UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

ENEL373: Digital Electronics and Devices

University of Canterbury

Development of a Reaction Timer in VHDL

Philip Brand *(15776928)*
Michael Brown *(48571923)*
Boston Black *(24668421)*

May 11, 2025

Contents

Contents . . . . . **i**

1 Introduction **1**

2 Design Summary **1**

3 Expanded Design Summary **2**

3.1 Design Methods 2

3.1.1 Diagramming 2

3.1.2 Naming and Syntax Conventions 2

3.1.3 Component Reuse 3

3.1.4 Programming Techniques 3

3.2 Design Justification 3

4 Module Testing **4**

5 Design & Implementation Problems **5**

6 Conclusions **6**

7 References **7**

8 Appendix A: Code Listings **8**

9 Appendix B: Testbench & Waveforms **10**

1 Introduction

Human reaction time is the interval between a stimulus, and the response to that stimulus. There are two main types of reaction time; simple, and choice. Simple reaction involves reacting to a singular stimulus, while choice reaction involves choosing the correct response from multiple stimuli. This project aims to implement a simple reaction time recorder on an FPGA in VHDL.

The reaction stimulus test starts with three LEDs that turn off sequentially, with a random delay between each LED turning off. Once the last LED turns off, the user must press a button, and the delay is displayed in milliseconds on 8 7-segment displays. The FPGA stores the reaction delays from the last three trials, and can display the fastest, slowest, and average reaction times. These values can be reset by the user.

2 Design Summary

The FPGA reaction stimulus test depends on four major components, those being the 8-digit counter, the ALU, the reaction countdown, and the output select and override. These components are shown in Figure 1. Each of these components are enabled and disabled via a finite state machine.

Figure 1: Overview of VHDL reaction timer structure

An 8-digit 5-bit-BCD counter is the primary mechanism used to track the reaction time from each stimulus test to the nearest millisecond. Each BCD digit is passed into an 8x5-to-5 mux. The purpose of this mux

is to sync the selected output digit with the currently active seven segment display anode, outputting each digit onto its associated display.

All the BCD digits from the 8-digit counter are passed through the bcd_8_to_binary module which converts the BCD numbers into a 28-bit binary number. This number is then passed into a circular buffer which is read by the ALU to calculate the average, minimum, and maximum of the last three reaction times. These times are converted back into BCD using the double dabble algorithm [1]. Another 8x5-to-5 mux is used to select which digit is output to the display.

A linear shift register is used to provide some randomisation for the countdown sequence at the start of every reaction stimulus test. This time randomisation acts as the trigger for the countdown state, Dotiey, and also has an 8x5-to-5 mux for digit selection.

All three 8x5-to-5 muxes from the three components described above are piped into an 8x5-to-5 mux used for output selection. This mux selects which module to take output from depending on what state is active. The mux output is piped into a text override, which overwrites some of the digits with letters based on state. This is finally piped into a 7 segment decoder, which takes a 5-bit BCD digit and converts it to a 8-bit seven segment display light combination which is piped to hardware.

3 Expanded Design Summary

3.1 Design Methods

The primary focus for the reaction timer project project was to develop VHDL code that is easy to maintain and further develop. This led to a heavy focus on modular design and code reuse. Since the project was developed as a group, the other focus was for components to be developed and tested individually, and then put together with minimal issues.

3.1.1 Diagramming

The primary program development method used in this project was to draw program or component flow diagrams before VHDL was written. These digrams detailed the communication between between individual modules, and outlined the overall program structure. Figure 1 shows one such digram outlining each major component in the design and how they're grouped into larger modules.

Clear communication between group members of the expected inputs and outputs from each component played a significant role in keeping the final assembly simple. Examples include the consistent transfer of characters to be displayed by using 8x5-to-5 muxes to transmit one character at a time unless otherwise required, such as for the ALU.

3.1.2 Naming and Syntax Conventions

Another important aspect of the project ensuring high code quality, readability, and maintainability was strict rules around naming and syntax. This included consistent whitespace, indentation, case, and naming style.

