ENGS 31: 21S

MasterMind

Final Project Report

MARSHALL, WALDEN
6-7-2021

Abstract

The goal of this project was to create a version of the game MasterMind which could be played on a Basys 3 board. MasterMind is a two player game in which one player, the codemaker, chooses a four digit code and the other player, the codebreaker, tries to guess the code. After each guess, the codemaker provides the codebreaker with feedback in terms of red and white pegs. In the digital version that was created, the Basys 3 Board performs the task of providing feedback based on the guess. The digital circuit used consists of three major components: a user interface, a datapath to compare the guess and code, and a controller to perform gameflow operations. The project ultimately never became fully functional on the Basys 3, though hardware validation and simulation results show that all components are functional expect for the 7 segment display on the FPGA.

Table Of Contents

Introduction

In this project, I attempted to build a version of the classic strategy game MasterMind that could be played on an the Basys 3 board. Using the buttons on the Basys 3 board, a user inputs a four digit guess. The guess is compared to the code and the board responds with LEDs to let the user know how many white and red pegs were earned.

Solution

Specifications

While there were certainly different levels of functionality that could have been implemented, there were certain bare minimum specifications that had to be met, and others that were not necessary but would improve the overall quality of the game.

At a bare minimum, the game had to be able to correctly provide feedback for a guess, given some code. This meant that it had to be able to provide some form of output on the Basys board that tells the user how many red and white pegs correspond to a guess. Additionally, it had to have some form of user interface that allowed the user to enter a guess.

Among other less critical specifications that were implemented in the final design was a form of gameflow logic to be able to be play multiple games. Additionally, an intuitive user interface that allowed the user to navigate between digits before finalizing their guess was included.

Several other specifications were considered but not implemented due to time constraints. Most notably, only a two-person version of the game was used in the final design, although a one-person version would have been implemented given more time. This would have involved setting the code using a pseudo-random number generator rather than by

entering the digits manually. Additionally, functionality to display gameflow information such as the number of guesses, the number of guesses left, and the number of games won and lost was considered but ultimately prioritized behind the other specifications.

Components

The four principal components of the MasterMind game are the User Interface, Data Loader, Comparison Logic, and Gameflow Controller [**Appendix A**]. From the functional block diagram, it becomes is clear that in traditional RTL design terminology, the UI, Data Loader, and Comparison Logic comprise the datapath while the Gameflow Controller is a controller. While the datapath has a clear flow of information from the UI to the Data Loader to the Comparison Logic, I will describe the details in the reverse order, and then describe the controller. This is because this is the order of relative importance in relation to the specifications, and also the order I built the components in.

Comparison Logic

The high level function of the Comparison Logic is simple: given a code and a guess that are both stored in memory, compare the two to output *num_white* and *num_red*, which are 3-bit vectors corresponding to the number of white and red pegs associated with a certain guess. Additionally, the comparison logic should output the signal *correct_guess*, which goes high if the guess matches the code. This comparison takes place during the duration *compare_en* is high, which is 16 clock cycles.

The Comparison Logic block consists of two 2-bit counters, two 4x4 register files, two 16-bit serial-to-parallel shift registers, and numerous blocks of asynchronous logic[**Appendix B**]. The counter, *counter1*, enabled by the input *compare_en*, provides a

counter_1_TC signal to enable the other counter, *counter2*. This means that *counter1* is incremented four times as fast as *counter2*, so in 16 clock cycles they cycle through every possible combination of their *count* outputs. The first counter's *count* output is connected to the *r_addr* pin of the register file containing the code and the second counter is connected similarly to the register file containing the stored guess. Both these register files' *r_en* pins are connected to *compare_*en so that they output data at the same time the counters are incrementing. This creates two serial signals out of the register files which sequentially compare each digit in the code to each digit in the code. By using the *r_data* outputs of the two register files as inputs to a comparator, I create a one-bit serial signal *data_equal*. I also simultaneously create a one-bit serial signal *addr_equal* by comparing the signals at the *r_addr* pins of the register files. Using asynchronous logic, two signals *ser_white* and *ser_red* are created from the *addr_equal* and *data_equal signals*. As per the rules of Mastermind, the *ser_red* signal is equal to (*data_equal* AND *addr_equal*) and the *ser_white* signal is (*data_equal* AND NOT(*addr_equal*)).

These two serial signals are loaded into 16-bit serial-to-parallel shift registers. The *shift_en* pin of these registers is connected to *compare_en* so that the counters, register files, and shift registers are all enabled simultaneously, loading the 16 pertinent bits into the shift registers. So keeping *compare_en* high for exactly 16 clock cycles effectively shifts the 16 bits resulting from the comparison of the digits and addresses of the code to the digits and addresses of the guess. After *compare_en* is turned off, the data must be loaded into the ouputs of the serial-to-parallel registers. A *load_en* signal for the registers is generated by taking the asynchronous output (*counter_1_TC* AND *counter_2_TC*) and delaying it by one clock cycle with a register. After loading the registers, the two signals *par_white_16* and *par_red_16* are then ready to be processed using asynchronous logic to remove the dubious true *par_white_16* bits which result from double counting of repeated digits. The process of figuring out how to remove these overcounts was by far the most challenging aspect of the project, but

using combinatorial analysis, I was able to remove any possibility of overcounting due to repeated digits.

To get from the *par_red_16* to *num_red* is quite straight forward since there is no possibility of overcounting the number of red pegs. Of the bits of *par_red_16*, the only relevant bits are the ones where the addresses of the register files being compared were equal. These are bits 0, 5, 10, and 15. These bits are assigned to a signal *par_red_4*, which is fed to a look-up-table, *num_true_LUT*, which outputs a 3-bit vector corresponding to the number of bits which were high. This look up table was actually implemented in code using the asynchronous logic derived from the Karnaugh maps relating any 4-bit input signal to its 3-bit output which tells the number of input bits that are high.

To reiterate, getting from *par_white_16* to *num_white* is not as straight forward because of the possibility of overcounting. There are two ways to overcount white pegs in *par_white_16*: when a single digit of the guess matches multiple digits in the code, and when a single digit of the code matches multiple digits of the guess. To combat this problem, I had to create two 4-bit parallel signals analogous to *par_red_4*: *par_white_4_by guess* and *par_white_4_by_code*. Since each digit of the code/guess is compared to the four digits of the guess/code, we can eliminate overcounts by ORing together the indices of *par_white_16* which correspond to the same digit in either the guess or code. For example, the indices 0, 1, 2 and 3, correspond to the comparisons of the first digit of the guess with all digits of the code. Likewise, the indices, 0, 4, 8, and 12 correspond to the comparisons of the first digit of the code with all comparisons of the guess. By feeding groups of indices like this into 3-input OR gates (ignoring indices 0, 5, 10, and 15, since those correspond to red pins), I was able to create the signals *par_white_4_by guess* and *par_white_4_by_code*. Each of these 4-bit vectors is fed to a *num_true_LUT*, yielding the signals *num_white_by_guess* and *num_white_by_code*. By assigning the minimum of these two signals to the final output, *num_white*, I ensure that no repeated digits in the code or guess will result in overcounting the number of white pegs.

Data Loader

At a high level, the Data Loader serves as an interface between the User Interface and the comparison logic. While it could have been seamlessly integrated into the comparison logic to create a single coherent datapath block, I deemed that since the Data Loader performs a fundamentally different task than any of the elements in the Comparison Logic, it should be its own entity in the high level block diagram.

The fundamental task of the Data Loader is to provide the register files in the Comparison Logic with signals *bcd_*digit and *addr* which are synchronized such that the guess or code is loaded into memory. The data loader takes signals *code* and *guess* from the User Interface which are assumed to be 16-bit parallel signals representing the four BCD digits of a code or guess. The Data Loader consists of two four-input multiplexers and a 2-bit counter[**Appendix C**].

The first multiplexer is controlled by the 2-bit concatenation of *w_en_guess* and *w_en_code*, which are also the signals used to control the *w_en* pins of the register files. This multiplexer's inputs "01" and "10" are tied to the 16-bit inputs *code* and *guess* respectively, while the multiplexer's inputs "00" and "11" are tied to the constant value 0. While the same functionality here can be implemented with more concise asynchronous logic than a multiplexer, the multiplexer reflects better the high level purpose of this portion of the Data Loader: to switch between reading the code and the guess.

The 16-bit output of the first multiplexer is then split into its individual 4-bit BCD digit components, each of which is fed to a different input of the second 4-input multiplexer. This multiplexer is controlled by the output of the 2-bit counter. The 2-bit counter is enabled by (*w_en_code X*OR *w_en_guess*). Assuming the control signals *w_en_code* and *w_en_guess* go high for exactly four clock cycles, the output of the second multiplexer, *bcd_digit*, increments

through all four digits of either the code or the guess. The counter's output provides a signal, *addr*, that can be connected to the *w_addr* pins of the register files while *bcd_digit* is connected to the register files' *w_data* pins. The counter also serves the purpose of providing an output status signal, *done_storing*, from its *TC* pin.

While the Comparison Logic block code has separate inputs for the *w_data* and *w_addr* of the code register file and the guess register file, these inputs ultimately got connected to the same data and address signal. This was not problematic, though, because the two register files are still enabled by different signals. Assuming that *w_en_code* and *w_en_guess* are never high at the same time, the Data Loader will never accidently be writing to the wrong register file.

User Interface

At high level, the User Interface needs to convert the users' button pushes to signals that are usable by the Gameflow Controller and Data Loader. Separately, it needs to convert signals from the button pushes, Gameflow Controller, and Comparison Logic to signals which can be used to drive LEDs and the 7 segment display of the Basys 3.

As referenced in the **Specifications** subsection, the bare minimum requirement was that the User Interface be able to output signals *guess* and *code* correctly. This requirement could have been met with an inconvenient, unintuitive user interface. For example, I could have built a User Interface where, using only an increment button and enter button, the user chooses the first digit of their guess, presses enter, chooses the second digit of their guess, presses enter, etc. and can't modify the first digit once they are on to the second and cant go back down to three once they are at four. However, I chose to go further here and make a User Interface where the user can see all four digits of their guess at once, can shift back and forth

between digits, and increment and decrement a digit. Additionally, there is an indicator to show the user which digit is being modified by increment/decrement commands.

The User Interface consists of five monopulser circuits for the five buttons, four 4-bit counters with up/down functionality, a single 2-bit counter with up/down functionality, a 2-to-4 decoder, a 2-output demultiplexer, a 16-bit register, and a plethora of combinational logic gates[**Appendix D**]. The monopulsers are directly attached to physical buttons, and output signals *dig_up_mp*, *dig_down_mp*, *dig_right_mp*, *dig_left_mp*, *enter_mp* which are used for all other elements of the User Interface. The 2-bit counter is used to keep track of the active digit. The *up_down* pin of the counter is controlled by the combinational signal (*dig_right_mp* AND NOT(*dig_left_mp*)). The CE pin of the counter is controlled by the combinational signal (*dig_right_mp* XOR *dig_left_mp*). Using the XOR gate ensures that in the unlikely occasion that both the left and right buttons are pushed at the same time, nothing will happen to the active digit. The 2-bit counter's output is attached to the input of a 2-to-4 decoder. The decoder having one output high at a time allows for a single digit at a time to be the active digit.

Each of the four counters stores a single BCD digit. The counters' up/down functionality is of course indirectly controlled by the *dig_up_mp* and *dig_down_mp* signals. The up/down control signal for all the 4-bit counters, *UD_4b*, is the combinational output (*dig_up_mp* AND NOT(*dig_down_mp*)). Another combinational signal, *CE_all_4b*, is (*dig_up_mp* XOR *dig_down_mp*). *CE_all_4b* is ANDed with a different signal from the decoder at the *CE* pin of every counter. This means a counter is only incremented or decremented upon the push of *dig_up* or *dig_down* when it is the active counter. The counters have the additional feature that they will not roll over, i.e. increment above 9 or decrement below 0. The *RST* pins of the 4-bit counters are all connected to *enter_mp*. This means that when the user submits a new guess, they will start with a clean slate of zeros for the next guess. The outputs of all four counters are concatenated together to form the 16-bit signal, *counters_concat*, which is then passed to a demultiplexer. The demuxer routs *counters_concat* to the output *guess* if the control signal

code_or_guess is low and routs it to the output *code* if the output is high. This User Interface then produces the 16-bit *code* and *guess* signals that the Data Loader I designed requires as an input, and implements an intuitive way for the users to enter codes and guesses.

Gameflow Controller

The high level purpose of the Gameflow Controller is to provide control signals in order to moderate the process of switching between setting a code, making guesses, and giving feedback. The controller consists of a finite state machine, a monopulser and 4-bit counter to keep track of guesses, and a 5-bit counter to facilitate a time-based switching of states.

The finite state machine consists of eight states. In the order they would actually occur, these states are *setting_code*, *storing_code*, *setting_guess*, *storing_guess*, *comparing guess*, *comparing_guess2*, *giving_feedback*, and *game_over* [**Appendix E**]. For the most part, the high level behavior of these states is self-explanatory, with the exception of the distinction of *comparing_guess* and *comparing_guess_2*, which will be discussed later. The finite state machine's inputs from other blocks are *done_storing*, *enter_mp*, and *correct_guess*. It also has inputs from the counters: *guess_limit*, *eq_16*, and *eq_20*. The finite state machine has outputs that go to other blocks, *code_or_guess*, *w_en_code*, *w_en_guess*, *compare_en*, *feedback_en*, and outputs that go to the counters, *CE_compare_time*, *RST_compare_time*, *rst_guess_num* and *new_ guess*.

When the FSM is in state *setting_code*, all outputs are low. The user indicates that their code is ready by triggering *enter_mp*, which moves the FSM to *storing_code*. In *storing_code*, the only output that goes high is *w_en_code*, which triggers the Data Loader to load the code into the regter file. When it finishes this task, the Data Loader outputs *done_storing*, which transfers the FSM to *setting_guess*, where *code_or_guess* is the only output that goes high. Once the user is done entering a guess and *enter_mp* goes high, the FSM

moves to state *storing_guess*, where *w_en_guess* and *new_guess* are high. The signal *new_guess* is passed through a monopulser before going to the 4-bit guess counter's *CE* pin. This ensures that the guess counter gets incremented only once for each new guess. Once *done_storing* goes high, the FSM transitions to the state *comparing_guess*, where *compare_en* and *CE_compare_time* are high. *CE_compare_time* going high lets the 5-bit counter run, and when it has gone for 16 clock cycles, *eq_16* goes high, triggering a transition to state *comparing_guess_2*. In this state, *CE_compare_time* is still high but *compare_en* goes back to low. This additional state *comparing_guess* is needed because *compare_en* must be high for exactly 16 clock cycles for the comparison logic to work, but it needs a few clock cycles beyond that to reach a point where the feedback is actually ready to be given. Hence, the counter triggers *eq_20* after 20 clock cycles, which facilitates a transition to the state *giving_feedback*. In this state, *RST_compare_time* goes high in order to get the counter ready for the next guess, and *feedback_en* goes high. In this state, the user is able to see the number of red and white pins they earned, and must acknowledge the number by pressing *enter_mp*. If either *correct_guess* or *guess_limit* is true than the FSM transitions to *game_over*. Otherwise, it goes back to *setting_guess*. Finally, in the *game_over* state, *feedback_en* remains high so that the user knows if they won or lost. Additionally, *rst_guess_number* goes high, getting the FSM ready for the next game. When the user presses *enter_mp*, the game resets to state *setting_code*.

Other Components

Two other components were used that didn't fit well into the high level block diagram and therefore were not tested independently. First was the clocking mechanism.

