# ENEL373
# Reaction Timer Report

*Monday Group10*
*Date: 16 May 2025*

Xiaoping Ma – xma45

Harvey Ulzii – kul12

Conner Horn - cho183

# 1. Introduction

This report details the design and implementation of a reaction-speed-timer-program using a Nexys-4 DDR FPGA development kit. Using this setup and VHDL code, a bitstream output file is generated, displayed and interacted with via the LEDs, seven-segment displays, buttons and switches on the Nexys-4 DDR board. This allows for the reaction speed of the user to be tested and displayed.

The assignment called for the timer to meet a set of core specifications. Upon initialization of the program, the user should be prompted by three lit decimal points that sequentially turnoff. This provides a “count down” to start the timer. The time between decimal points turning off has been randomized to aid in accurate testing and prevent preemptive reaction times. After the countdown has ended and the timer has begun, when BTNC (Center Button) is pushed, the counter stops, and the reaction time is stored and displayed. Pressing BTNC again clears the display and reinitializes the countdown process. If a button press is detected before the countdown has completed, the program detects an error which is then displayed for the user to see. While the program is idle (neither the countdown nor timer are active) the program can access the results from the tests in various ways via different buttons being pressed. BTNR (Right Button) displays the average, BTNL (Left Button) clears the stored data, BTNU (Up Button) displays the longest (worst) reaction time, BTND (Down Button) displays the shortest (best) reaction time.

# 2. Design

## 2.1. Top Level Diagram

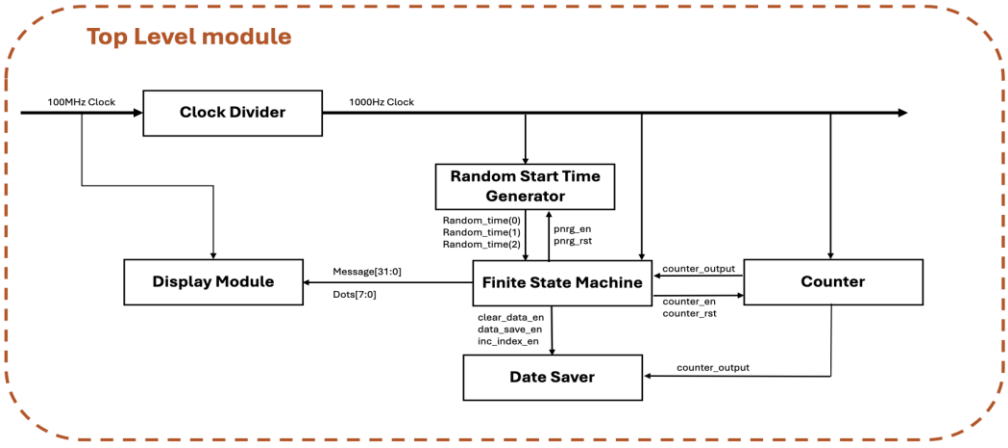


**Figure 1:** top level diagram.

## 2.2. Module Design Summary

The design presented in this report is structured into several functional modules, as illustrated in **Figure 1**. To promote cohesion and maintainability, a modular design approach was adopted. This modularity enhanced the clarity of the system architecture and facilitated easier debugging and verification of individual components.

### Clock Divider

A critical component of the project is the design of the counter. The 100 MHz clock provided by the Nexys board operates at a much higher frequency than required for the reaction timer, which needs a minimum display resolution of milliseconds. To achieve this, a clock divider is implemented to reduce the 100 MHz clock to a 1 kHz clock. This is done by accumulating and counting the fast clock pulses until the count reaches 999 (which is equivalent to dividing by 1000). As a result, the fast clock is effectively divided by a factor of 1000, producing a 1 kHz signal suitable for the reaction timer.

### Display Module

As for the display module, the important part for user interaction and feedback. It required input information to decode from a 4-bit binary array into a 7-bit binary array, with the 7-bit array indicating the Cathodes that shows the number and there is an extra bit that is the Cathode for the decimal point. The messages are assigned to each of the seven segments. Each of the 8 individual cathodes share a common anode. A display selector depending on the fast clock can turn on each of the seven segments sequentially in a short amount of time and display the information.