For example, all signals within an entity were postfixed with an underscore and whether that signal was an input or output. This can be seen in Listing 1 in Appendix A. This ensured when the components were used in the top-level Behavioural architecture with port mapping, it was easy to see which signals were inputs and outputs.

To make the difference between inputs/outputs and internal signals clear, input/output names were in all capitals while internal signals were in all lowercase. the signal types were also capatilised for inputs/outputs eg. STD_LOGIC instead of std_logic. This can be seen in Appendix B.

When naming modules, an effort was made to name them in a way that described the capabilities of the module. For example, timer_8_num_selectable has 8 numbers and one can be selected. The postfix _xb was used to indicate a module had an x bit input/output. This is used in modules such as counter_3b and decoder_3b.

3.1.3 Component Reuse

In order to reduce the number of components that needed to be created and maintained, components were reused where possible.

The most versitile example of this was the 8 channel 5 bit mux shown in Appendix 3. The mux took up to 8 5-bit BCD inputs, and selected one of those to be the output. The select lines and number of channels for the mux corresponded with the segment display, allowing both to be controlled with the same signal. This capability made it easy to use for displaying timer digits and error message letters. The same mux was also used to select what output to send through to the display depending on the state. While not all the channels of the mux were used in all implementations, it stil made sense to use the same component as it was proven to work, and integrated well with the rest of the system.

Another module that was reused a lot was clk_divider. Having an input to set the upperbound before toggling the slower clock signal allowed it to be reused for a wide range of output clock signals. This input allowed it to output a variable speed timer for a pseudorandom dot delay as the upperbound could be changed resulting in different clock speeds and therefore delays.

3.1.4 Programming Techniques

Another method used to implement the FPGA design was pair programming. This had to be done with care, as since the project was done in three-person groups, it would have been easy to leave one person out of the process resulting in them lacking undersstanding of the code. However, pair programming did help ensure that minimal time was spent debugging, as the observer frequently caught subtleties of VHDL that the programmer missed. Pair programming was used to develop the circular buffer, counters, and FSM.

Since the project was significantly modularised, it was important that individual components could be developed and tested without hindering the development of other components. The method used to achieve this was to use git branching and merging. Individual components were developed on separate branches, tested, and then merged into the main branch. This ensured that the main branch was always stable, and that broken changes could be easily rolled back.

3.2 Design Justification

The design of the reaction stimulus test is intended to allow ease of modification or expansion. Keeping modularisation as a high priority during component design allowed each module to view the other modules as a “black box”, interacting only through standardised interfaces. These interfaces were typically a single 5-bit encoded number or letter based on BCD, and were controlled by the generalised 8x5-to-5 mux shown in Appendix 3 instead of custom muxes for each output. This further reduced the number of different components required. The only exception to this was the output of all counted BCD digits to the BCD-to-binary convertor as it operated on all digits at once, as shown in Appendix 4.

A second benefit of each component being a "black box" is that as most components interacted only through standardised interfaces, how the components operated behind the scenes was irrelevant. This allowed for significant changes to how a module completed a task without disturbing inter-module communication.

The usage the 8x5-to-5 mux shown in Appendix 5 to select between outputs allows for addition of states as desired by utilizing more mux inputs, with output select lines controlled by the finite state machine. If more states were to be added, then additional muxes can be linked up for more input lines. This shows clear capability for design expansion.

A selectively enabled text override was used to display text in the first two seven-segment displays as it centralised the control of text display. This additionally simplified the other modules as it limited their design to the numeric part of the code.

4 Module Testing

During development of the circular buffer, ALU, and associated data-type converters used to calculate the user reaction time statistics, the output numbers were too small. For example a minimum reaction time of 246 milliseconds would be displayed as 91 milliseconds. Since the delay would be displayed correctly immediate after a test, the problem lay somewhere within the blue block in Figure 1. An assumption was made that the BCD-to-binary converter would be fault-free, since the logic it required was simple. To determine in which module the problem lay, a testbench comprising of the circular buffer, ALU, and binary-to-BCD converter was created. The testbench simulated three write operations to the circular buffer, an ALU operation select, and a trigger for the binary to BCD double-dabble algorithm, followed by reset signal to the circular buffer and binary-to-BCD converter. The testbench stimulus VHDL is in Listing 6 in Appendix x.