All synchronous logic blocks were connected to the clock signal *clk100*, which had a frequency of 100 kHz. Since no operation in my circuit took more than 20 clock periods, I

deemed it was better to have a clock run on the slow side, to avoid risking running into metastability issues. *clk100* was the output of a BUFG component from the UNISIM library, which allowed the signal to be properly introduced into the clocking tree.

The other component was the Display Driver. This was the mux7seg component used in Labs 4 and 5. Its *y3*, *y2*, *y1* and *y0* pins were connected to the four 4-bit counters in the User Interface. Additionally, the *dp_set* pin was connected to the decoder output in the User Interface used to determine the active digit. This should light up the decimal point next to the digit being modified.

Evaluation

While the vast majority of the required work was completed on this project, the last piece, successful hardware validation, was not completed, so ultimately it cannot be declared a complete success.

However, the project performed very well in some respects. Most importantly, its Comparison Logic block was completely free from bugs when tested with numerous edge cases, and the combinatorial analysis shows that the algorithm should never produce incorrect results. Additionally, while I never actually got to see it work, the User Interface far exceeded the minimum requirements in its digit entering functionality. Finally, the project's high-level block diagram is very easy to understand for other digital designers who would want to modify the circuit.

There were several parts of the design which could have been improved upon, given additional time. Firstly, the intuitive high level block diagram came at the cost of having blocks which are much harder to interpret. The comparison logic block, for example, is so big that it can't even fit on a single page, and the UI block nearly exceeds a page. With blocks this big, I can also be fairly certain I did not produce the most concise design possible. For example,

my UI has shift registers dedicated to storing the code/guess, and my Comparison Logic has memory blocks for storing the code and guess. This is certainly redundant. Additionally, the controller uses its own counter to keep track of how long *compare_en* needs to be high for, but this could have been achieved instead with a status signal from the counters of the Comparison Logic.

Perhaps most notably though, my project had the opportunity to add more customizability to the game. It only had a single mode, which was two-player. A one-player mode could have been implemented fairly easily by adding a pseudorandom number generator connected to the *code* input of the Data Loader. This pseudo random number generator could be a block of memory containing all possible 4-digit codes, which is constantly being cycled through at very high speed by a counter. When the user is in *game_over* state, the pressing of *enter* would enable the data being read out at that moment to be loaded into the Comparison Logic. Adding additional states to the FSM, and perhaps and additional multiplexer in the Data Loader, I could create switching capability between the one-player and two-player modes. Some additional customizability I would have liked to includ given more time is easy, medium and hard modes. This could be achieved by simply changing the changing two constants: the upper limit of allowed digits for the code i.e. (0 through 6, 0 through 9, 0 through F), and the number of guesses allowed.

The project had other shortcomings that could most likely not be improved upon as well. For instance, the LEDs were used to tell the user how many red and white pegs they had earned. An LED on at 14, 13, 12, etc. indicated 4, 3, or 2 white pegs and an LED at 4, 3, 2, indicated 4, 3, or 2 red pegs. This was a somewhat confusing way to give the user feedback, but given the constraints of the Basys board, there were not many good options available. Additionally, to actually be able to perform the deductive logic necessary to guess the code, the user must right down each guess and its respective feedback on a piece of paper. It would be

desirable to have all guesses and all feedback displayed simultaneously, but given the constraints of the Basys board, this is not possible.

Conclusion

While the final hardware result was unsatisfying, I still feel proud of what I accomplished in this project. I devoted huge amounts of time to design, coding, and testing, and ended up creating a robust system that worked perfectly in every simulation I ran, and was probably only one or two bugs away from working on hardware.

I went through multiple iterations of each high level block of code, improving it each time. I managed to avoid spending too much debugging, since I was very careful in considering all possible scenarios as I designed my subcircuits. The one bug I did encounter, overcounting of white pegs, was due to a conceptual error which I struggled to rectify but ended being successful by using combinatorial techniques.

This project has taught me valuable lessons about myself, as well. When I first started RTL design, I was wildly unsuccessful in almost every daily exercise, and I started to think Electrical Engineering might not be for me. But by the end of this project, I feel confident I could tackle any of the final project options without too much trouble. I also learned that I need to manage my time much better. My project ended up not working on hardware because I didn't finish the other components in time to get assistance implementing the 7 segment display. Hopefully in the future, I will be able to exercise the same level of competence in RTL design while not falling behind on the work.

References

1. Regfile component from ENGS 31/CoSC 56 video Memory 1, courtesy of Prof. Geoffrey Luke
2. Mux7seg from ENGS 31/CoSC 56 Lab 4/5, courtesy of Prof. Eric Hansen

Appendices

Appendix A

Figure A1: High level state flow diagram

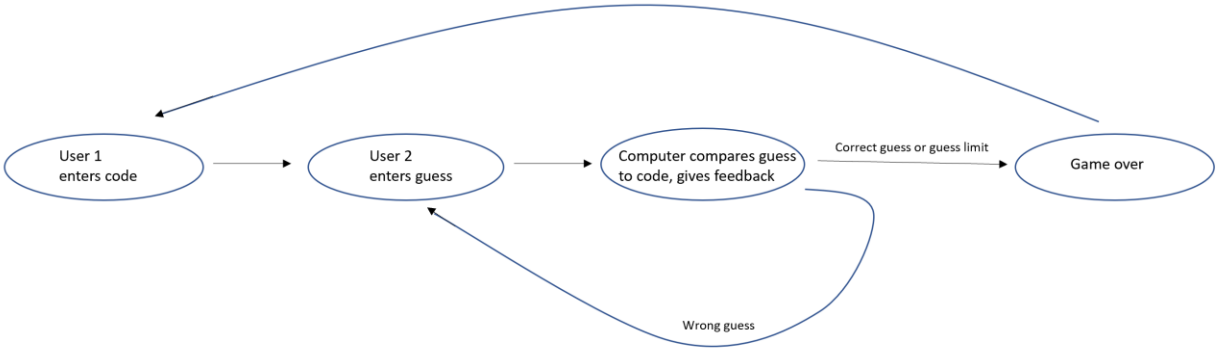


Figure A2: High level circuit diagram

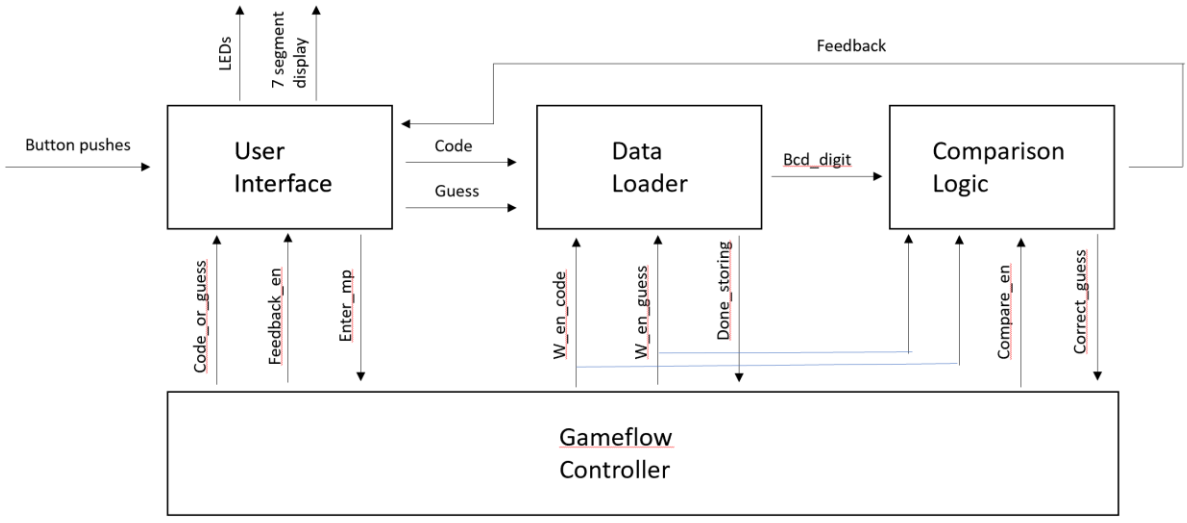


Figure A3: Final shell code

```
21
22  library IEEE;
23  use IEEE.STD_LOGIC_1164.ALL;
24  use IEEE.NUMERIC_STD.ALL;
25  library UNISIM;                    -- needed for the BUFG component
26  use UNISIM.Vcomponents.ALL;
27
28  entity shell_draft3 is
29      Port ( mclk : in STD_LOGIC;
30             enter : in STD_LOGIC;
31             dig_right : in STD_LOGIC;
32             dig_left : in STD_LOGIC;
33             dig_up : in STD_LOGIC;
34             dig_down : in STD_LOGIC;
35             --LED signals
36             red_LED_0 : out STD_LOGIC;
37             red_LED_1 : out STD_LOGIC;
38             red_LED_2 : out STD_LOGIC;
39             red_LED_3 : out STD_LOGIC;
40             red_LED_4 : out STD_LOGIC;
41             white_LED_0 : out STD_LOGIC;
42             white_LED_1 : out STD_LOGIC;
43             white_LED_2 : out STD_LOGIC;
44             white_LED_3 : out STD_LOGIC;
45             white_LED_4 : out STD_LOGIC;
46             --7 seg display signals
47             seg : out std_logic_vector(0 to 6);
48             dp : out std_logic;
49             an : out std_logic_vector(3 downto 0));
50  end shell_draft3;
51
52  architecture Behavioral of shell_draft3 is
53
54  --COMPONENT DECLARATIONS
55  ----------------------------------------------------
56
57  component entry_4dig is
58      Port ( clk : in STD_LOGIC;
59             dig_up : in STD_LOGIC;
60             dig_down : in STD_LOGIC;
61             dig_right : in STD_LOGIC;
```

```
62              dig_left : in STD_LOGIC;
63              enter : in STD_LOGIC;
64              code_or_guess : in STD_LOGIC;
65              enter_mp : out STD_LOGIC;
66              guess : out STD_LOGIC_VECTOR (15 downto 0);
67              code : out STD_LOGIC_VECTOR (15 downto 0);
68              --7 segment display signals
69              y3 : out STD_LOGIC_VECTOR (3 downto 0);
70              y2 : out STD_LOGIC_VECTOR (3 downto 0);
71              y1 : out STD_LOGIC_VECTOR (3 downto 0);
72              y0 : out STD_LOGIC_VECTOR (3 downto 0);
73              dp_set : out STD_LOGIC_VECTOR (3 downto 0));
74  end component;
75
76  component data_loader is
77      Port ( clk : in STD_LOGIC;
78             code : in STD_LOGIC_VECTOR (15 downto 0);
79             guess : in STD_LOGIC_VECTOR (15 downto 0);
80             w_en_guess : in STD_LOGIC;
81             w_en_code : in STD_LOGIC;
82             rst_count : in STD_LOGIC;  --superfluous
83             bcd_digit : out STD_LOGIC_VECTOR (3 downto 0);
84             addr : out STD_LOGIC_VECTOR (1 downto 0);
85             done_storing : out STD_LOGIC);
86  end component;
87
88  component guess_comparison_logic is
89  PORT (
90      clk: in std_logic;
91      compare_en: in std_logic;
92      rst_count: in std_logic;    --superfluous
93      w_addr_code: in std_logic_vector(1 downto 0);
94      w_data_code: in std_logic_vector(3 downto 0);
95      w_en_code: in std_logic;
96      w_addr_guess: in std_logic_vector(1 downto 0);
97      w_data_guess: in std_logic_vector(3 downto 0);
98      w_en_guess: in std_logic;
99      correct_guess: out std_logic;
100     num_white: out std_logic_vector(2 downto 0);
101     num_red: out std_logic_vector(2 downto 0));
```

```
end component;

component gameflow_controller is
    Port ( clk : in STD_LOGIC;
           done_storing : in STD_LOGIC;
           enter_mp : in STD_LOGIC;
           correct_guess : in STD_LOGIC;
           code_or_guess : out STD_LOGIC;
           compare_en : out STD_LOGIC;
           feedback_en : out STD_LOGIC;
           w_en_guess : out STD_LOGIC;
           w_en_code : out STD_LOGIC);
end component;

component mux7seg is
    Port (  clk : in  STD_LOGIC;
            y0, y1, y2, y3 : in  STD_LOGIC_VECTOR (3 downto 0);
            dp_set : in std_logic_vector(3 downto 0);
            seg : out  STD_LOGIC_VECTOR (0 to 6);
            dp : out std_logic;
            an : out  STD_LOGIC_VECTOR (3 downto 0) );
end component;

--LOCAL SIGNAL DECLARATIONS
-----------------------------------------

--signals to connect data loader to comparison logic
signal rst_count_DL: std_logic := '0';      --rst_count of the data loader
signal rst_count_CL: std_logic := '0';      --rst_count of the comparison logic
signal bcd_digit: std_logic_vector(3 downto 0) := "0000";         --output of data_loader; goes to both memories in comparison logic
signal addr: std_logic_vector(1 downto 0) := "00"; --output of data_loader; goes to both memories in comparison logic

--signals to connect UI to data loader
signal guess, code: std_logic_vector(15 downto 0) := (others => '0');

--signals to connect controller to all other components
signal done_storing, enter_mp, correct_guess, code_or_guess, compare_en, w_en_guess, w_en_code: std_logic := '0';

--signals to connect 7 segment display to UI
signal y3 : std_logic_vector(3 downto 0) := (others => '0');
signal y2 : std_logic_vector(3 downto 0) := (others => '0');
```

```
143 signal y1 : std_logic_vector(3 downto 0) := (others => '0');
144 signal y0 : std_logic_vector(3 downto 0) := (others => '0');
145 signal dp_set : std_logic_vector(3 downto 0) := (others => '0');
146 --signal ian : std_logic_vector(3 downto 0) := (others => '0');
147
148 --LED related signals
149 signal feedback_en : std_logic := '0';
150 signal num_white, num_red : std_logic_vector(2 downto 0) := "000";
151
152 --timing signals
153 constant clk100_tc: integer := 1000/2;
154 signal clk100_count: unsigned(9 downto 0) := (others => '0');
155 signal clk100_tog: std_logic := '0';
156 signal clk100 : std_logic := '0';
157
158 begin
159
160 --TIMING
161 -------------------------------
162 Slow_clock_buffer: BUFG
163     port map (  I => clk100_tog,
164                 O => clk100);
165
166 clock_divider: process(mclk)
167 begin
168     if rising_edge(mclk) then
169         if clk100_count = clk100_tc-1 then
170             clk100_count<=(others => '0');
171             clk100_tog<=NOT(clk100_tog);
172         else
173             clk100_count<=clk100_count+1;
174         end if;
175     end if;
176 end process;
177 ---------------------------------
178
179 LED_gen: process(num_red, num_white, feedback_en)
180     begin
181         if feedback_en='1' then
182             --defaults
```

```
183             red_LED_0 <='0';
184             red_LED_1 <= '0';
185             red_LED_2 <= '0';
186             red_LED_3 <= '0';
187             red_LED_4 <= '0';
188             white_LED_0 <= '0';
189             white_LED_1 <= '0';
190             white_LED_2 <= '0';
191             white_LED_3 <= '0';
192             white_LED_4 <= '0';
193              case num_red is
194                  when "000" => red_LED_0 <='1';
195                  when "001" => red_LED_1 <='1';
196                  when "010" => red_LED_2 <='1';
197                  when "011" => red_LED_3 <='1';
198                  when "100" => red_LED_4 <='1';
199                  when others =>
200                      red_LED_0 <='0';
201                      red_LED_1 <= '0';
202                      red_LED_2 <= '0';
203                      red_LED_3 <= '0';
204                      red_LED_4 <= '0';
205              end case;
206              case num_white is
207                  when "000" => white_LED_0 <= '1';
208                  when "001" => white_LED_0 <= '1';
209                  when "010" => white_LED_0 <= '1';
210                  when "011" => white_LED_0 <= '1';
211                  when "100" => white_LED_0 <= '1';
212                  when others =>
213                      white_LED_0 <= '0';
214                      white_LED_1 <= '0';
215                      white_LED_2 <= '0';
216                      white_LED_3 <= '0';
217                      white_LED_4 <= '0';
218              end case;
219          else
220             red_LED_0 <='0';
221             red_LED_1 <= '0';
222             red_LED_2 <= '0';
```

```
223             red_LED_3 <= '0';
224             red_LED_4 <= '0';
225             white_LED_0 <= '0';
226             white_LED_1 <= '0';
227             white_LED_2 <= '0';
228             white_LED_3 <= '0';
229             white_LED_4 <= '0';
230         end if;
231     end process;
232
233 --COMPONENT PORT MAPS
234 ----------------------------------------
235 UI: entry_4dig port map(
236     clk => clk100,
237     dig_up => dig_up,
238     dig_down => dig_down,
239     dig_right => dig_right,
240     dig_left => dig_left,
241     enter => enter,
242     code_or_guess => code_or_guess,
243     enter_mp => enter_mp,
244     guess => guess,
245     code => code,
246     y3 => y3,
247     y2 => y2,
248     y1 => y1,
249     y0 => y0,
250     dp_set => dp_set);
251
252 DL: data_loader port map(
253     clk => clk100,
254     code => code,
255     guess => guess,
256     w_en_guess => w_en_guess,
257     w_en_code => w_en_code,
258     rst_count => rst_count_DL,
259     bcd_digit => bcd_digit,
260     addr => addr,
261     done_storing => done_storing);
262
263 CL: guess_comparison_logic port map(
```

```
264         clk=>clk100,
265         compare_en=>compare_en,
266         rst_count=>rst_count_CL,
267         w_addr_code=>addr,
268         w_data_code=>bcd_digit,
269         w_en_code=>w_en_code,
270         w_addr_guess=>addr,
271         w_data_guess=>bcd_digit,
272         w_en_guess=>w_en_guess,
273         correct_guess=> correct_guess,
274         num_white=>num_white,
275         num_red=>num_red);
276 
277     controller: gameflow_controller port map(
278         clk => clk100,
279         done_storing => done_storing,
280         enter_mp => enter_mp,
281         correct_guess => correct_guess,
282         code_or_guess => code_or_guess,
283         compare_en => compare_en,
284         feedback_en => feedback_en,
285         w_en_guess => w_en_guess,
286         w_en_code => w_en_code);
287 
288     display: mux7seg port map(
289         clk => clk100,
290         y3 => y3,
291         y2 => y2,
292         y1 => y1,
293         y0 => y0,
294         dp_set => dp_set,
295         seg => seg,
296         dp => dp,
297         an => an );
298 
299     end Behavioral;
300 
```