### Counter

A counter is used to track the reaction time. While a simple accumulator can be used to count the reaction time, it requires the numbers to be converted back to 4 four bit binary array so that the display module can use them for displaying. One 4 bit binary number represents the decimal number from 0 to 9, in which “0000” to “1001”. After the less significant digit reaches “1001”, the next digit should increment by one and reset the current digit to “0000”. Using a counter that counts from “0000” to “1001” following the clock signal and resetting the number to “0000” on the next clock signal along with increment onto the next digit as in **Appendix A figure1** is helpful when displaying the number at the seven segments.

### Data Saver

In Milestone 1, the 16-bit binary array was only stored within the counter. Once the finite state machine (FSM) restarted the counting process, the previously stored data would be replaced.

To address this, as shown in **Appendix A Figure 2**, a circular buffer was implemented to store the data. A ram_type array with a size of 3 was created, where each element stores a 16-bit binary array. When the number of samples exceeds 3, the newest sample will overwrite the oldest one.

This ensures that the maximum, minimum, and average calculations are always performed on the most recent 3 samples, providing a rolling window of data for analysis.

### Arithmetic logic unit

The data stored in ram consistently includes three samples, even if fewer than three tests have been conducted. Upon storing the first test data, it is written to the next two data slots of the most recent test until this repeat data is overwritten by the latest reaction test.

## 2.3. Finite State Machine

At the core of the design is a **finite state machine (FSM)**, which governs the behavior of the system by determining outputs based on the current state and inputs. It is composed of multiple states that manage the control flow of the system based on user interaction and internal timing logic. These states are interconnected in a way that enforces a linear and logical progression through the reaction timer cycle. The key states and their interactions are as follows:

- **random_gen**: Initializes three pseudo-random countdown delays before transitioning to warning_3.
- **warning_3, warning_2, warning_1**: Countdown phases, each turning off one decimal point. Pressing BTNC early triggers error state.
- **counting**: Starts the timer; pressing BTNC stops it and moves to printing.
- **printing**: Displays the recorded time. This display can be toggled via the previously stated buttons and their intended uses.
- **printing_Avg / Max / Min**: Show calculated values using ALU; BTNC returns to printing.
- **clear**: Resets stored times, then returns to random_gen.
- **error**: Triggered by early BTNC press during countdown; BTNC returns to warning_3.

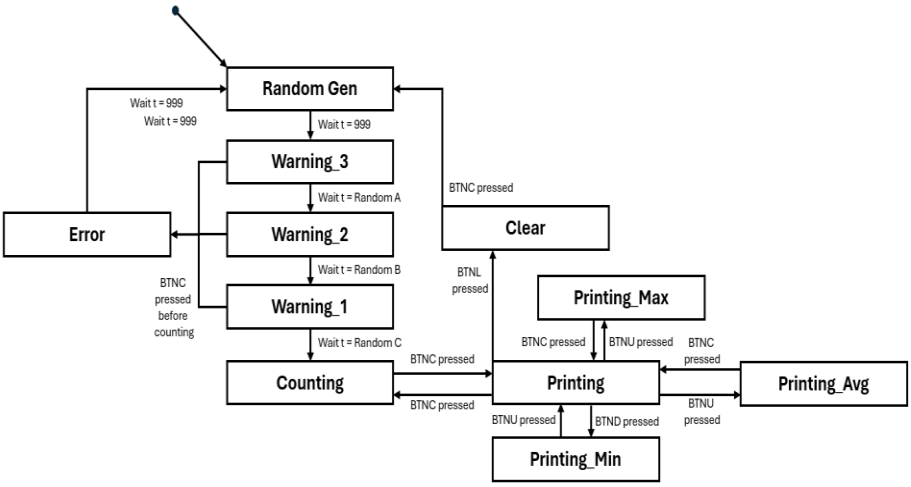


**Figure 2**: FSM diagram.

The design process started by creating a state transition diagram to show how the system moves between states based on user inputs or timing conditions. This helped simplify the VHDL implementation by clearly linking each state to its outputs and transitions.

For example, the FSM moves from the counting state to the printing state only when BTNC is pressed, which stops the counter and displays the reaction time. In the printing state, BTNU shows the maximum time, BTNR shows the average, and BTND shows the minimum. These actions only work when the system is idle and not running a test.

