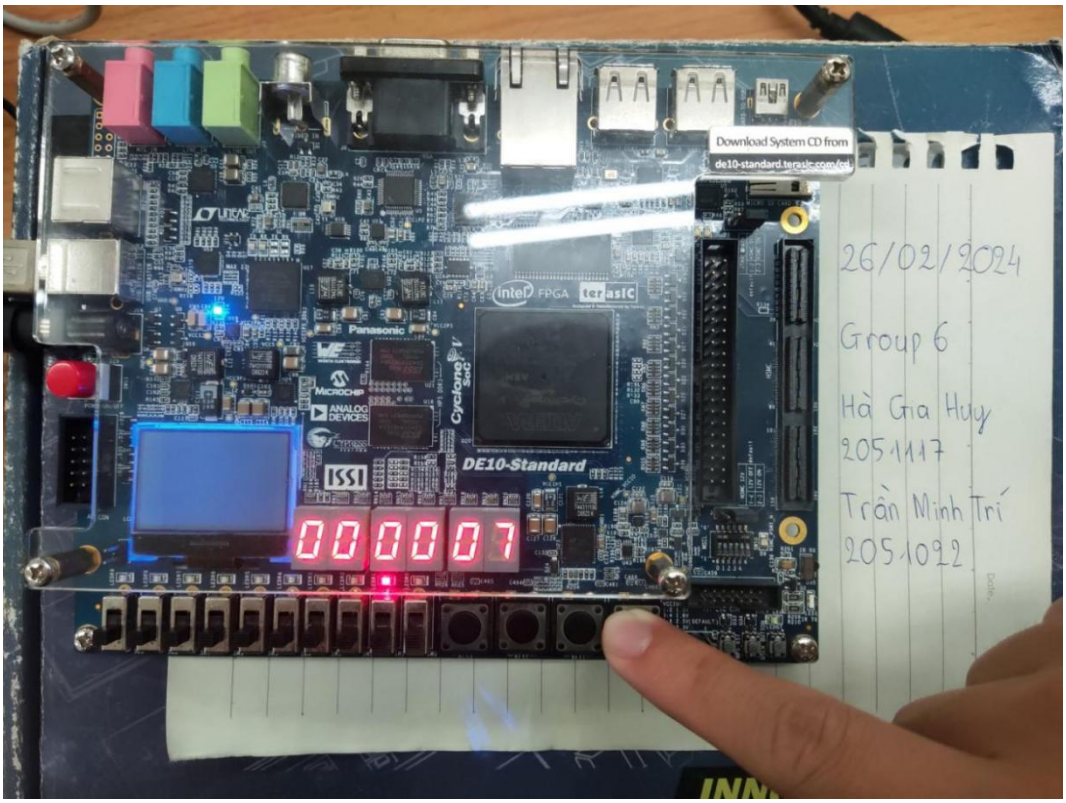


Figure 9. Write Operation

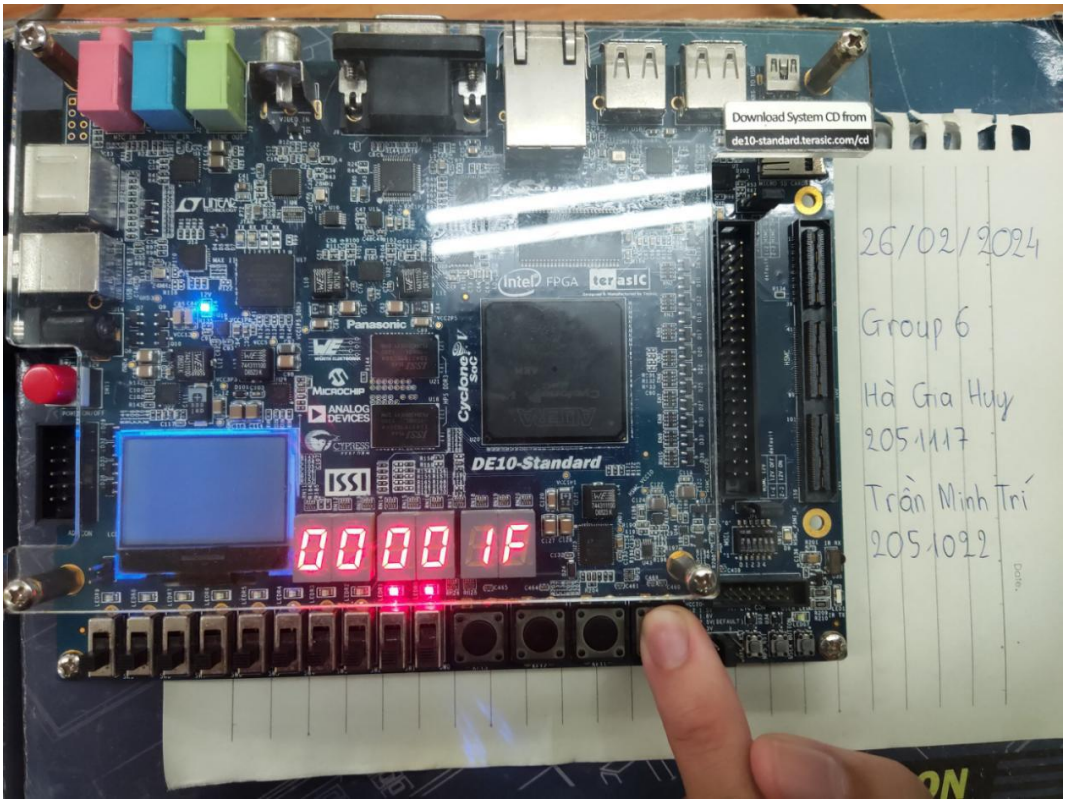


Figure 10. FIFO_full status

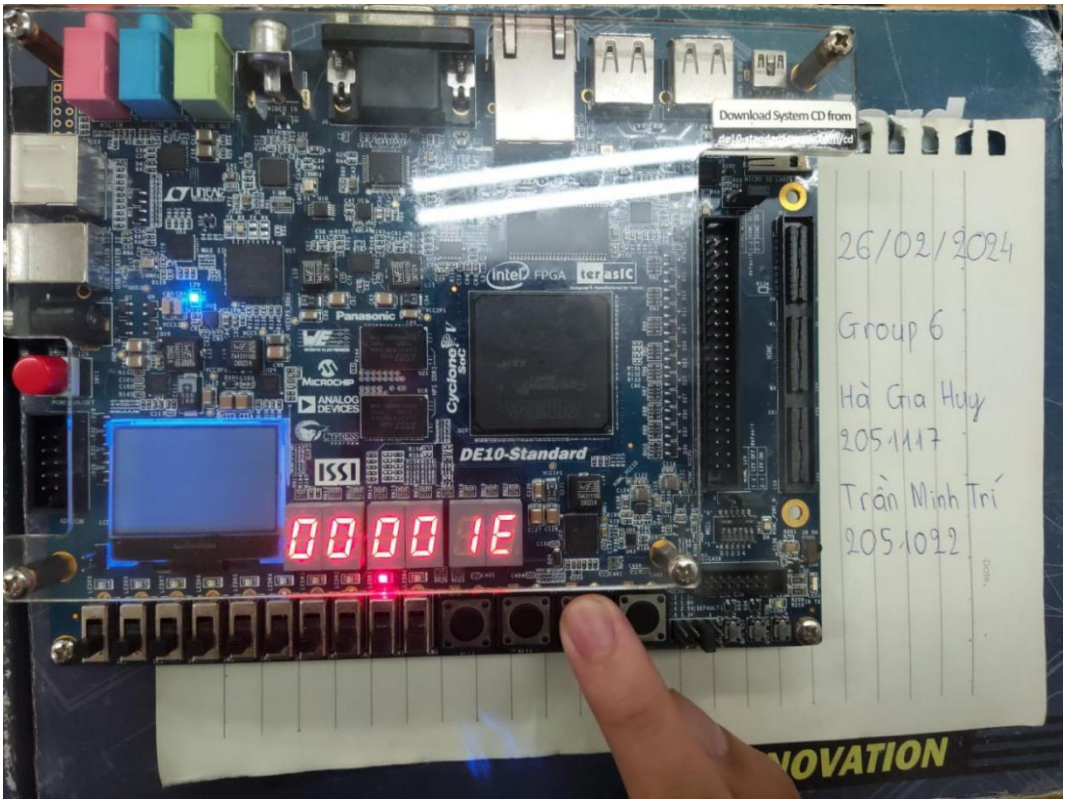


Figure 11. Read Operation

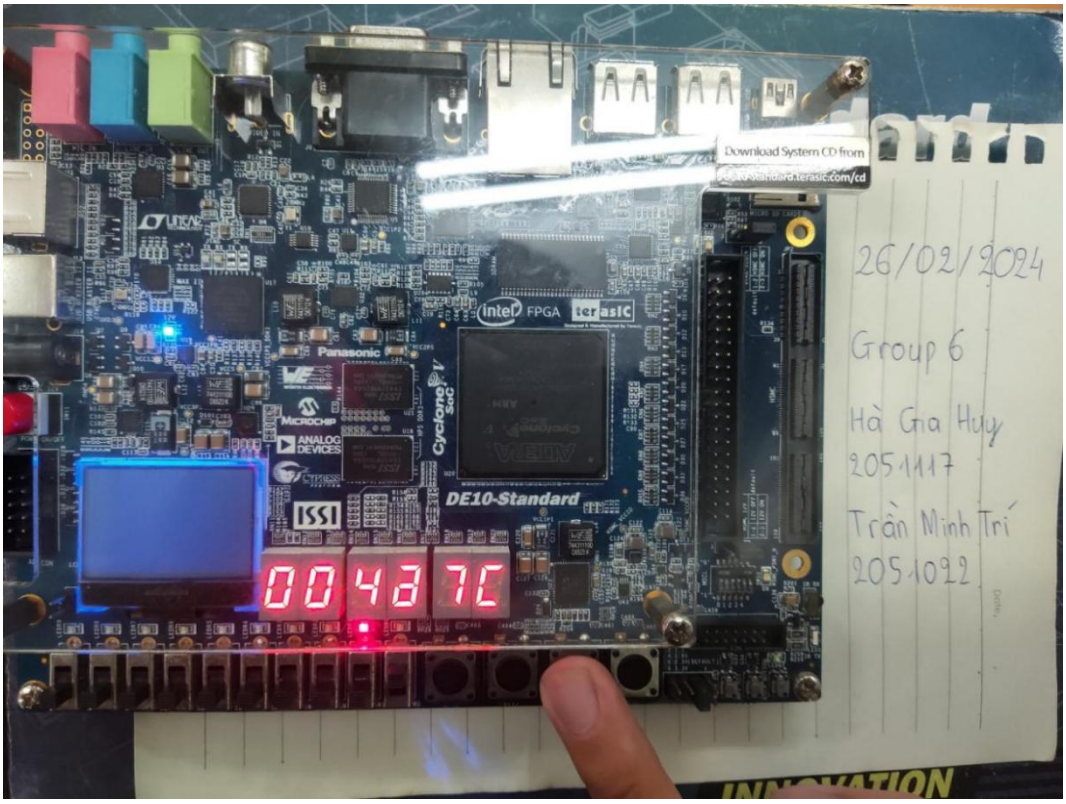


Figure 12. Read Data

VII. CONCLUSION

In our current work, we learn about memory control techniques for read and write operations, in particularly FIFO structure, as well as the creation of multiple clocks to support asynchronous designs. In addition, we are utilizing the Altera IP Catalog Memory as demonstrated in Lab 2. We have reviewed how to implement these memories and incorporate the megafunction files necessary for simulation.

The assert macro represents a potent feature within SystemVerilog HVL (Hardware Verification Language). This functionality empowers programmers to stipulate assertions and affirm the correct implementation of their code. Presently, it holds widespread adoption within most design verification projects, rendering it advantageous to acquire proficiency in its application. Perhaps one day we will transition into roles as design verification engineers, where the ability to write numerous assertions will be essential. Therefore, it would be advantageous to start learning and practicing assertion usage now.

VIII. REFERENCES

1. Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide

access from: https://cdrdv2-public.intel.com/667041/ug_ram_rom-683240-667041.pdf

2. FIFO (computing and electronics)

access from: https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)

3. Frequency divider

access from: https://en.wikipedia.org/wiki/Frequency_divider

4. Internal Memory (RAM and ROM) User Guide

access from: https://www.intel.com/programmable/technical-pdfs/654378.pdf

5. Linear-feedback shift register

access from: https://en.wikipedia.org/wiki/Linear-feedback_shift_regi*ster*

6. SystemVerilog for Verification: A Guide to Learning the Testbench Language Features

access from: https://link.springer.com/book/10.1007/978-1-4614-0715-7