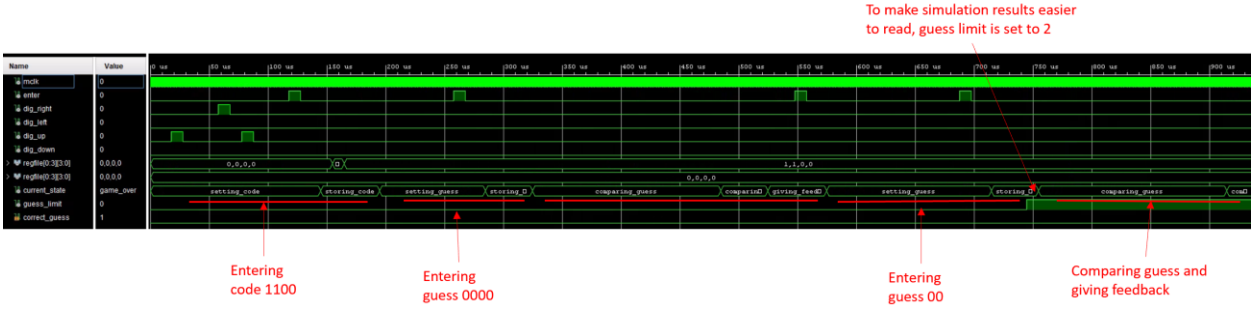
Figure A4: Stim_proc of testbench for entire digital system

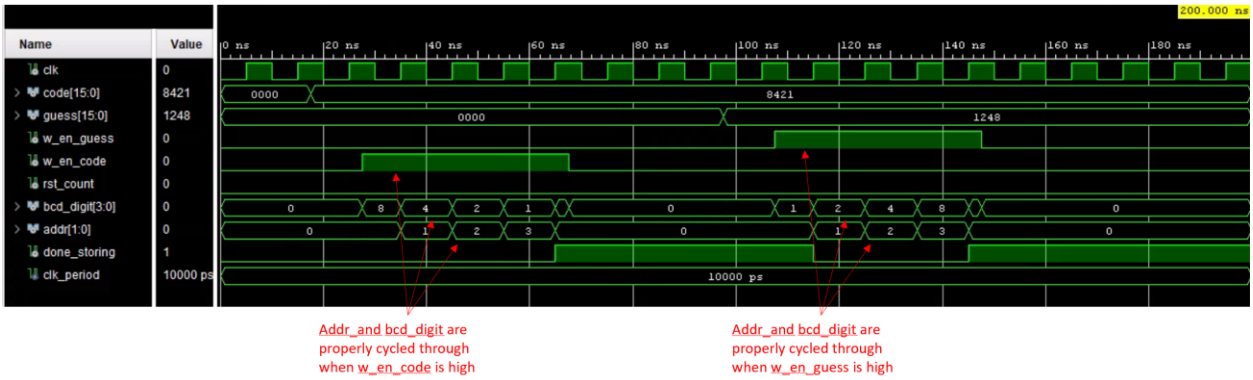
```
72      stim_proc: process
73      begin
74          wait for 1.75*clk_period;
75
76          --set first digit of code to 1
77          dig_up<='1';
78          wait for clk_period;
79          dig_up<='0';
80          wait for clk_period;
81          --move to 2nd digit
82          wait for 2*clk_period;
83          dig_right<='1';
84          wait for clk_period;
85          dig_right<='0';
86          wait for clk_period;
87          --set second digit of code to 1
88          dig_up<='1';
89          wait for clk_period;
90          dig_up<='0';
91          wait for clk_period;
92
93          --store the code
94          wait for 2*clk_period;
95          enter<='1';
96          wait for clk_period;
97          enter<='0';
98          wait for clk_period;
99
100         wait for 10*clk_period;
101
102         --enter guess 0000
103         wait for 2*clk_period;
104         enter<='1';
105         wait for clk_period;
106         enter<='0';
107         wait for clk_period;
108
109         wait for 25* clk_period;
110
```

```
111         --enter guess 0000
112         wait for 2*clk_period;
113         enter<='1';
114         wait for clk_period;
115         enter<='0';
116         wait for clk_period;
117
118         wait for 10* clk_period;
119
120         --enter guess 0000
121         wait for 2*clk_period;
122         enter<='1';
123         wait for clk_period;
124         enter<='0';
125         wait for clk_period;
126
127         wait for 25* clk_period;
128
129         --enter guess 0000
130         wait for 2*clk_period;
131         enter<='1';
132         wait for clk_period;
133         enter<='0';
134         wait for clk_period;
135
136         wait for 10* clk_period;
137
138         --
139         wait for 2*clk_period;
140         enter<='1';
141         wait for clk_period;
142         enter<='0';
143         wait for clk_period;
144
145         wait for 5* clk_period;
146
147         --
148         wait for 2*clk_period;
149         enter<='1';
150         wait for clk_period;
```

```
151     enter<='0';
152     wait for clk_period;
153
154     wait for 10*clk_period;
155
156
157      --
158     wait for 2*clk_period;
159     enter<='1';
160     wait for clk_period;
161     enter<='0';
162     wait for clk_period;
163
164     wait for 25* clk_period;
165
166
167      --
168     wait for 2*clk_period;
169     enter<='1';
170     wait for clk_period;
171     enter<='0';
172     wait for clk_period;
173
174     wait for 5* clk_period;
175
176
177
178
179 wait;
180 end process;
```

Figure B5: Full circuit simulations





Appendix B

Figure B1: Comparison Logic circuit part 1

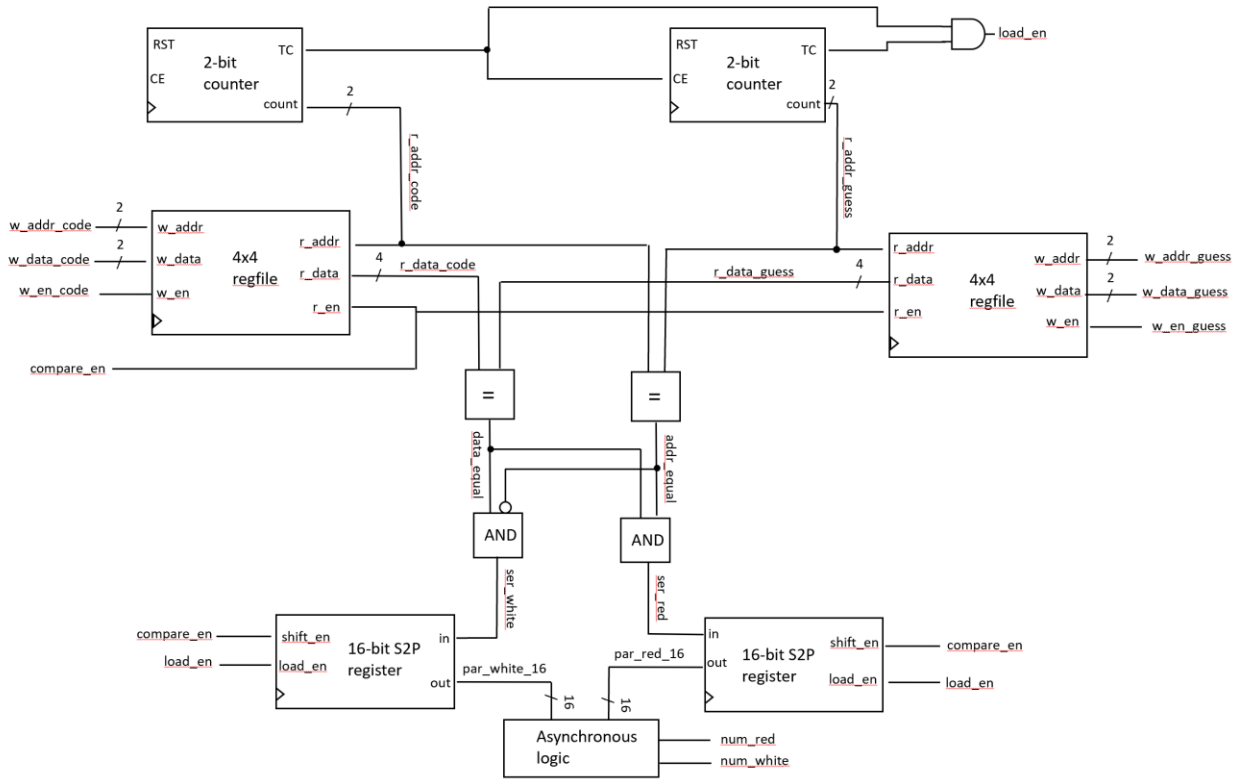


Figure B2: Comparison Logic circuit part 2. This shows what is inside the block labeled "Asynchronous Logic" in **B1**.

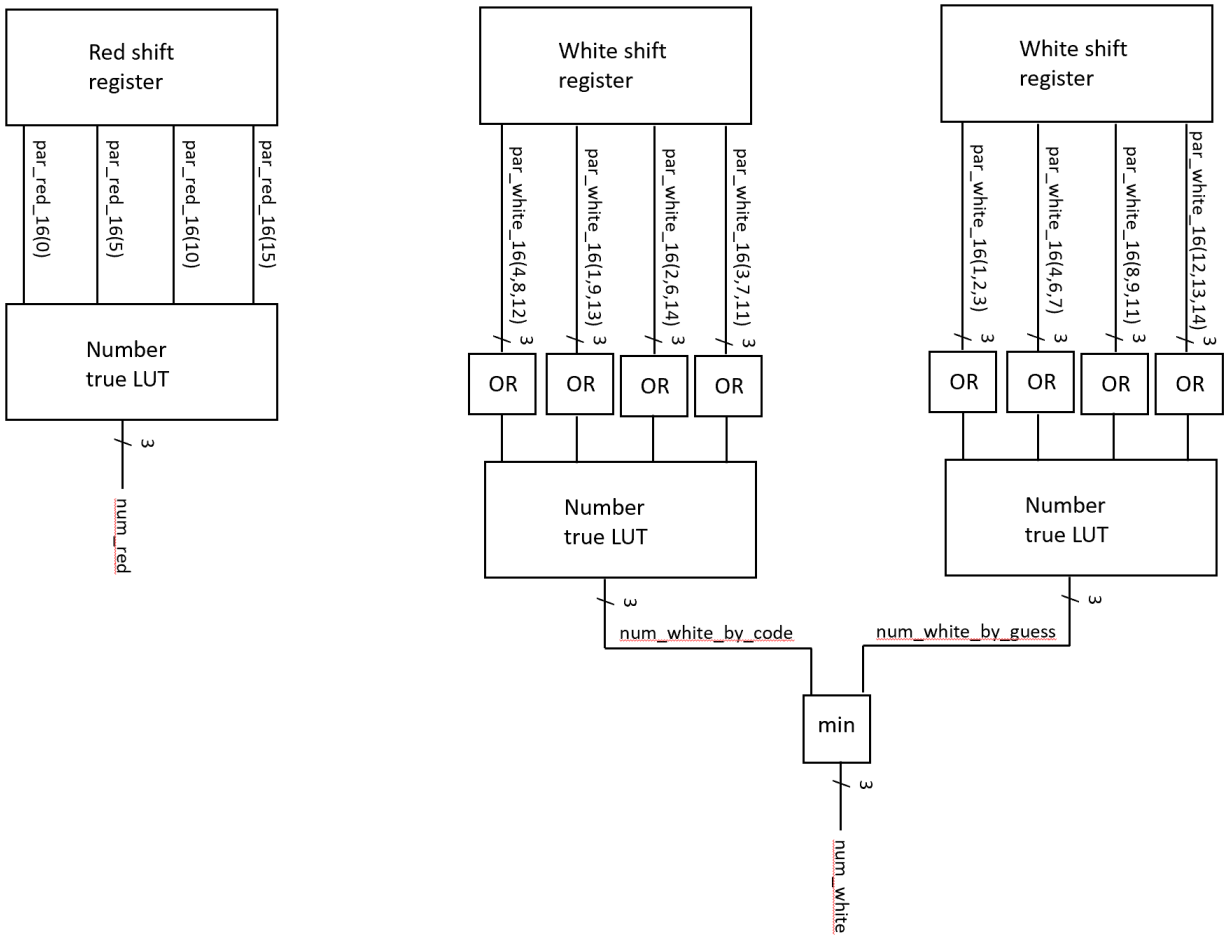


Figure B3.1: Code for the entire Comparison Logic component

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


entity guess_comparison_logic is
PORT (
    --general
    clk: in std_logic;
    compare_en: in std_logic;    --enables the first counter and the r_en of both regfiles
    --counter signals
    rst_count: in std_logic;    --is superfluous if you have compare_en going high for exactly 16 cycles
    --memory signals
    w_addr_code: in std_logic_vector(1 downto 0);
    w_data_code: in std_logic_vector(3 downto 0);
    w_en_code: in std_logic;
    w_addr_guess: in std_logic_vector(1 downto 0);
    w_data_guess: in std_logic_vector(3 downto 0);
    w_en_guess: in std_logic;
    --shift register signals