The FSM takes inputs such as the 1 kHz clock, five push buttons (BTNC, BTNU, BTND, BTNR, BTNL), switch SW[0], the current reaction time from the counter, and three random timing values. Based on these inputs, it controls outputs to manage the counter, pseudo-random generator, and data storage. It also handles arithmetic operations (average, min, max) and generates the display output.

## 2.4. Pseudo Random Number Generator

For the extension task, our group chose to implement a pseudo-random number generator (PRNG). The purpose of the PRNG was to create a random time interval between the decimal points turning off at the beginning of the reaction time test. This meant the user was unable to predict when the final decimal point would disappear, making the reaction time test more authentic.

An 8-bit liner feedback shift register was used as core of the PNRG as shown in Appendix B, figure #. The LFSR produces a pseudo-random sequence by shifting and feeding back a combination of selected bits. The output of the LFSR is scaled and converted to generate three random delay times, each ranging between 1000ms and 2900ms. These times are generated during the random_gen state before warning_3 state.

# 3. Module Testing

To ensure each component worked correctly before combining them into the full system, testbenches were created for important modules (see **Appendix B, Figure 1**). The ALU, which calculates the average of the last three reaction times, was carefully tested using known input values. For example, when given the BCD-coded values 1111, 2222, and 3333, the ALU correctly output 2222. This confirmed the ALU's BCD handling and averaging functions worked as expected.

For instance, in one test case, the values 1111, 2222, and 3333 (all in 16-bit BCD format) were input to the ALU. The expected average of these values was 2222, which the ALU produced correctly. This confirmed that the module's decomposition of BCD digits and arithmetic averaging logic worked as intended.

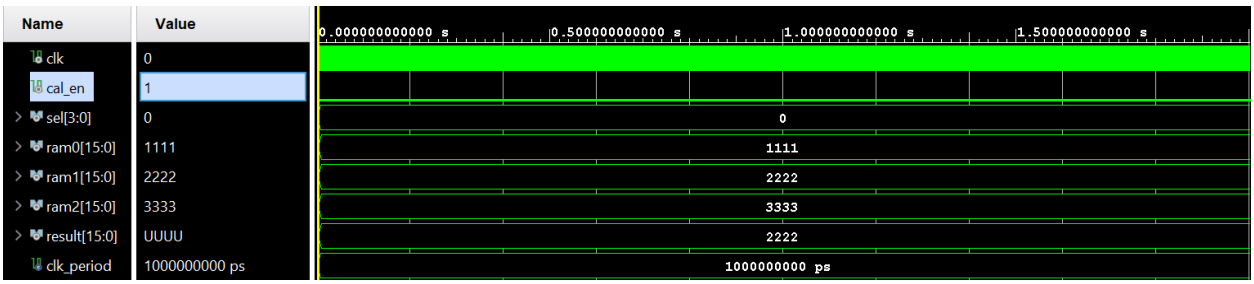


**Figure 4:** illustrates this successful test case, highlighting the cal_en signal being asserted and the corresponding result.

Simulations were also used for the clock divider and counter modules. The clock divider was validated to generate a 1 kHz signal from the 100 MHz input clock, critical for synchronizing human-readable updates on the 7-segment display. The counter was confirmed to be incremented in BCD format and rolled over correctly.

Ultimately, these testbenches provided early validation that minimized integration issues later in the design process. They also helped identify subtle bugs—such as the RAM module not properly initializing—which initially led to unexpected ALU outputs. Fixes were made to ensure all stored data were reset to zero upon system clear.

# 4. Problems Encountered

Throughout the development of the reaction timer project, several challenges were encountered.

**Clock Frequency Mismatch:** The provided 100 MHz system clock was too fast for the required millisecond-level timing. This required careful design of a reliable clock divider to ensure accurate display time and reaction measurements.

As shown in **Appendix C, Figure 1,** the signal slow_clk_cnt was intended to reset when it reached "11000011010100000" ticks, which corresponds to 0.5 seconds at a 100 MHz clock. This was designed so that the divided clock would have a period of 1 second, toggling on every rising edge.

However, the initially selected count value was incorrect. This mistake was only discovered later during testing, as the generated clock was noticeably slower than expected. The issue was caused by miscalculating the required tick count for a 0.5-second interval.

**Random Start Time Stability:** During the integration of the PRNG module, it was observed that the timer occasionally stopped at a state before entering the counting state. This issue was caused by an incorrect clock counting limit for variable t, as shown in **Appendix C, Figure 2**.