The major values output by the testbench are the circular buffer contents, the ALU operation select and ALU output, and the binary-to-BCD converted output. These waveforms, as well as the other waveforms such as the clock, circular buffer write, and resets can be seen in Figure 2.

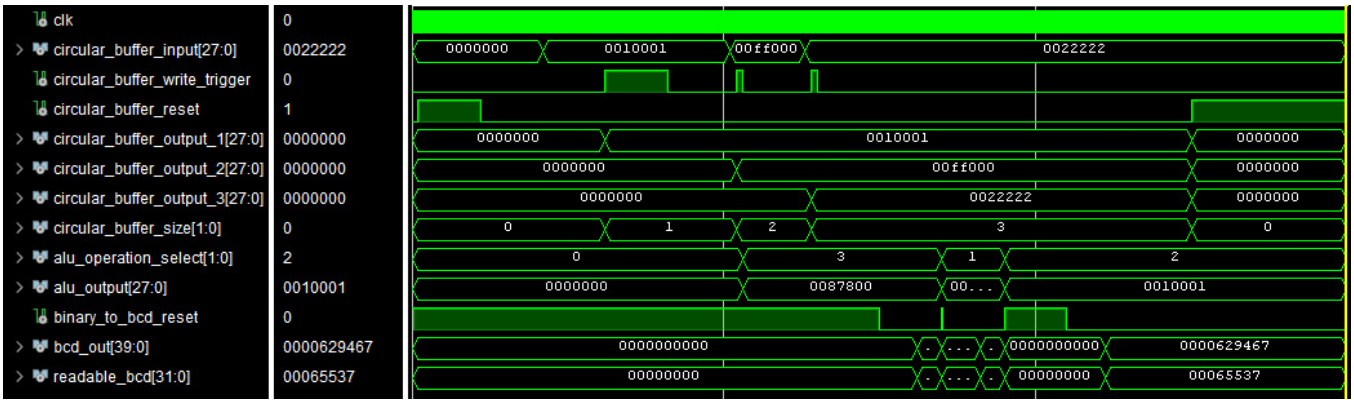

Figure 2: Reaction statistic calculation testbench.

The first 750 ns of testbench simulation show the three write operations to the circular buffer. The three circular buffer outputs change exactly as expected, with each write operation setting a new value. The ALU operation select is 3 at 700 ns, and the ALU output is 0087000, which is the correct hexadecimal representation of the average of the three values in the circular buffer. (CHECK THAT). The ALU operation is then set to 1, and the ALU outputs the correct maximum time of 00ff000. The final ALU operation is 2, and the ALU outputs the correct minimum time of 0010001. The binary-to-BCD converter

is then triggered, and the BCD output is 00065537, which the correct BCD representation of the 0010001 in hexadecimal.

These waveforms indicated that the problem did not lie where expected, and must have been with the BCD-to-binary converter originally thought of as fault-free. Another testbench was created to test that converter, and, as expected, the converter did not convert correctly. This was rectified by modifying the changing the multiplication factors of 10 from hexadecimal to binary. The fixed testbench waveform can be seen in Appendix something in Figure 3.

The waveform for the fixed BCD-to-binary converter can be seen in Figure 3.

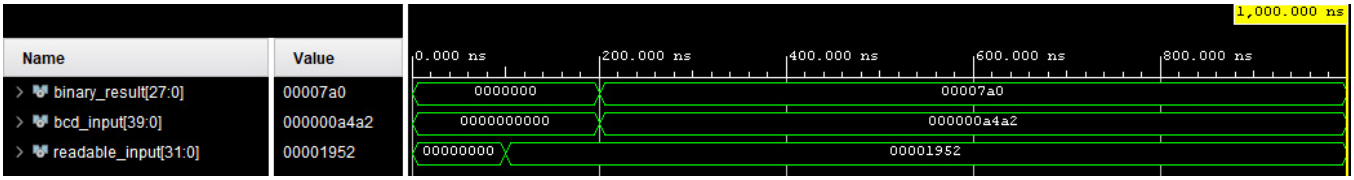

Figure 3: Corrected BCD to binary converter testbench.