    --outputs
    correct_guess: out std_logic;
    num_white: out std_logic_vector(2 downto 0);
    num_red: out std_logic_vector(2 downto 0));
end guess_comparison_logic;

architecture Behavioral of guess_comparison_logic is

--counter
component counter_2bit is
     Port ( RST : in STD_LOGIC;
           CE : in STD_LOGIC;
           clk : in STD_LOGIC;
           TC : out STD_LOGIC;
           count : out STD_LOGIC_VECTOR(1 downto 0));
end component;
--memory
component regfile_4by4 is
    Port ( clk : in STD_LOGIC;
```

```
41          w_en : in STD_LOGIC;
42          w_addr : in STD_LOGIC_VECTOR(1 downto 0);
43          w_data : in STD_LOGIC_VECTOR(3 downto 0);
44          r_en : in STD_LOGIC;
45          r_addr : in STD_LOGIC_VECTOR(1 downto 0);
46          r_data : out STD_LOGIC_VECTOR(3 downto 0));
47  end component;
48  --shift register
49  component shift_register_16b is
50      Port ( clk : in STD_LOGIC;
51          ser_in : in STD_LOGIC;
52          shift_en : in STD_LOGIC;
53          load_en : in STD_LOGIC;
54          par_out : out STD_LOGIC_VECTOR(15 downto 0));
55  end component;
56  --LUTs
57  component num_true_LUT is
58      Port ( par_4 : in STD_LOGIC_VECTOR (3 downto 0);
59          num_true : out STD_LOGIC_VECTOR (2 downto 0));
60  end component;
61
62  --local signal declaration
63
64  --counters
65  signal r_addr_code: std_logic_vector(1 downto 0) := "00";       --connects counter1 to code memory address
66  signal r_addr_guess: std_logic_vector(1 downto 0) := "00";      --connects counter2 to guess memory address
67  signal counter_1_TC: std_logic := '0';                          --connects counter1 TC to counter2 CE
68  signal counter_2_TC: std_logic := '0';                          --no connection
69  signal r_data_code: std_logic_vector(3 downto 0) := "0000";     --connects code data to comparator
70  signal r_data_guess: std_logic_vector(3 downto 0) := "0000";    --connects guess data to comparator
71  --comparators
72  signal data_equal: std_logic := '0';                            --output of data comparison
73  signal addr_equal: std_logic := '1';                            --output of addr_commarison
74  --control signals
75  signal compare_en_delay: std_logic_vector(1 downto 0) := "00";  --used to delay the counter1 en by one clock cycle
76  signal load_en_delay: std_logic_vector(1 downto 0) := "00";     --used to delay the ser2par registers' load function. goes high after 16 cycles of compare_en
77  signal load_en: std_logic := '0';                              --undelayed signal generated from counters' TCs
78  --shift register signals
79  signal ser_white: std_logic := '0';                             --asynchronous logic signal going to white shift register
80  signal ser_red: std_logic := '0';                               --asynchronous logic signal going to red shift register
81  signal par_white_16: std_logic_vector(15 downto 0) := (others => '0'); --connect ser2par register to asynchrounous logic; 16 bits
```

```
signal par_white_16: std_logic_vector(15 downto 0) := (others => '0'); --connect ser2par register to asynchrounous logic; 16 bits
signal par_red_16: std_logic_vector(15 downto 0) := (others => '0'); --connects ser2par register to asynchronous logic; 16 bits
--LUT signals
signal par_white_4_by_guess: std_logic_vector(3 downto 0) := "0000";--comparing 1 digit of the guess to all digits of code. ie par_16(0:3), par_16(4:7) etc. each go to an OR gate
signal par_white_4_by_code: std_logic_vector(3 downto 0) := "0000"; --comparing 1 digit of the code to all digits of guess. ie par_16(0,4,8,12), par_16(1,5,9,13) etc. each go to an OR gate
signal par_red_4: std_logic_vector(3 downto 0) := "0000";           --only looking at indices 0,5,10,15 of par_16. No or gates necessary
signal num_white_by_guess: std_logic_vector(2 downto 0) := "000";   --output of the num_true_LUT for par_white_4_by_guess
signal num_white_by_code: std_logic_vector(2 downto 0) := "000";    --output of the num_true_LUT for par_white_4_by_code

signal num_red_buf: std_logic_vector(2 downto 0) := "000";          --signal used to create the real num_red and correct_guess

begin

--processes
clockcycledelay: process(clk)
    begin
        if rising_edge(clk) then
            compare_en_delay(1)<=compare_en_delay(0);
            compare_en_delay(0)<=compare_en;
            load_en_delay(1)<=load_en_delay(0);
            load_en_delay(0)<=load_en;
        end if;
    end process;

data_comparison: process(r_data_code,r_data_guess)
    begin
        if r_data_code=r_data_guess then
            data_equal<='1';
        else
            data_equal<='0';
        end if;
    end process;

addr_comparison: process(r_addr_code,r_addr_guess)
    begin
        if r_addr_code=r_addr_guess then
            addr_equal<='1';
        else
            addr_equal<='0';
        end if;
```

```
121     end process;
122
123 ser_redwhite_logic: process(data_equal, addr_equal)
124     begin
125         if data_equal='1' then
126             if addr_equal='1' then
127                 ser_red<='1';
128                 ser_white<='0';
129             else
130                 ser_white<='1';
131                 ser_red<='0';
132             end if;
133         else
134             ser_white<='0';
135             ser_red<='0';
136         end if;
137     end process;
138
139 load_en_gen: process(counter_1_TC, counter_2_TC)
140     begin
141         if counter_1_TC='1' AND counter_2_TC='1' then
142             load_en<='1';
143         else
144             load_en<='0';
145         end if;
146     end process;
147
148 par16_to_par4_byguess: process(par_white_16,par_red_16,par_red_4)
149     begin
150         --assigning red pins
151         if par_red_16(0)='1' then
152             par_red_4(0)<='1';
153         else
154             par_red_4(0)<='0';
155         end if;
156         if par_red_16(5)='1' then
157             par_red_4(1)<='1';
158         else
159             par_red_4(1)<='0';
160         end if;
```

```
161         if par_red_16(10)='1' then
162             par_red_4(2)<='1';
163         else
164             par_red_4(2)<='0';
165         end if;
166         if par_red_16(15)='1' then
167             par_red_4(3)<='1';
168         else
169             par_red_4(3)<='0';
170         end if;
171         --assigning white pins by guess
172         if ((par_white_16(1)='1' OR par_white_16(2)='1' OR par_white_16(3)='1') AND par_red_4(0)='0') then
173             par_white_4_by_guess(0)<='1';
174         else
175             par_white_4_by_guess(0)<='0';
176         end if;
177         if ((par_white_16(4)='1' OR par_white_16(6)='1' OR par_white_16(7)='1') AND par_red_4(1)='0')then
178             par_white_4_by_guess(1)<='1';
179         else
180             par_white_4_by_guess(1)<='0';
181         end if;
182         if ((par_white_16(8)='1' OR par_white_16(9)='1' OR par_white_16(11)='1') AND par_red_4(2)='0') then
183             par_white_4_by_guess(2)<='1';
184         else
185             par_white_4_by_guess(2)<='0';
186         end if;
187         if ((par_white_16(12)='1' OR par_white_16(13)='1' OR par_white_16(14)='1') AND par_red_4(3)='0') then
188             par_white_4_by_guess(3)<='1';
189         else
190             par_white_4_by_guess(3)<='0';
191         end if;
192         --assigning white pins by code
193         if ((par_white_16(4)='1' OR par_white_16(8)='1' OR par_white_16(12)='1') AND par_red_4(0)='0') then
194             par_white_4_by_code(0)<='1';
195         else
196             par_white_4_by_code(0)<='0';
197         end if;
198         if ((par_white_16(1)='1' OR par_white_16(9)='1' OR par_white_16(13)='1') AND par_red_4(1)='0')then
199             par_white_4_by_code(1)<='1';
```

```
200         else
201             par_white_4_by_code(1)<='0';
202         end if;
203         if ((par_white_16(2)='1' OR par_white_16(6)='1' OR par_white_16(14)='1') AND par_red_4(2)='0') then
204             par_white_4_by_code(2)<='1';
205         else
206             par_white_4_by_code(2)<='0';
207         end if;
208         if ((par_white_16(3)='1' OR par_white_16(7)='1' OR par_white_16(11)='1') AND par_red_4(3)='0') then
209             par_white_4_by_code(3)<='1';
210         else
211             par_white_4_by_code(3)<='0';
212         end if;
213     end process;
214 
215 min_numbyguess_numbycode: process(num_white_by_guess,num_white_by_code)
216     begin
217         if num_white_by_guess <= num_white_by_code then
218             num_white<=num_white_by_guess;
219         else
220             num_white<=num_white_by_code;
221         end if;
222     end process;
223 
224 red_gen_correct_gen: process(num_red_buf)
225     begin
226         num_red<=num_red_buf;
227         if num_red_buf="100" then
228             correct_guess<='1';
229         else
230             correct_guess<='0';
231         end if;
232     end process;
233 
234 --component port maps
235 counter1: counter_2bit port map(
236     RST => rst_count,
237     CE => compare_en_delay(1),
238     clk => clk,
```

```
239         TC => counter_1_TC,
240         count => r_addr_code);
241
242     counter2: counter_2bit port map(
243         RST => rst_count,
244         CE => counter_1_TC,
245         clk => clk,
246         TC => counter_2_TC,
247         count => r_addr_guess);
248
249     mem_code: regfile_4by4 port map(
250         clk => clk,
251         w_en => w_en_code,
252         w_addr => w_addr_code,
253         w_data => w_data_code,
254         r_en => compare_en,
255         r_addr => r_addr_code,
256         r_data => r_data_code);
257
258     mem_guess: regfile_4by4 port map(
259         clk => clk,
260         w_en => w_en_guess,
261         w_addr => w_addr_guess,
262         w_data => w_data_guess,
263         r_en => compare_en,
264         r_addr => r_addr_guess,
265         r_data => r_data_guess);
266
267     white_shift_register: shift_register_16b port map(
268         clk => clk,
269         ser_in => ser_white,
270         shift_en => compare_en_delay(1),
271         load_en => load_en_delay(1),
272         par_out => par_white_16);
273
274     red_shift_register: shift_register_16b port map(
275         clk => clk,
276         ser_in => ser_red,
277         shift_en => compare_en_delay(1),
```

```
279         par_out => par_red_16);
280 
281     white_LUT_by_guess: num_true_LUT port map(
282         par_4 => par_white_4_by_guess,
283         num_true => num_white_by_guess);
284 
285     white_LUT_by_code: num_true_LUT port map(
286         par_4 => par_white_4_by_code,
287         num_true => num_white_by_code);
288 
289     red_LUT: num_true_LUT port map(
290         par_4 => par_red_4,
291         num_true => num_red_buf);
292 
293 end Behavioral;
```

Figure B3.2: Code for the counter_2bit component

```
22  library IEEE;
23  use IEEE.STD_LOGIC_1164.ALL;
24  use IEEE.NUMERIC_STD.ALL;
25
26  entity counter_2bit is
27      Port ( RST : in STD_LOGIC;
28             CE : in STD_LOGIC;
29             clk : in STD_LOGIC;
30             TC : out STD_LOGIC;
31             count : out STD_LOGIC_VECTOR(1 downto 0));
32  end counter_2bit;
33
34  architecture Behavioral of counter_2bit is
35
36  signal TC_buf: std_logic := '0';
37  signal ucount: unsigned(1 downto 0) := "00";
38  begin
39
40  process(clk)
41      begin
42          if rising_edge(clk) then
43              if RST='1' then
44                  ucount<="00";
45              elsif CE='1' then
46                  ucount<=ucount+1;
47              end if;
48          end if;
49      end process;
50
51  process(ucount)
52      begin
53          if ucount="11" then
54              TC_buf<='1';
55          else
56              TC_buf<='0';
57          end if;
58      end process;
59  count<=std_logic_vector(ucount);
60  TC<=TC_buf;
61  end Behavioral;
```

Figure B3.2: Code for the regfile_4by4 component

```
22  library IEEE;
23  use IEEE.STD_LOGIC_1164.ALL;
24  use IEEE.NUMERIC_STD.ALL;
25
26
27  entity regfile_4by4 is
28      Port ( clk : in STD_LOGIC;
29             w_en : in STD_LOGIC;
30             w_addr : in STD_LOGIC_VECTOR(1 downto 0);
31             w_data : in STD_LOGIC_VECTOR(3 downto 0);
32             r_en : in STD_LOGIC;
33             r_addr : in STD_LOGIC_VECTOR(1 downto 0);
34             r_data : out STD_LOGIC_VECTOR(3 downto 0));
35  end regfile_4by4;
36
37  architecture Behavioral of regfile_4by4 is
38
39  type regfile_type is array(0 to 3) of std_logic_vector(3 downto 0);
40  signal regfile : regfile_type := (others => "0000");
41
42  begin
43
44  --write
45  process(clk)
46      begin
47          if rising_edge(clk) then
48              if w_en = '1' then
49                  regfile(to_integer(unsigned(w_addr)))<=w_data;
50              end if;
51          end if;
52      end process;
53
54  --read
55  process(r_addr, regfile)
56      begin
57          r_data<=regfile(to_integer(unsigned(r_addr)));
58      end process;
59
60  end Behavioral;
```

Figure B3.3: Code for the shift_register_16b component

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4
5  entity shift_register_16b is
6      Port ( clk : in STD_LOGIC;
7             ser_in : in STD_LOGIC;
8             shift_en : in STD_LOGIC;
9             load_en : in STD_LOGIC;
10            par_out : out STD_LOGIC_VECTOR(15 downto 0));
11 end shift_register_16b;
12
13 architecture Behavioral of shift_register_16b is
14
15 signal ser_data_reg: std_logic_vector(15 downto 0) := (others => '0');
16
17 begin
18
19     shift_register: process(clk)
20     begin
21         if rising_edge(clk) then
22             if shift_en = '1' then
23                 ser_data_reg(0)<=ser_in;
24                 ser_data_reg(1)<=ser_data_reg(0);
25                 ser_data_reg(2)<=ser_data_reg(1);
26                 ser_data_reg(3)<=ser_data_reg(2);
27                 ser_data_reg(4)<=ser_data_reg(3);
28                 ser_data_reg(5)<=ser_data_reg(4);
29                 ser_data_reg(6)<=ser_data_reg(5);
30                 ser_data_reg(7)<=ser_data_reg(6);
31                 ser_data_reg(8)<=ser_data_reg(7);
32                 ser_data_reg(9)<=ser_data_reg(8);
33                 ser_data_reg(10)<=ser_data_reg(9);
34                 ser_data_reg(11)<=ser_data_reg(10);
35                 ser_data_reg(12)<=ser_data_reg(11);
36                 ser_data_reg(13)<=ser_data_reg(12);
37                 ser_data_reg(14)<=ser_data_reg(13);
38                 ser_data_reg(15)<=ser_data_reg(14);
39             end if;
40         end if;
```

```
41      end process shift_register;
42
43      output_register: process(clk)
44      begin
45          if rising_edge(clk) then
46              if load_en = '1' then
47                  par_out<=ser_data_reg;
48              end if;
49          end if;
50      end process output_register;
51
52  end Behavioral;
```