# 5. Conclusion

The reaction timer project met the specified functionality requirements and demonstrated effective use of modular VHDL design. Key modules such as the FSM, ALU, counter, and display controller operated cohesively to deliver an interactive and responsive system. The addition of a pseudo-random countdown improved the authenticity of the reaction test, while robust state handling ensured system reliability.

However, some issues were encountered because certain modules were not thoroughly tested before integration. Additionally, the signal assignment and code organization were not done clearly and well enough. Since the signals were not carefully added to the sensitivity list even though sometimes it does not influence the output. These problems slow down the development progress of the project. A structured approach and the addition of detailed comments related to the program can be helpful to these problems and would aid in improving the project development.

# 6. Appendices

## Appendix A

```
process(clk, counter_rst, counter_en)
begin
    if counter_rst = '1' then
        c1 <=(others => '0');
        c2 <=(others => '0');
        c3 <=(others => '0');
        c4 <=(others => '0');
    elsif counter_en = '1' and rising_edge(clk) then
        c1 <= n1;
        c2 <= n2;
        c3 <= n3;
        c4 <= n4;
    end if;
end process;
n4 <= "0000" when (c4 = "1001" and c3 = "1001" and c2 = "1001" and c1 = "1001") else
      c4 + 1 when (c3 = "1001" and c2 = "1001" and c1 = "1001") else
      c4;
n3 <= "0000" when (c3 = "1001" and c2 = "1001" and c1 = "1001") else
      c3 + 1 when (c2 = "1001" and c1 = "1001") else
      c3;
n2 <= "0000" when (c2 = "1001" and c1 = "1001") else
      c2 + 1 when (c1 = "1001") else
      c2;
n1 <= "0000" when (c1 = "1001") else
      c1 + 1;

counter_output <= c4 & c3 & c2 & c1;
```

**Figure 1:** Clock divider module.

```
architecture Behavioral of data_saver is
    type ram_type is array(2 downto 0) of std_logic_vector(15 downto 0);
    signal RAM: ram_type;
    signal write_index: integer range 0 to 2:= 2;
begin
    DATA_SAVING: process(clk, write_index, clear_data_en, data_save_en, inc_index_en, counter_output)
    begin
    if rising_edge(clk) then
        if (data_save_en = '1') then
            if (RAM(0) = "0000000000000000") then
                RAM(0) <= counter_output;
                RAM(1) <= counter_output;
                RAM(2) <= counter_output;
            else
                RAM(write_index) <= counter_output;
            end if;
        elsif (inc_index_en = '1') then
            write_index <= (write_index + 1) mod 3;
        elsif (clear_data_en = '1') then
            RAM <= (others => (others => '0'));
            write_index <= 2;
        end if;
    end if;
    end process;
    max <= RAM(0) when (RAM(0) >= RAM(1) and RAM(0) >= RAM(2)) else
           RAM(1) when (RAM(1) >= RAM(2)) else
           RAM(2);

    min <= RAM(0) when (RAM(0) <= RAM(1) and RAM(0) <= RAM(2)) else
           RAM(1) when (RAM(1) <= RAM(2)) else
           RAM(2);

    ram0 <= RAM(0);
    ram1 <= RAM(1);
    ram2 <= RAM(2);

end Behavioral;
```

**Figure 2:** Data saver.

## Appendix B

```
process(CLK100MHZ)
begin
    if (rising_edge(CLK100MHZ)) then
        if slow_clk_cnt >= "11000011010100000" then
            slow_clk_cnt <= (others => '0');
        else
            slow_clk_cnt <= slow_clk_cnt + 1;
        end if;
    end if;
end process;
CLK1000HZ <= '1' when slow_clk_cnt = "11000011010100000" else '0';
```

**Figure 1:** Clock divider module.

## Appendix c

```
stim_proc: process
begin
    wait for 2 * clk_period;

    -- Test case: inputs = 1111, 2222, 3333
    ram0 <= "0001000100010001";
    ram1 <= "0010001000100010";
    ram2 <= "0011001100110011";
    sel <= "0000";
    cal_en <= '1';

    wait for clk_period;
    cal_en <= '0';

    wait for 2 * clk_period;

    wait;
end process;
```

**Figure 1:** ALU testbench