5 Design & Implementation Problems

One issue that arose during the development of the seven segment display module which mapped a BCD input to a combination of segments on a display to represent that BCD number was that the provided mappings did not represent the numbers correctly. The solution approach was to first determine whether the BCD input or mapping was incorrect. This was done by tying the BCD input to the green LEDs on the FPGA. This asserted BCD input was correct. To rectify the incorrect mappings, individual segments on a display were tied to the FPGA user switches. Combinations of switches were tested, and when the segments represented a number, the combination of switches was recorded as a BCD mapping.

However, once all the mappings were implemented, we found that our countdown timer failed to display the dots. Alteration of the countdown timer to display temporary digits instead of dots revealed the countdown to be functional. Stepping through the code by hand revealed that, due to the use of the unfamiliar provided seven segment decoder, a line of code was overlooked that overwrote the decimal place value. Disabling this line of code allowed the countdown timer to show dots.

A significantly unexpected issue that appeared during the testing of the finite state machine was deemed as the “schrodinger’s States” problem. This originated from the FSM running lines of code designed to change states even while they were commented out. To narrow down the source of the bug, the FSM was reduced to only what was necessary to run a simple reaction timer test without any ALU or error message. From there, states were slowly added back into the FSM and were tied to LEDs that would light up when the new states were activated. This showed additional issues in which an LED would trigger, but the state change wouldn’t complete. With the suspicion that a race condition was in play,the issue was fixed by removing the FSM inputs from the sensitivity list of the FSM process, instead tying it to a 10 kHz clock.

There were issues with the simulation software built into the Vivado toolchain. During the design and subsequent simulation of the BCD-to-Binary numeric converter, the Vivado simulator would stop the simulation at the beginning of the numeric conversion, but would not give an error message to indicate an issue. The issue was narrowed down to being a problem with the simulator itself when a bitstream was generated for the whole system so that the converter could be tested as part of the whole, and no issue was found.

Another issue that arose was regarding case sensitivity in VHDL. Although VHDL is not case sensitive, the constraints file is. Early on during the project, while developing the clock divider, an error message

"insert error message here" was encountered. The approach to solving this problem was to ask the TAs in the lab what the message meant. However, they were unable to explain the meaning. From there, the clock divider code we wrote was thoroughly compared to the provided clock divider code, and every small difference tested to see if it repaired the error. This determined the issue to be the case of the clock input declared in entity of main module, as lowercase had been used rather than uppercase. This correct case can be seen in Listing 2.

6 Conclusions

When developing the reaction timer, an number of problems were encountered. One of these involved the FSM changing states unexpectedly. To find the cause of this problem, the FSM was simplified to a minimal working version, then slowly expanded. Another problem where the display module did not show the correct numbers was solved by creeating a small program mapping switches to the BCD inputs of the display module. Both of these problems were solved by reducing the scope of the program allowing the problem to be found without confusing things with extra complexity. This method, as well as the use of testbenches was an effective way of finding problems in VHDL code.

Modularity was a major strength allowing new features to be integrrated easily with the rest of the project, and making it easy to make changes to a module without breaking everything else. The modularity also reduced the effort needed to make changes such as increasing the number of bits in the BCD signals as there are less seperate modules in which the change needs to be made.

The project could be expanded and improved by implementing the ALU in such a way that it could take an arbitrary amount of numbers as input then calculate the average, max, and min. Currently it is harcoded to only accept three numbers and each of the input number needs its own signal going into the ALU module. this doesn't really fit with the modularity of the rest of the project as the ALU would need significant changes to work with more than three numbers.

And they all lived happily ever after, and VHDL never bothered them again.

The End

7 References

[1] Support Team. “Vhdl code for binary to bcd converter. ”[Online]. Available: https://allaboutfpga.com/vhdl-code-for-binary-to-bcd-converter/.