Figure B3.4: Code for the num_true_LUT component

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

entity num_true_LUT is
    Port ( par_4 : in STD_LOGIC_VECTOR (3 downto 0);
           num_true : out STD_LOGIC_VECTOR (2 downto 0));
end num_true_LUT;

architecture Behavioral of num_true_LUT is

begin

process(par_4)
    begin
        --num_true(2)
        if (par_4(0)='1' and par_4(1)='1' and par_4(2)='1' and par_4(3)='1') then
            num_true(2)<='1';
        else
            num_true(2)<='0';
        end if;
        --num_true(1)
        if (par_4(3)='1' AND par_4(2)='1' AND par_4(1)='0') OR (par_4(0)='1' AND par_4(2)='1' AND par_4(3)='0') OR
            (par_4(3)='1' AND par_4(0)='1' AND par_4(2)='0') OR (par_4(3)='1' AND par_4(1)='1' AND par_4(0)='0') OR
            (par_4(2)='1' AND par_4(1)='1' AND par_4(3)='0') OR (par_4(1)='1' AND par_4(0)='1' AND par_4(3)='0') then

            num_true(1)<='1';
        else
            num_true(1)<='0';
        end if;
        --num_true(0)
        if (par_4(3)='1' xor par_4(2)='1' xor par_4(1)='1' xor par_4(0)='1') then
            num_true(0)<='1';
        else
            num_true(0)<='0';
        end if;
    end process;

end Behavioral;
```

Figure B4: Stim_proc of the Comparison Logic's testbench

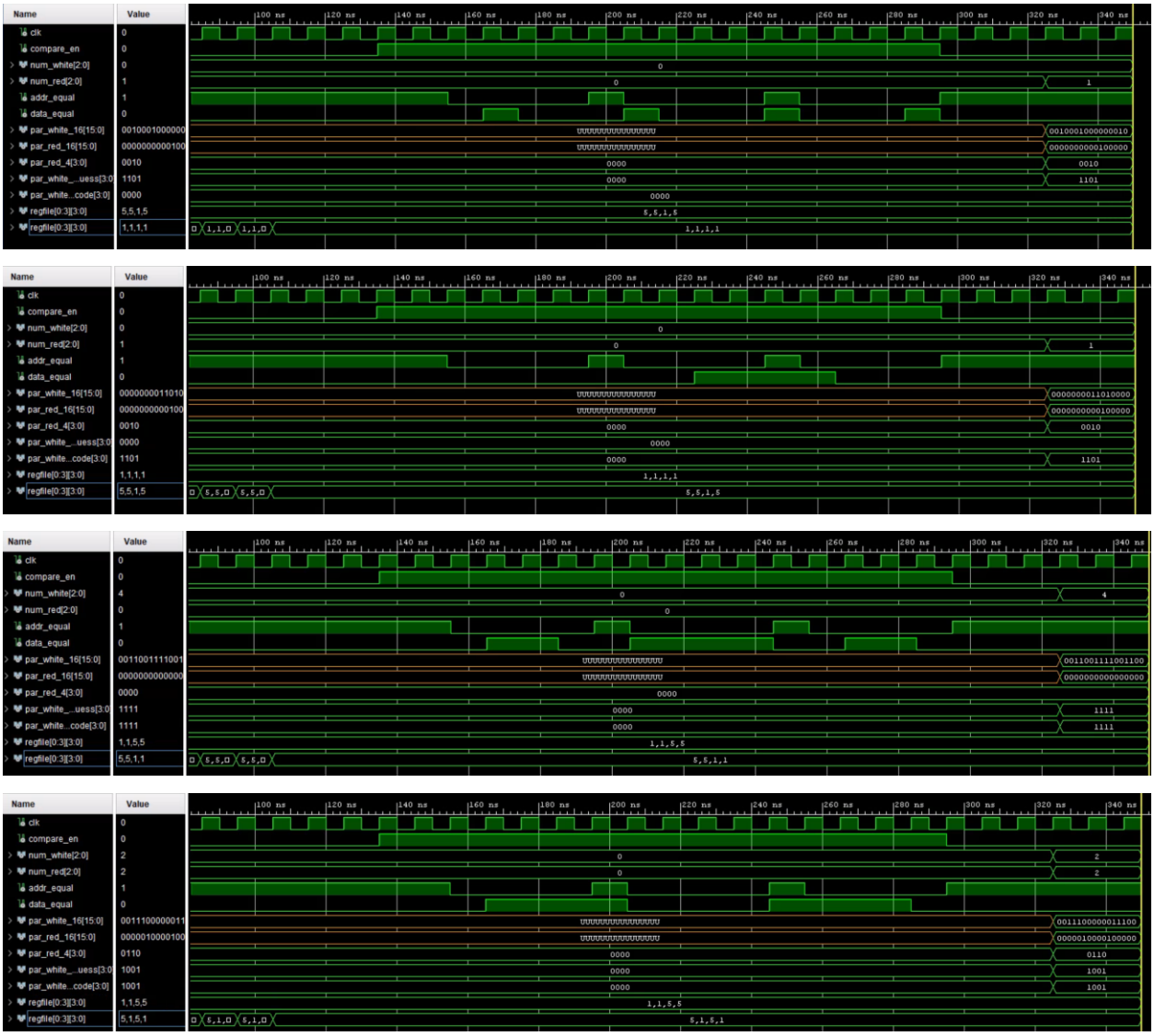
```
66  stim_proc: process
67  begin
68      wait for 1.75*clk_period;
69
70      rst_count<='1';
71      wait for clk_period;
72      rst_count<='0';
73      wait for clk_period;
74
75      --load code into the memory
76      w_en_code<='1';
77      w_addr_code<="00";
78      w_data_code<="0001";
79      wait for clk_period;
80      w_addr_code<="01";
81      w_data_code<="0011";
82      wait for clk_period;
83      w_addr_code<="10";
84      w_data_code<="0101";
85      wait for clk_period;
86      w_addr_code<="11";
87      w_data_code<="0111";
88      wait for clk_period;
89      w_en_code<='0';
90      w_addr_code<="00";
91
92      --load guess into the memory
93      w_en_guess<='1';
94      w_addr_guess<="00";
95      w_data_guess<="0001";
96      wait for clk_period;
97      w_addr_guess<="01";
98      w_data_guess<="0011";
99      wait for clk_period;
100     w_addr_guess<="10";
101     w_data_guess<="0101";
102     wait for clk_period;
103     w_addr_guess<="11";
104     w_data_guess<="0011";
```

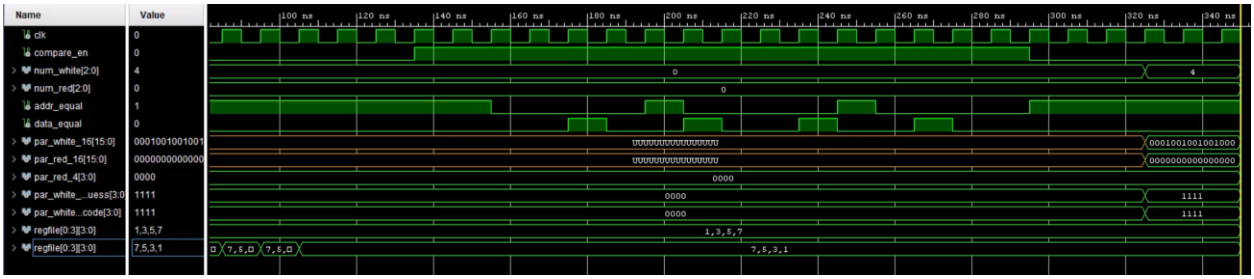
```
105         wait for clk_period;
106         w_en_guess<='0';
107         w_addr_guess<="00";
108
109         --compare code and guess
110         wait for 2*clk_period;
111         compare_en<='1';
112         wait for 16*clk_period;
113         compare_en<='0';
114
115         wait;
116     end process;
```

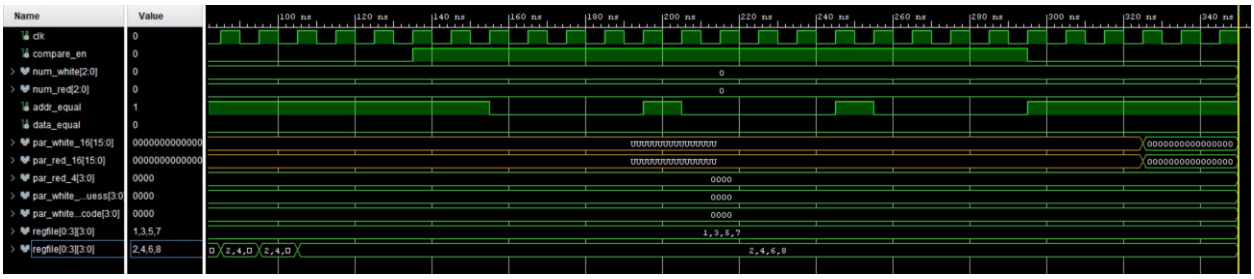
Figure B5.1: Table of edge cases tested to confirm Comparison Logic is effective

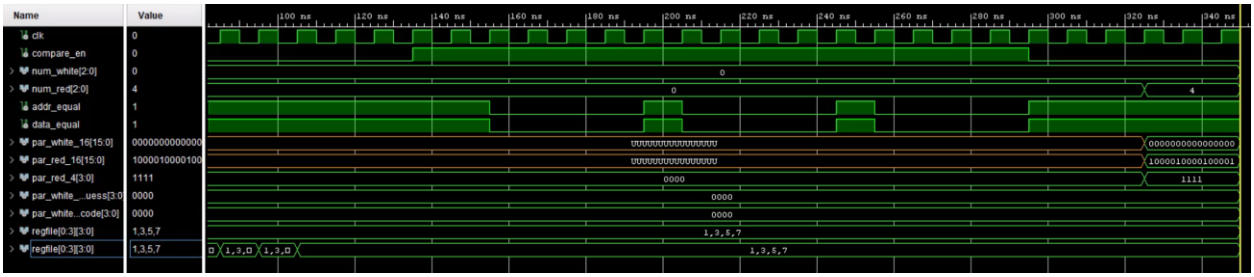
Code	Guess	Red	White
5515	11111	1	0
1111	5515	1	0
1155	5511	0	4
1155	5151	2	2
1357	7531	0	4
1357	2468	0	0
1357	1357	4	0
1377	1313	2	0
1377	3131	0	2
1357	1753	2	2
1357	1353	3	0

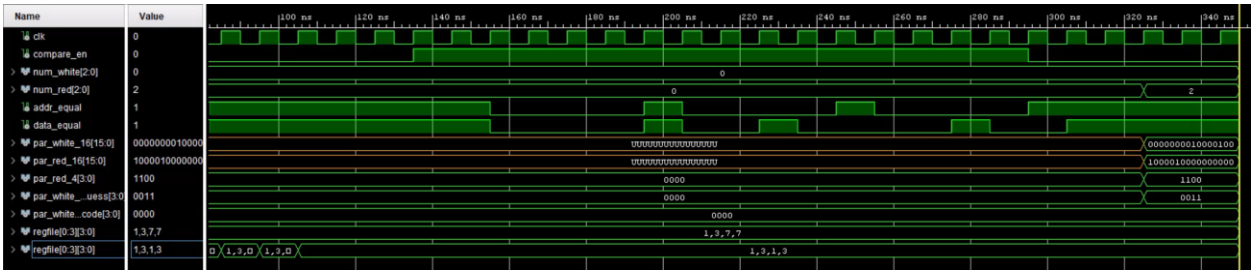
Figure B5.2: Simulations showing proper white and red pegs for each test case in **B5.1**. The important signals in these diagrams are *regfile[0:3][3:0]*, *num_white[2:0]*, and *num_red[2:0]*.

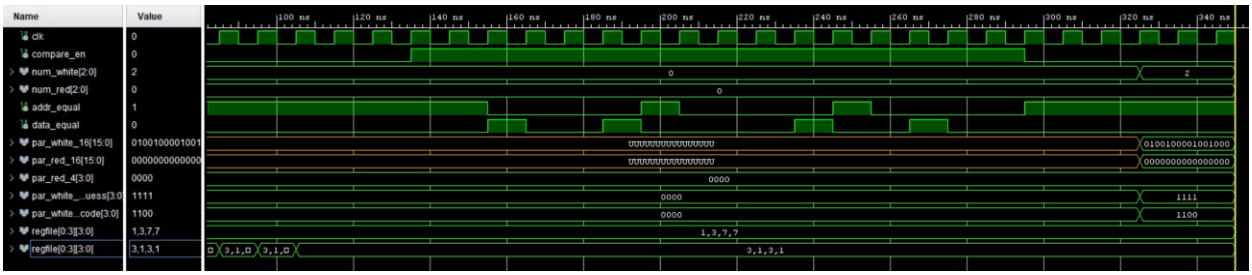


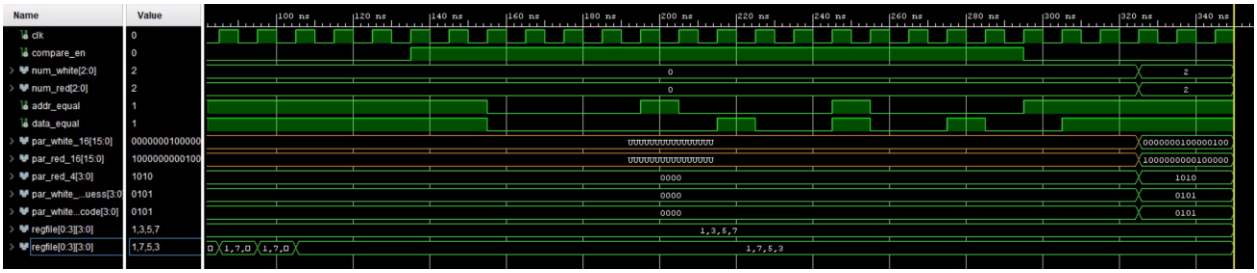


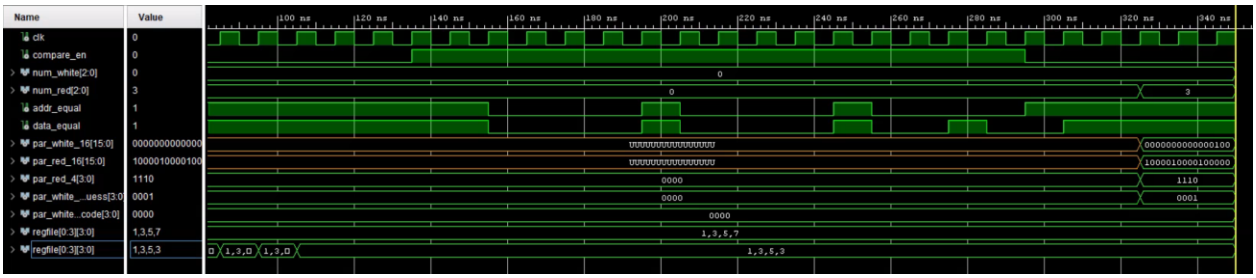












Appendix C

Figure C1: Data Loader circuit

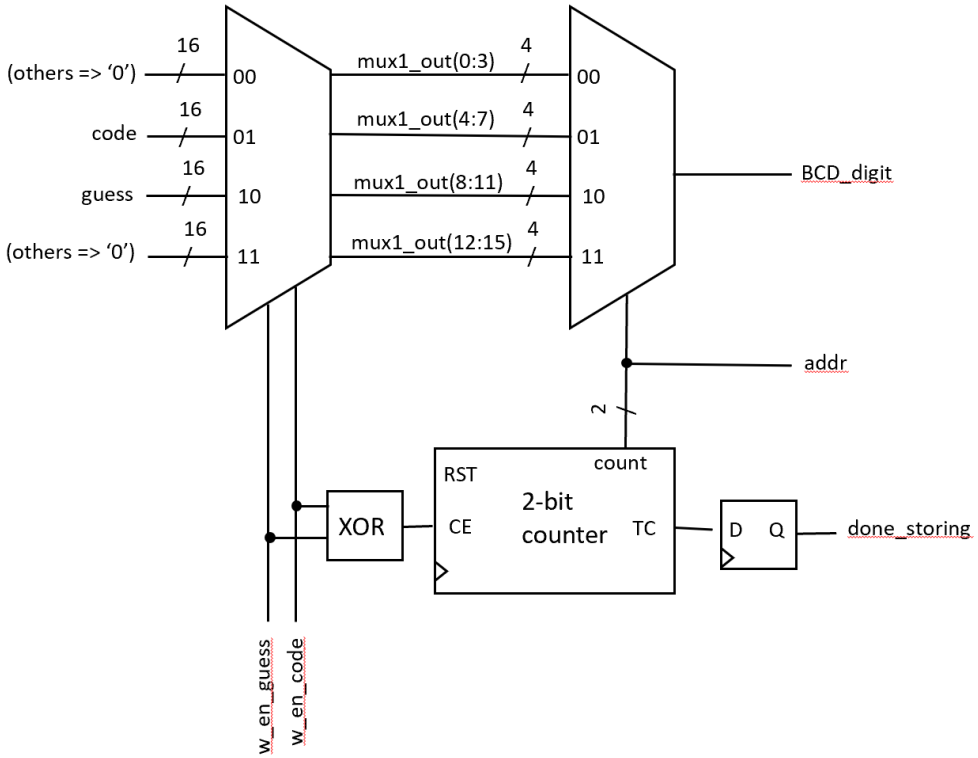


Figure C2: Data Loader Component code. Only internal component is the 2-bit counter, whose code is shown in **B3.2**

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


entity data_loader is
    Port ( clk : in STD_LOGIC;
           code : in STD_LOGIC_VECTOR (15 downto 0);
           guess : in STD_LOGIC_VECTOR (15 downto 0);
           w_en_guess : in STD_LOGIC;
           w_en_code : in STD_LOGIC;
           rst_count : in STD_LOGIC;  --superfluous
           bcd_digit : out STD_LOGIC_VECTOR (3 downto 0);
           addr : out STD_LOGIC_VECTOR (1 downto 0);
           done_storing : out STD_LOGIC);
end data_loader;

architecture Behavioral of data_loader is

--counter
component counter_2bit is
     Port ( RST : in STD_LOGIC; --superfluous
           CE : in STD_LOGIC;
           clk : in STD_LOGIC;
           TC : out STD_LOGIC;
           count : out STD_LOGIC_VECTOR(1 downto 0));
end component;

--local signal declaration
signal CE: std_logic := '0';
signal CE_delay: std_logic_vector(1 downto 0) := "00";
signal TC: std_logic := '0';
signal TC_delay: std_logic_vector(1 downto 0) := "00";
signal count: std_logic_vector(1 downto 0) := "00";
signal mux1_out: std_logic_vector(15 downto 0) := (others => '0');      --the 4 BCD digits of either a new code or a new guess

begin

--processes

```

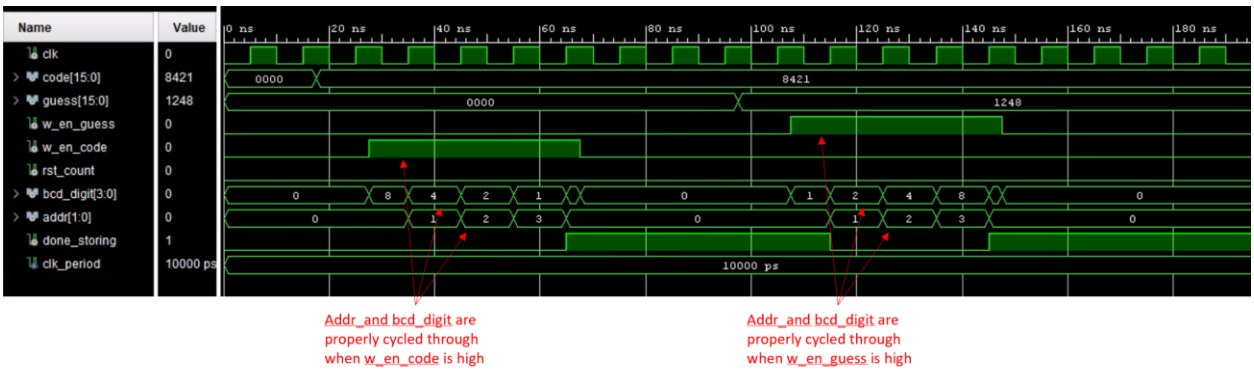
```
41  CE_gen: process(w_en_guess, w_en_code)
42      begin
43          if (w_en_guess='1' XOR w_en_code='1') then
44              CE<='1';
45          else
46              CE<='0';
47          end if;
48      end process;
49
50  clockcycledelay: process(clk)
51      begin
52          if rising_edge(clk) then
53              CE_delay(1)<=CE_delay(0);
54              CE_delay(0)<=CE;
55              TC_delay(1)<=TC_delay(0);
56              TC_delay(0)<=TC;
57          end if;
58      end process;
59
60  mux1: process(w_en_guess, w_en_code, code, guess)
61      begin
62      if (w_en_guess='0' AND w_en_code='1') then
63          mux1_out<=code;
64      elsif (w_en_guess='1' AND w_en_code='0') then
65          mux1_out<=guess;
66      else
67          mux1_out<=(others => '0');
68      end if;
69
70      end process;
71
72  mux2: process(count, mux1_out)
73      begin
74          case count is
75              when "00" =>
76                  bcd_digit <= mux1_out(15) & mux1_out(14) & mux1_out(13) & mux1_out(12);
77              when "01" =>
78                  bcd_digit <= mux1_out(11) & mux1_out(10) & mux1_out(9) & mux1_out(8);
79              when "10" =>
80                  bcd_digit <= mux1_out(7) & mux1_out(6) & mux1_out(5) & mux1_out(4);
```

```
81              when "11" =>
82                  bcd_digit <= mux1_out(3) & mux1_out(2) & mux1_out(1) & mux1_out(0);
83              when others =>
84                  bcd_digit <= "0000";
85          end case;
86      end process;
87
88  assignments: process(count)
89      begin
90          addr<=count;
91          done_storing<=TC;
92      end process;
93
94  --component port maps
95  counter: counter_2bit port map(
96      RST => rst_count,
97      CE => CE,
98      clk => clk,
99      TC => TC,
100     count => count);
101
102 end Behavioral;
```

Figure C3: Stim_proc of the testbench for the Data Loader

```
stim_proc: process
begin
    wait for 1.75*clk_period;

    --load code 8421 into memory
    code<="1000010000100001";
    wait for clk_period;
    w_en_code<='1';
    wait for 4*clk_period;
    w_en_code<='0';

    wait for 3*clk_period;

    --load guess 1248 into memory
    guess<="0001001001001000";
    wait for clk_period;
    w_en_guess<='1';
    wait for 4*clk_period;
    w_en_guess<='0';

    wait;
end process;

end testbench;
```

Figure C4: Simulation showing the Data Loader block works as expected



Appendix D

Figure D1: User Interface circuit

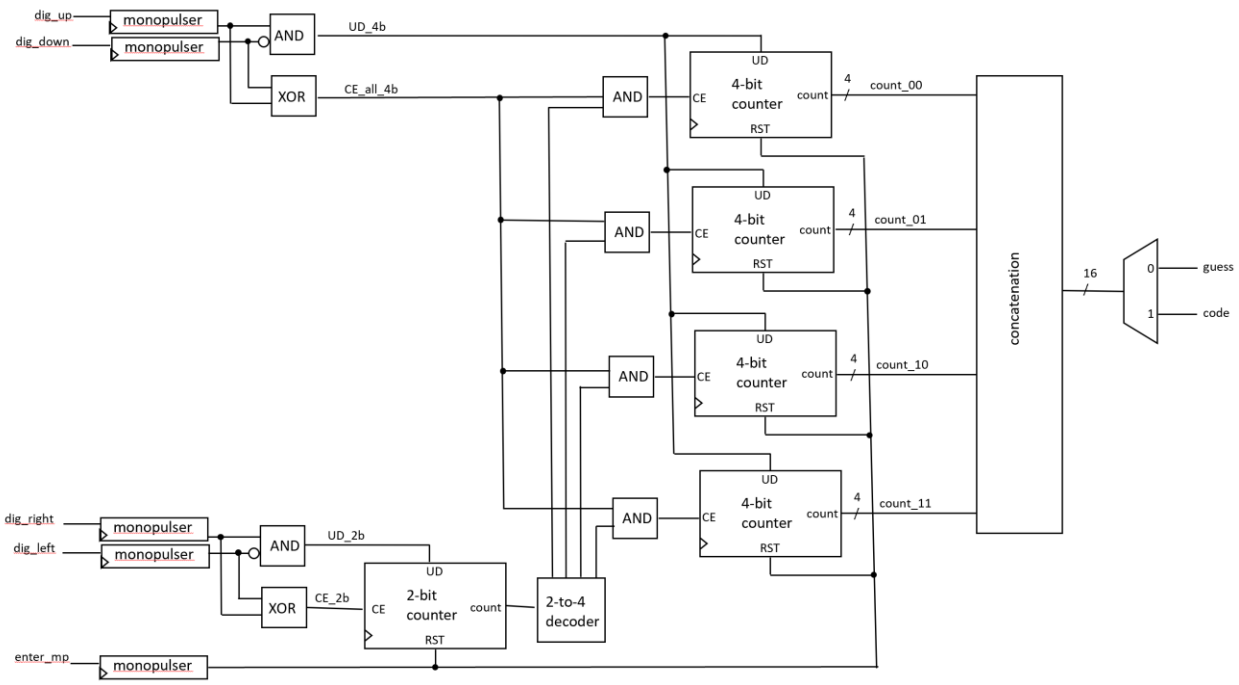


Figure D2.1: Code for entire User Interface

```
22  library IEEE;
23  use IEEE.STD_LOGIC_1164.ALL;
24  use IEEE.NUMERIC_STD.ALL;
25
26
27  entity entry_4dig is
28      Port ( clk : in STD_LOGIC;
29             dig_up : in STD_LOGIC;
30             dig_down : in STD_LOGIC;
31             dig_right : in STD_LOGIC;
32             dig_left : in STD_LOGIC;
33             enter : in STD_LOGIC;
34             code_or_guess : in STD_LOGIC;
35             enter_mp : out STD_LOGIC;
36             guess : out STD_LOGIC_VECTOR (15 downto 0);
37             code : out STD_LOGIC_VECTOR (15 downto 0);
38             --7 segment display signals
39             y3 : out STD_LOGIC_VECTOR (3 downto 0);
40             y2 : out STD_LOGIC_VECTOR (3 downto 0);
41             y1 : out STD_LOGIC_VECTOR (3 downto 0);
42             y0 : out STD_LOGIC_VECTOR (3 downto 0);
43             dp_set : out STD_LOGIC_VECTOR (3 downto 0));
44
45  end entry_4dig;
46
47  architecture Behavioral of entry_4dig is
48
49  --component declarations
50  component monopulser is
51      Port ( clk : in STD_LOGIC;
52             input : in STD_LOGIC;
53             output_mp : out STD_LOGIC);
54  end component;
55
56  component ud_counter_2b is
57      Port ( clk : in STD_LOGIC;
58             CE : in STD_LOGIC;
59             up_down : in STD_LOGIC;
60             RST : in STD_LOGIC;
61             count : out STD_LOGIC_VECTOR (1 downto 0));
62  end component;
```

```
63
64  component ud_counter_4b is
65      Port ( clk : in STD_LOGIC;
66             CE : in STD_LOGIC;
67             up_down : in STD_LOGIC;
68             RST : in STD_LOGIC;
69             count : out STD_LOGIC_VECTOR (3 downto 0));
70  end component;
71
72  --local signal declarations
73  signal CE_all_4b: std_logic := '0';     --each individual counter gets this ANDed with its respective decoder enable signal
74  signal CE_2b: std_logic := '0';
75  signal UD_4b: std_logic := '0';
76  signal UD_2b: std_logic := '0';
77  signal count_00: std_logic_vector(3 downto 0) := "0000";
78  signal count_01: std_logic_vector(3 downto 0) := "0000";
79  signal count_10: std_logic_vector(3 downto 0) := "0000";
80  signal count_11: std_logic_vector(3 downto 0) := "0000";
81  signal CE_00: std_logic := '0';
82  signal CE_01: std_logic := '0';
83  signal CE_10: std_logic := '0';
84  signal CE_11: std_logic := '0';
85  signal count_2b: std_logic_vector(1 downto 0) := "00";
86  signal counters_concat: std_logic_vector(15 downto 0) := (others => '0');
87  signal dig_up_mp : STD_LOGIC := '0';
88  signal dig_down_mp : STD_LOGIC := '0';
89  signal dig_right_mp : STD_LOGIC := '0';
90  signal dig_left_mp : STD_LOGIC := '0';
91  signal enter_mp_buf : STD_LOGIC := '0';
92
93  begin
94
95  --processes
96  CE_UD_2b_gen: process(dig_right_mp, dig_left_mp)    --asynchronous logic for the inputs of the 2 bit counter
97      begin
98          if (dig_right_mp='1' and dig_left_mp='0') then
99              UD_2b<='1';
100         else
101             UD_2b<='0';
102         end if;
103         if (dig_right_mp='1' xor dig_left_mp='1') then
```

```
104             CE_2b<='1';
105         else
106             CE_2b<='0';
107         end if;
108     end process;
109 
110 CE_UD_4b_gen: process(dig_up_mp, dig_down_mp)    --asynchronous logic for the inputs of the 4 bit counters
111     begin
112         if (dig_up_mp='1' and dig_down_mp='0') then
113             UD_4b<='1';
114         else
115             UD_4b<='0';
116         end if;
117         if (dig_up_mp='1' xor dig_down_mp='1') then
118             CE_all_4b<='1';
119         else
120             CE_all_4b<='0';
121         end if;
122     end process;
123 
124 individual_CE: process(count_2b,CE_all_4b)  --does the decoding and ANDing indirectly
125     begin
126         if CE_all_4b='1' then
127             case count_2b is
128                 when "00" =>
129                     CE_00<='1';
130                     CE_01<='0';
131                     CE_10<='0';
132                     CE_11<='0';
133                 when "01" =>
134                     CE_00<='0';
135                     CE_01<='1';
136                     CE_10<='0';
137                     CE_11<='0';
138                 when "10" =>
139                     CE_00<='0';
140                     CE_01<='0';
141                     CE_10<='1';
142                     CE_11<='0';
143                 when "11" =>
144                     CE_00<='0';
```

```
145                     CE_01<='0';
146                     CE_10<='0';
147                     CE_11<='1';
148                 when others =>
149                     CE_00<='0';
150                     CE_01<='0';
151                     CE_10<='0';
152                     CE_11<='0';
153                 end case;
154         else
155             CE_00<='0';
156             CE_01<='0';
157             CE_10<='0';
158             CE_11<='0';
159         end if;
160     end process;
161 
162 count_contat_gen: process(count_00,count_01,count_10,count_11)
163     begin
164         counters_concat(15)<=count_00(3);
165         counters_concat(14)<=count_00(2);
166         counters_concat(13)<=count_00(1);
167         counters_concat(12)<=count_00(0);
168         counters_concat(11)<=count_01(3);
169         counters_concat(10)<=count_01(2);
170         counters_concat(9)<=count_01(1);
171         counters_concat(8)<=count_01(0);
172         counters_concat(7)<=count_10(3);
173         counters_concat(6)<=count_10(2);
174         counters_concat(5)<=count_10(1);
175         counters_concat(4)<=count_10(0);
176         counters_concat(3)<=count_11(3);
177         counters_concat(2)<=count_11(2);
178         counters_concat(1)<=count_11(1);
179         counters_concat(0)<=count_11(0);
180     end process;
181 
182 output_reg_and_demux: process(clk)
183     begin
184     if rising_edge(clk) then
185         if enter_mp_buf='1' then
```

```
            if code_or_guess='1' then
                guess<=counters_concat;
            else
                code<=counters_concat;
            end if;
        end if;
    end if;
    end process;

process(enter_mp_buf)
    begin
        enter_mp<=enter_mp_buf;
    end process;

display_7seg_digits: process(count_00,count_01,count_10,count_11,count_2b)
    begin
        y3<=count_00;
        y2<=count_01;
        y1<=count_10;
        y0<=count_11;
        case count_2b is
            when "00" => dp_set<="1000";
            when "01" => dp_set<="0100";
            when "10" => dp_set<="0010";
            when "11" => dp_set<="0001";
            when others =>
        end case;
    end process;

--component port maps
mp_right: monopulser port map(
    clk => clk,
    input => dig_right,
    output_mp => dig_right_mp);

mp_left: monopulser port map(
    clk => clk,
    input => dig_left,
    output_mp => dig_left_mp);