8 Appendix A: Code Listings

```
-- Define module IO
entity counter_decade is
    Port ( EN_IN : in STD_LOGIC;
           RESET_IN : in STD_LOGIC;
           INCREMENT_IN : in STD_LOGIC;
           COUNT_OUT : out STD_LOGIC_VECTOR (3 downto 0);
           TICK_OUT : out STD_LOGIC);
end counter_decade;
```

Code Listing 1: Entity naming convention example.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity main is
    Port ( CLK100MHZ : in STD_LOGIC; -- This must be capitalised.
           -- This, and the rest, must be too.
           AN : out STD_LOGIC_VECTOR (7 downto 0) := X"00";
           SEVEN_SEG : out STD_LOGIC_VECTOR (7 downto 0) := X"00";
           BTNC : in STD_LOGIC;
           BTNR : in STD_LOGIC;
           BTNL : in STD_LOGIC;
           BTNU : in STD_LOGIC;
           BTND : in STD_LOGIC);
end main;
```

Code Listing 2: Capitalised port names to avoid errors.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity multiplexer_8_1_4b is
    Port ( MUX_IN_0 : in STD_LOGIC_VECTOR (4 downto 0);
           MUX_IN_1 : in STD_LOGIC_VECTOR (4 downto 0);
           MUX_IN_2 : in STD_LOGIC_VECTOR (4 downto 0);
           MUX_IN_3 : in STD_LOGIC_VECTOR (4 downto 0);
           MUX_IN_4 : in STD_LOGIC_VECTOR (4 downto 0);
           MUX_IN_5 : in STD_LOGIC_VECTOR (4 downto 0);
           MUX_IN_6 : in STD_LOGIC_VECTOR (4 downto 0);
           MUX_IN_7 : in STD_LOGIC_VECTOR (4 downto 0);
           SELECT_IN : in STD_LOGIC_VECTOR (2 downto 0);
           MUX_OUT : out STD_LOGIC_VECTOR (4 downto 0));
end multiplexer_8_1_4b;
```

```
architecture Behavioral of multiplexer_8_1_4b is
begin
    process (SELECT_IN, MUX_IN_0, MUX_IN_1, MUX_IN_2, MUX_IN_3, MUX_IN_4, MUX_IN_5, MUX_IN_6,
    begin
        case(SELECT_IN) is
            when "000" => MUX_OUT <= MUX_IN_0;
            when "001" => MUX_OUT <= MUX_IN_1;
            when "010" => MUX_OUT <= MUX_IN_2;
            when "011" => MUX_OUT <= MUX_IN_3;
            when "100" => MUX_OUT <= MUX_IN_4;
            when "101" => MUX_OUT <= MUX_IN_5;
            when "110" => MUX_OUT <= MUX_IN_6;
            when "111" => MUX_OUT <= MUX_IN_7;
        end case;
    end process;
end Behavioral;
```

Code Listing 3: Generalised 8x5-to-5 mux.

```
entity bcd_8_to_binary is
    Port ( BCD_BUS_IN : in STD_LOGIC_VECTOR (39 downto 0);
           BINARY_OUT : out STD_LOGIC_VECTOR (27 downto 0));
end bcd_8_to_binary;
```

Code Listing 4: BCD to binary converter entity defintion.

```
ff11: multiplexer_8_1_4b port map (MUX_IN_0 => encoded_reaction_time_digit,
                                   MUX_IN_1 => selected_alu_bcd_digit,
                                   MUX_IN_2 => selected_alu_bcd_digit,
                                   MUX_IN_3 => selected_alu_bcd_digit,
                                   MUX_IN_4 => encoded_error_text,
                                   MUX_IN_5 => encoded_display_placeholder,
                                   MUX_IN_6 => encoded_display_placeholder,
                                   MUX_IN_7 => encoded_dots,
                                   SELECT_IN => encoded_display_input_select,
                                   MUX_OUT => encoded_segment_data);
```

Code Listing 5: Component instantiation of output select 8x5-to-5 mux.