```

```
226  mp_up: monopulser port map(
227      clk => clk,
228      input => dig_up,
229      output_mp => dig_up_mp);
230
231  mp_down: monopulser port map(
232      clk => clk,
233      input => dig_down,
234      output_mp => dig_down_mp);
235
236  mp_enter: monopulser port map(
237      clk => clk,
238      input => enter,
239      output_mp => enter_mp_buf);
240
241  counter_2b: ud_counter_2b port map(
242      clk => clk,
243      CE => CE_2b,
244      up_down => UD_2b,
245      RST => enter_mp_buf,
246      count => count_2b);
247
248  counter_00: ud_counter_4b port map(
249      clk => clk,
250      CE => CE_00,
251      up_down => UD_4b,
252      RST => enter_mp_buf,
253      count => count_00);
254
255  counter_01: ud_counter_4b port map(
256      clk => clk,
257      CE => CE_01,
258      up_down => UD_4b,
259      RST => enter_mp_buf,
260      count => count_01);
261
262  counter_10: ud_counter_4b port map(
263      clk => clk,
264      CE => CE_10,
265      up_down => UD_4b,
266      RST => enter_mp_buf,
```

```
267      count => count_10);
268
269  counter_11: ud_counter_4b port map(
270      clk => clk,
271      CE => CE_11,
272      up_down => UD_4b,
273      RST => enter_mp_buf,
274      count => count_11);
275
276  end Behavioral;
277
```

Figure D2.2: Monopulser component code

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity monopulser is
5      Port ( clk : in STD_LOGIC;
6             input : in STD_LOGIC;
7             output_mp : out STD_LOGIC);
8  end monopulser;
9
10 architecture Behavioral of monopulser is
11
12 signal sh_reg: std_logic_vector(1 downto 0) := "00";
13
14 begin
15
16 process(clk)
17     begin
18         if rising_edge(clk) then
19             sh_reg(1)<=sh_reg(0);
20             sh_reg(0)<=input;
21         end if;
22     end process;
23
24 process(clk)
25     begin
26         if rising_edge(clk) then
27             if sh_reg(1)='0' and sh_reg(0)='1' then
28                 output_mp<='1';
29             else
30                 output_mp<='0';
31             end if;
32         end if;
33     end process;
34 end Behavioral;
35
```

Figure D2.3: Up/down counter component code. 4-bit and 2-bit versions differ only in the number of bits

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity ud_counter_2b is
6      Port ( clk : in STD_LOGIC;
7             CE : in STD_LOGIC;
8             up_down : in STD_LOGIC;      --1 for up, 0 for down
9             RST : in STD_LOGIC;
10            count : out STD_LOGIC_VECTOR (1 downto 0));
11 end ud_counter_2b;
12
13 architecture Behavioral of ud_counter_2b is
14
15 signal ucount: unsigned(1 downto 0) := "00";--type casting
16 signal TC: std_logic := '0';               --used to prevent roll over
17
18 begin
19
20 countgen: process(clk)
21     begin
22         if rising_edge(clk) then
23             if RST='1' then
24                 ucount<="00";
25             else
26                 if CE='1' then
27                     if up_down='1' and TC='0' then
28                         ucount<=ucount+1;
29                     elsif up_down='0' and TC='0' then
30                         ucount<=ucount-1;
31                     end if;
32                 end if;
33             end if;
34         end if;
35     end process;
36
37 TCgen: process(ucount,up_down)
38     begin
39         if (up_down='0' and ucount="00") OR (up_down='1' and ucount="11") then
40             TC<='1';
41         else
42             TC<='0';
43         end if;
44     end process;
45
46 count<=std_logic_vector(ucount);
47
48 end Behavioral;
```

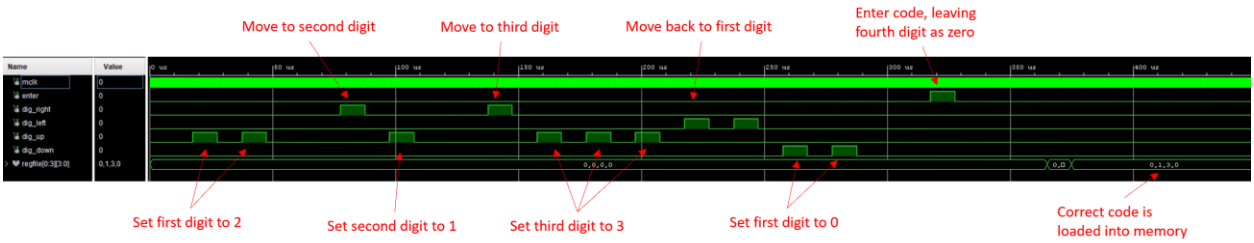
Figure D3: Stim_proc of testbench used for UI

```
52  stim_proc: process
53  begin
54      wait for 1.75*clk_period;
55
56      --set first digit to 4
57      dig_up<='1';
58      wait for clk_period;
59      dig_up<='0';
60      wait for clk_period;
61      dig_up<='1';
62      wait for clk_period;
63      dig_up<='0';
64      wait for clk_period;
65      dig_up<='1';
66      wait for clk_period;
67      dig_up<='0';
68      wait for clk_period;
69      dig_up<='1';
70      wait for clk_period;
71      dig_up<='0';
72      wait for clk_period;
73
74
75      --move to second digit
76      dig_right<='1';
77      wait for clk_period;
78      dig_right<='0';
79      wait for clk_period;
80
81      --set second digit to 3
82      dig_up<='1';
83      wait for clk_period;
84      dig_up<='0';
85      wait for clk_period;
86      dig_up<='1';
87      wait for clk_period;
88      dig_up<='0';
89      wait for clk_period;
90      dig_up<='1';
```

```
91          wait for clk_period;
92          dig_up<='0';
93          wait for clk_period;
94
95          --move to third digit
96          dig_right<='1';
97          wait for clk_period;
98          dig_right<='0';
99          wait for clk_period;
100
101         --set third digit to 2
102         dig_up<='1';
103         wait for clk_period;
104         dig_up<='0';
105         wait for clk_period;
106         dig_up<='1';
107         wait for clk_period;
108         dig_up<='0';
109         wait for clk_period;
110
111         --move to fourth digit
112         dig_right<='1';
113         wait for clk_period;
114         dig_right<='0';
115         wait for clk_period;
116
117         --set fourth digit to 1
118         dig_up<='1';
119         wait for clk_period;
120         dig_up<='0';
121         wait for clk_period;
122
123         --move back to second digit
124         dig_left<='1';
125         wait for clk_period;
126         dig_left<='0';
127         wait for clk_period;
128         dig_left<='1';
129         wait for clk_period;
130         dig_left<='0';
```

```
131         wait for clk_period;
132 
133         --decrement second digit to 1
134         dig_down<='1';
135         wait for clk_period;
136         dig_down<='0';
137         wait for clk_period;
138         dig_down<='1';
139         wait for clk_period;
140         dig_down<='0';
141         wait for clk_period;
142 
143         --sent data out as a code
144         code_guess<='0';
145         wait for clk_period;
146         enter<='1';
147         wait for clk_period;
148         enter<='0';
149         wait for clk_period;
150 
151         --sent data out as a guess
152         code_guess<='1';
153         wait for clk_period;
154         enter<='1';
155         wait for clk_period;
156         enter<='0';
157         wait for clk_period;
158 
159         wait;
160 
161     end process;
162 
163     end testbench;
```