9 Appendix B: Testbench & Waveforms

```
----------------------------------------------------------------------------------
-- Engineers: Michael Brown, Philip Brand
-- Create Date: 25.04.2025 02:04:31
-- Module Name: alu_setup_tb - Behavioral
-- Project Name: ALU Setup Test Bench
-- Description: Tests the alu, circular_buffer, and binary_to_bcd_8 components.
-- Additional Comments: Used to test the more complex parts of the
-- reaction statistic calculations.
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity alu_setup_tb is
--  Port ( );
end alu_setup_tb;

architecture Behavioral of alu_setup_tb is
    signal circular_buffer_output_1 : std_logic_vector (27 downto 0) := (others => '0');
    signal circular_buffer_output_2 : std_logic_vector (27 downto 0) := (others => '0');
    signal circular_buffer_output_3 : std_logic_vector (27 downto 0) := (others => '0');
    signal circular_buffer_input : std_logic_vector (27 downto 0) := (others => '0');
    signal circular_buffer_size : std_logic_vector (1 downto 0) := (others => '0');
    signal alu_operation_select : std_logic_vector (1 downto 0) := (others => '0');
    signal circular_buffer_write_trigger : std_logic := '0';
    signal circular_buffer_reset : std_logic := '0';
    signal clk : std_logic := '0';
    signal binary_to_bcd_reset : std_logic := '0';
    signal bcd_out : std_logic_vector (39 downto 0) := (others => '0');
    signal bcd_out_expanded : std_logic_vector (31 downto 0) := (others => '0');
    signal alu_output : std_logic_vector (27 downto 0) := (others => '0');

    component alu is
        Port ( NUM_1_IN, NUM_2_IN, NUM_3_IN : in STD_LOGIC_VECTOR (27 downto 0);
               BUFFER_SIZE_IN, OPERATION_SELECT_IN : in STD_LOGIC_VECTOR (1 downto 0);
               OUTPUT_OUT : out STD_LOGIC_VECTOR (27 downto 0));
    end component alu;

    component circular_buffer is
        Port ( NUMBER_IN : in STD_LOGIC_VECTOR (27 downto 0);
               NUMBER_1_OUT, NUMBER_2_OUT, NUMBER_3_OUT : out STD_LOGIC_VECTOR (27 downto 0);
               BUFFER_SIZE_OUT : out STD_LOGIC_VECTOR (1 downto 0);
               RESET_IN, WRITE_TRIGGER_IN : in STD_LOGIC);
    end component circular_buffer;

    component binary_to_bcd_8 is
```

```
        Port ( CLK_IN : IN  std_logic;
               RESET_IN : IN  std_logic;
               BINARY_IN : IN  std_logic_vector(27 downto 0);
               BCD_8_DIGIT_OUT : OUT std_logic_vector (39 downto 0) := (others => '0'));
    end component binary_to_bcd_8;

begin
    ff0: alu port map ( NUM_1_IN => circular_buffer_output_1,
                        NUM_2_IN => circular_buffer_output_2,
                        NUM_3_IN => circular_buffer_output_3,
                        BUFFER_SIZE_IN => circular_buffer_size,
                        OPERATION_SELECT_IN => alu_operation_select,
                        OUTPUT_OUT => alu_output);

    ff1: circular_buffer port map ( NUMBER_IN => circular_buffer_input,
                                    NUMBER_1_OUT => circular_buffer_output_1,
                                    NUMBER_2_OUT => circular_buffer_output_2,
                                    NUMBER_3_OUT => circular_buffer_output_3,
                                    BUFFER_SIZE_OUT => circular_buffer_size,
                                    RESET_IN => circular_buffer_reset,
                                    WRITE_TRIGGER_IN => circular_buffer_write_trigger);

    ff2: binary_to_bcd_8 port map ( CLK_IN => clk,
                                    RESET_IN => binary_to_bcd_reset,
                                    BINARY_IN => alu_output,
                                    BCD_8_DIGIT_OUT => bcd_out);

    bcd_out_expanded(31 downto 28) <= bcd_out(38 downto 35);
    bcd_out_expanded(27 downto 24) <= bcd_out(33 downto 30);
    bcd_out_expanded(23 downto 20) <= bcd_out(28 downto 25);
    bcd_out_expanded(19 downto 16) <= bcd_out(23 downto 20);
    bcd_out_expanded(15 downto 12) <= bcd_out(18 downto 15);
    bcd_out_expanded(11 downto 8) <= bcd_out(13 downto 10);
    bcd_out_expanded(7 downto 4) <= bcd_out(8 downto 5);
    bcd_out_expanded(3 downto 0) <= bcd_out(3 downto 0);

    simulation_clk : process
    begin
        wait for 1ns;
        clk <= '1';
        wait for 1ns;
        clk <= '0';
    end process;

    simulation : process
    begin
        binary_to_bcd_reset <= '1';
        wait for 10ns;
        circular_buffer_reset <= '1';
```

```
        circular_buffer_input <= X"0000000";
        wait for 100ns;
        circular_buffer_reset <= '0';
        wait for 100ns;
        circular_buffer_input <= X"0010001";
        binary_to_bcd_reset <= '1';
        wait for 100ns;
        circular_buffer_write_trigger <= '1';
        wait for 100ns;
        circular_buffer_write_trigger <= '0';
        wait for 100ns;
        circular_buffer_input <= X"00FF000";
        wait for 10ns;
        circular_buffer_write_trigger <= '1';
        wait for 10ns;
        alu_operation_select <= "11";
        circular_buffer_write_trigger <= '0';
        wait for 100ns;
        circular_buffer_input <= X"0022222";
        wait for 10ns;
        circular_buffer_write_trigger <= '1';
        wait for 10ns;
        circular_buffer_write_trigger <= '0';
        wait for 100ns;
        binary_to_bcd_reset <= '0';
        wait for 100ns;
        binary_to_bcd_reset <= '1';
        alu_operation_select <= "01";
        wait for 1ns;
        binary_to_bcd_reset <= '0';
        wait for 100ns;
        binary_to_bcd_reset <= '1';
        alu_operation_select <= "10";
        wait for 100ns;
        binary_to_bcd_reset <= '0';
        wait for 200ns;
        circular_buffer_reset <= '1';
        wait for 100000ns;
    end process;
```