Figure D4: Simulation showing that UI works as intended



Appendix E

Figure E1: Gameflow Controller circuit

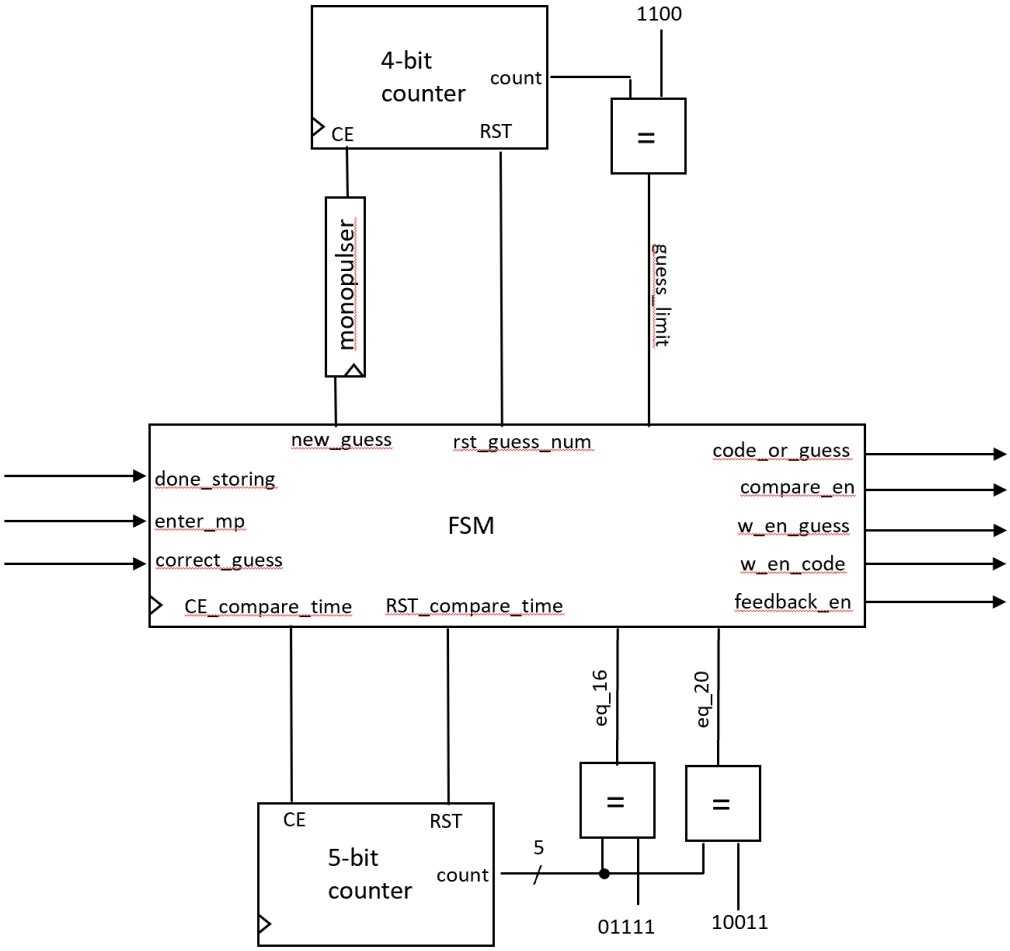


Figure E2: State Diagram

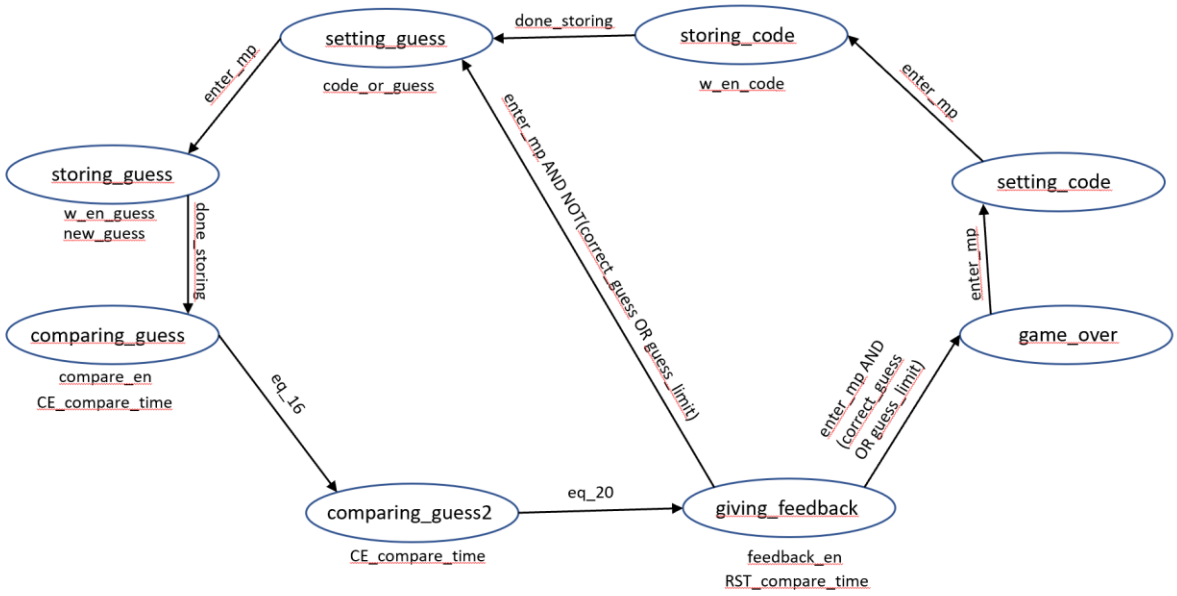


Figure E3: Code For the Gameflow Controller. Each of the counters are the same as others used, just with a different number of bits, so this component's separate code is not included. See **D2.2** for monopulser component code

```
22  library IEEE;
23  use IEEE.STD_LOGIC_1164.ALL;
24
25  entity gameflow_controller is
26      Port ( clk : in STD_LOGIC;
27             done_storing : in STD_LOGIC;
28             enter_mp : in STD_LOGIC;
29             correct_guess : in STD_LOGIC;
30             code_or_guess : out STD_LOGIC;
31             compare_en : out STD_LOGIC;
32             feedback_en : out STD_LOGIC;
33             w_en_guess : out STD_LOGIC;
34             w_en_code : out STD_LOGIC);
35  end gameflow_controller;
36
37  architecture Behavioral of gameflow_controller is
38
39  --component declarations
40  component ud_counter_4b is
41      Port ( clk : in STD_LOGIC;
42             CE : in STD_LOGIC;
43             up_down : in STD_LOGIC;
44             RST : in STD_LOGIC;
45             count : out STD_LOGIC_VECTOR (3 downto 0));
46  end component;
47
48  component monopulser is
49      Port ( clk : in STD_LOGIC;
50             input : in STD_LOGIC;
51             output_mp : out STD_LOGIC);
52  end component;
53
54  component counter_5b is
55      Port ( RST : in STD_LOGIC;
56             CE : in STD_LOGIC;
57             clk : in STD_LOGIC;
58             count : out STD_LOGIC_VECTOR(4 downto 0));
59  end component;
```

```
60  --local signal declarations
61
62  signal new_guess: std_logic := '0';
63  signal rst_guess_num: std_logic := '0';
64  signal guess_limit: std_logic := '0';
65  signal new_guess_mp: std_logic := '0';
66  signal guess_count: std_logic_vector(3 downto 0) := "0000";
67
68  signal CE_compare_time: std_logic := '0';
69  signal RST_compare_time: std_logic := '0';
70  signal eq_16: std_logic := '0';
71  signal eq_20: std_logic := '0';
72  signal count_compare_time: std_logic_vector(4 downto 0) := "00000";
73
74  type state_type is (setting_code,storing_code,setting_guess,storing_guess,comparing_guess,comparing_guess_2,giving_feedback,game_over);
75  signal current_state, next_state: state_type := setting_code;
76
77  begin
78  --processes
79  state_update: process(clk)
80      begin
81          if rising_edge(clk) then
82              current_state<=next_state;
83          end if;
84      end process;
85
86  next_state_logic: process(current_state,eq_16,eq_20,done_storing,enter_mp,correct_guess,guess_limit)
87      begin
88          --defaults
89          code_or_guess<='0';
90          compare_en<='0';
91          feedback_en<='0';
92          new_guess<='0';
93          rst_guess_num<='0';
94          w_en_guess<='0';
95          w_en_code<='0';
96          CE_compare_time<='0';
97          RST_compare_time<='0';
98          next_state<=current_state;
99              case current_state is
100                 when setting_code =>
```

```
101                 if enter_mp='1' then
102                     next_state<=storing_code;
103                 end if;
104             when storing_code =>
105                 w_en_code<='1';
106                 if done_storing='1' then
107                     next_state<=setting_guess;
108                 end if;
109             when setting_guess =>
110                 code_or_guess<='1';
111                 if enter_mp='1' then
112                     next_state<=storing_guess;
113                 end if;
114             when storing_guess =>
115                 w_en_guess<='1';
116                 new_guess<='1';
117                 if done_storing='1' then
118                     next_state<=comparing_guess;
119                 end if;
120             when comparing_guess =>
121                 compare_en<='1';
122                 CE_compare_time<='1';
123                 if eq_16='1' then
124                     next_state<=comparing_guess_2;
125                 end if;
126             when comparing_guess_2 =>
127                 CE_compare_time<='1';
128                 if eq_20='1' then
129                     next_state<=giving_feedback;
130                 end if;
131             when giving_feedback =>
132                 feedback_en<='1';
133                 RST_compare_time<='1';
134                 if (guess_limit='1' OR correct_guess='1') AND enter_mp='1' then
135                     next_state<=game_over;
136                 elsif enter_mp='1' then
137                     next_state<=setting_guess;
138                 end if;
139             when game_over =>
140                 feedback_en<='1';
141                 rst_guess_num<='1';
```

```
                    if enter_mp='1' then
                        next_state<=setting_code;
                    end if;
                when others =>
                    next_state<=current_state;

            end case;
    end process;

guess_lim_gen: process(guess_count)
    begin
        if guess_count="1100" then --max 12 guesses allowed
            guess_limit<='1';
        else
            guess_limit<='0';
        end if;
    end process;

eq_16_20_gen: process(count_compare_time)
    begin
        if count_compare_time="01111" then  --actually 15 because we start counting at 0
            eq_16<='1';
            eq_20<='0';
        elsif count_compare_time="10011" then   --actually 19
            eq_16<='0';
            eq_20<='1';
        else
            eq_16<='0';
            eq_20<='0';
        end if;
    end process;
--component port maps
guess_counter: ud_counter_4b port map(
    clk => clk,
    CE => new_guess_mp,
    up_down => '1',
    RST => rst_guess_num,
    count => guess_count);

mp: monopulser port map(
     clk => clk,
     input => new_guess,
     output_mp => new_guess_mp);

compare_time_counter: counter_5b port map(
     clk => clk,
     CE => CE_compare_time,
     RST => RST_compare_time,
     count => count_compare_time);

end Behavioral;
```

Figure E4: Stim_proc of testbench for Gameflow Controller

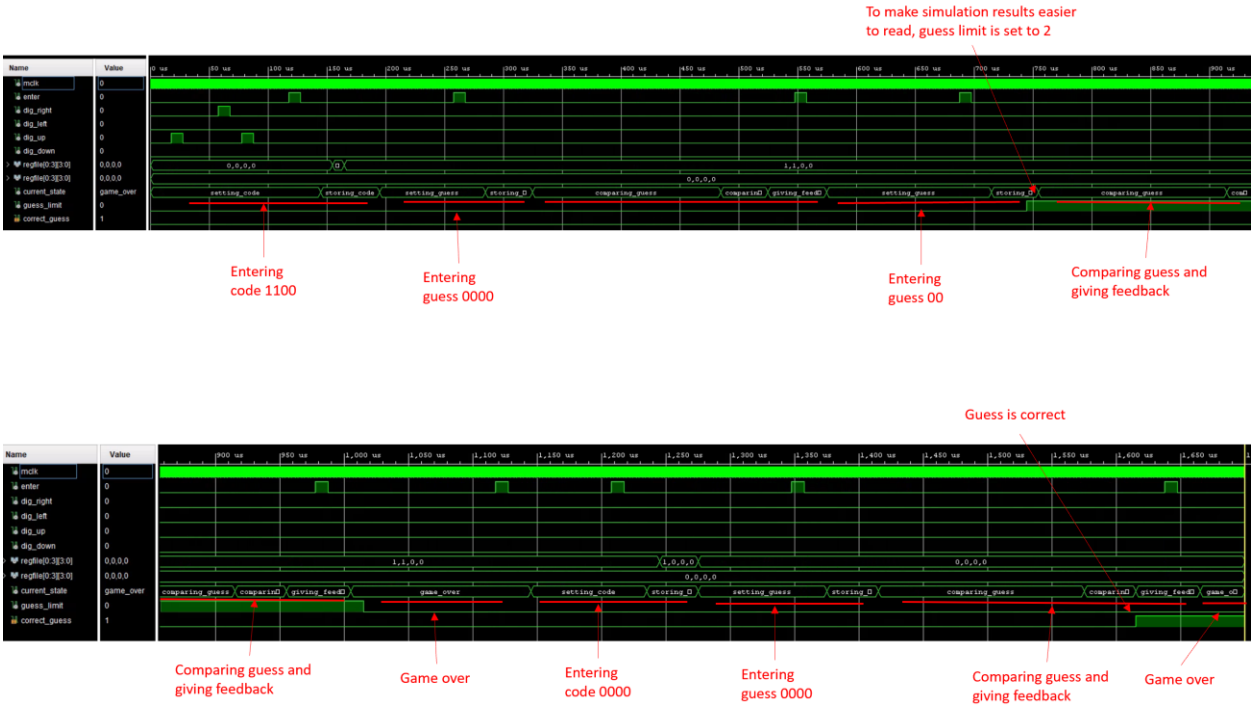
```
44  stim_proc: process
45  begin
46      wait for 1.75*clk_period;
47
48      --enter a code
49      enter_mp<='1';
50      wait for clk_period;
51      enter_mp<='0';
52      wait for clk_period;
53
54      --wait while coad is loaded
55      wait for 4*clk_period;
56      done_storing<='1';
57      wait for clk_period;
58      done_storing<='0';
59      wait for clk_period;
60
61      --enter a guess
62      enter_mp<='1';
63      wait for clk_period;
64      enter_mp<='0';
65      wait for clk_period;
66
67      --wait while guess is loaded
68      wait for 4*clk_period;
69      done_storing<='1';
70      wait for clk_period;
71      done_storing<='0';
72      wait for clk_period;
73
74      --wait while comparison occurs
75      wait for 19*clk_period;
76
77      --return to guess screen
78      enter_mp<='1';
79      wait for clk_period;
80      enter_mp<='0';
81      wait for clk_period;
82
```

```
83      --enter a guess
84      enter_mp<='1';
85      wait for clk_period;
86      enter_mp<='0';
87      wait for clk_period;
88
89      --wait while guess is loaded
90      wait for 4*clk_period;
91      done_storing<='1';
92      wait for clk_period;
93      done_storing<='0';
94      wait for clk_period;
95
96      --wait while comparison occurs
97      wait for 19*clk_period;
98
99      --return to guess screen
100     enter_mp<='1';
101     wait for clk_period;
102     enter_mp<='0';
103     wait for clk_period;
104
105     --enter a guess
106     enter_mp<='1';
107     wait for clk_period;
108     enter_mp<='0';
109     wait for clk_period;
110
111     --wait while guess is loaded
112     wait for 4*clk_period;
113     done_storing<='1';
114     wait for clk_period;
115     done_storing<='0';
116     wait for clk_period;
117
118     --wait while comparison occurs
119     wait for 19*clk_period;
120
121     --return to guess screen
```

```
121         --return to guess screen
122         enter_mp<='1';
123         wait for clk_period;
124         enter_mp<='0';
125         wait for clk_period;
126 
127         --enter a guess
128         enter_mp<='1';
129         wait for clk_period;
130         enter_mp<='0';
131         wait for clk_period;
132 
133         --wait while guess is loaded
134         wait for 4*clk_period;
135         done_storing<='1';
136         wait for clk_period;
137         done_storing<='0';
138         wait for clk_period;
139 
140         --wait while comparison occurs
141         wait for 19*clk_period;
142 
143         --go to game over screen
144         enter_mp<='1';
145         wait for clk_period;
146         enter_mp<='0';
147         wait for clk_period;
148 
149         --return to code screen
150         enter_mp<='1';
151         wait for clk_period;
152         enter_mp<='0';
153         wait for clk_period;
154 
155         --enter a code
156         enter_mp<='1';
157         wait for clk_period;
158         enter_mp<='0';
159         wait for clk_period;
```

```
161     --wait while coad is loaded
162     wait for 4*clk_period;
163     done_storing<='1';
164     wait for clk_period;
165     done_storing<='0';
166     wait for clk_period;
167
168     --enter a guess
169     enter_mp<='1';
170     wait for clk_period;
171     enter_mp<='0';
172     wait for clk_period;
173
174     --wait while guess is loaded
175     wait for 4*clk_period;
176     done_storing<='1';
177     wait for clk_period;
178     done_storing<='0';
179     wait for clk_period;
180
181     --wait while comparison occurs
182     wait for 19*clk_period;
183
184     --give a correct guess
185     correct_guess<='1';
186     wait for 1*clk_period;
187
188     --go to game over screen
189     enter_mp<='1';
190     wait for clk_period;
191     enter_mp<='0';
192     wait for clk_period;
193
194     --go to code entry screen
195     enter_mp<='1';
196     wait for clk_period;
197     enter_mp<='0';
198     wait for clk_period;
199     wait;
200 end process;
```

Figure E5: Simulation demonstrating Gameflow Controller works as intended



Appendix F

Figure F1: Constraints file

```
## This file is a general .xdc for the Basys3 rev B board
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

# Clock signal
#Bank = 34, Pin name = CLK,                 Sch name = CLK100MHZ
set_property PACKAGE_PIN W5 [get_ports mclk]
set_property IOSTANDARD LVCMOS33 [get_ports mclk]
create_clock -period 20.000 -name sys_clk_pin -waveform {0.000 10.000} -add [get_ports mclk]


#7 segment display
#Bank = 34, Pin name = ,                        Sch name = CA
set_property PACKAGE_PIN W7 [get_ports {seg[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[0]}]
#Bank = 34, Pin name = ,                    Sch name = CB
set_property PACKAGE_PIN W6 [get_ports {seg[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[1]}]
#Bank = 34, Pin name = ,                    Sch name = CC
set_property PACKAGE_PIN U8 [get_ports {seg[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[2]}]
#Bank = 34, Pin name = ,                        Sch name = CD
set_property PACKAGE_PIN V8 [get_ports {seg[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[3]}]
#Bank = 34, Pin name = ,                        Sch name = CE
set_property PACKAGE_PIN U5 [get_ports {seg[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[4]}]
#Bank = 34, Pin name = ,                        Sch name = CF
set_property PACKAGE_PIN V5 [get_ports {seg[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[5]}]
#Bank = 34, Pin name = ,                        Sch name = CG
set_property PACKAGE_PIN U7 [get_ports {seg[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[6]}]

#Bank = 34, Pin name = ,                        Sch name = DP
set_property PACKAGE_PIN V7 [get_ports {dp}]
set_property IOSTANDARD LVCMOS33 [get_ports {dp}]
```

```
#Bank = 34, Pin name = ,                          Sch name = AN0
set_property PACKAGE_PIN U2 [get_ports {an[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[0]}]
#Bank = 34, Pin name = ,                          Sch name = AN1
set_property PACKAGE_PIN U4 [get_ports {an[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[1]}]
#Bank = 34, Pin name = ,                          Sch name = AN2
set_property PACKAGE_PIN V4 [get_ports {an[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[2]}]
#Bank = 34, Pin name = ,                       Sch name = AN3
set_property PACKAGE_PIN W4 [get_ports {an[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[3]}]


##Buttons
set_property PACKAGE_PIN U18 [get_ports enter]
    set_property IOSTANDARD LVCMOS33 [get_ports enter]
set_property PACKAGE_PIN T18 [get_ports dig_up]
    set_property IOSTANDARD LVCMOS33 [get_ports dig_up]
set_property PACKAGE_PIN W19 [get_ports dig_down]
    set_property IOSTANDARD LVCMOS33 [get_ports dig_down]
set_property PACKAGE_PIN T17 [get_ports dig_right]
    set_property IOSTANDARD LVCMOS33 [get_ports dig_right]
set_property PACKAGE_PIN U17 [get_ports dig_left]
    set_property IOSTANDARD LVCMOS33 [get_ports dig_left]

##LEDS
set_property PACKAGE_PIN W3 [get_ports red_LED_0]
    set_property IOSTANDARD LVCMOS33 [get_ports red_LED_0]
set_property PACKAGE_PIN U3 [get_ports red_LED_1]
    set_property IOSTANDARD LVCMOS33 [get_ports red_LED_1]
set_property PACKAGE_PIN P3 [get_ports red_LED_2]
    set_property IOSTANDARD LVCMOS33 [get_ports red_LED_2]
set_property PACKAGE_PIN N3 [get_ports red_LED_3]
    set_property IOSTANDARD LVCMOS33 [get_ports red_LED_3]
set_property PACKAGE_PIN P1 [get_ports red_LED_4]
    set_property IOSTANDARD LVCMOS33 [get_ports red_LED_4]

```

```
78  set_property PACKAGE_PIN U16 [get_ports white_LED_0]
79      set_property IOSTANDARD LVCMOS33 [get_ports white_LED_0]
80  set_property PACKAGE_PIN E19 [get_ports white_LED_1]
81      set_property IOSTANDARD LVCMOS33 [get_ports white_LED_1]
82  set_property PACKAGE_PIN U19 [get_ports white_LED_2]
83      set_property IOSTANDARD LVCMOS33 [get_ports white_LED_2]
84  set_property PACKAGE_PIN V19 [get_ports white_LED_3]
85      set_property IOSTANDARD LVCMOS33 [get_ports white_LED_3]
86  set_property PACKAGE_PIN W18 [get_ports white_LED_4]
87      set_property IOSTANDARD LVCMOS33 [get_ports white_LED_4]
88
89
90
91
92
93  set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
94  set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]
95  set_property CONFIG_MODE SPIx4 [current_design]
96
97  set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
98
99  set_property CONFIG_VOLTAGE 3.3 [current_design]
100 set_property CFGBVS VCCO [current_design]
```