Code Listing 6: Testbench for ALU, circular buffer, and BCD to binary converter.

The waveform associated with the testbench in Listing 6 can be seen in Figure 4.

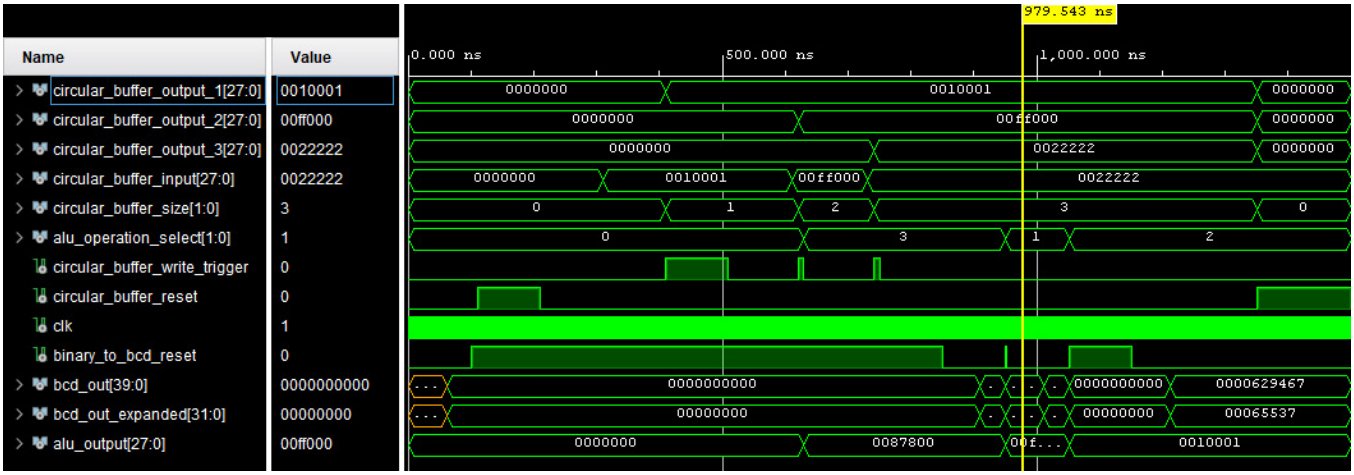

Figure 4: Reaction statistic calculation testbench.