WORDLE REPLICA

Engs. 31: Digital Electronics

Final Project

GitHub Repository

Demo Video (Requires Dartmouth Account for Access)

Ikeoluwa Abioye, Lobna Jbeniani, Mubarak Idoko

May 2022

Project Summary

In the past year, WORDLE has become one of the most played online games, with over 300,000 users in the first two months of its launch and over 2,000,000 users today. Given that WORDLE started out with less than 90 players in the fall of 2021, the fast-paced growth of the game has incentivized many to recreate it in one way or another. In this final report, our group describes how we built a WORDLE clone using problem-solving techniques that we learned in ENGS31: Digital Electronics.

We will describe our design process and how we eventually built the WORDLE replica using high-level block diagrams, Register-Transfer Level (RTL) diagrams, a top-level Finite State Machine (FSM), and eventually, VHDL code and Vivado Simulation. In addition to this, we have a demonstration video to show our version of WORDLE working on FPGA hardware.

Table of Contents

Introduction

First designed and engineered by Josh Wardle, WORDLE has become a household name in a short amount of time. In this paper, we report on how we replicated WORDLE using the concepts and tools of ENGS31: Digital Electronics.

To better understand this paper, it's critical to revisit the rules of the game. In short, the game allows the user 6 trials to guess a 5 word letter. The entered words have to be in the dictionary, which contains over 11,500 words or else the program will not accept them. When a user inputs a word, the program first ensures that each of the entered characters is a letter, then once the user has finished typing all 5 characters and hits the ENTER key, the program checks if the word is in the WORDLE dictionary. At the next stage, the program evaluates each character individually. If the letter is in the correct spot, the program highlights in green. If the letter is in the word but not in the correct spot, the program highlights it in yellow. If the letter is not in the word at all, the program highlights it in gray. The user uses these instructions to inform his or her decision on the next word choice. This process repeats 6 times for each entered word. Two other considerations are that letters can be used more than once (e.g., fleet, green, abbey..etc) and answers are never plural.

In this replica, we uphold the above principle of the game. However, given that we are using a Putty Teletype as our game interface, we have slightly altered the coloring of the different letters. We display a '?', question mark, if an entered character is in the word, but in the wrong location; the character, itself, if it is in the word and in the right location, and a blank space if an entered character is not in the correct word. In essence, when a user inputs a letter that exists in the answer word but they input it in an incorrect spot, they see a '?' at that spot. When they input a letter that does not exist in the answer word at all, they see a blank space at that location. And when the user inputs a letter that exists in the answer word at a spot matching its corresponding location in the answer word, they get that character back in that location. For invalid words, those are words that are not in the dictionary, the user simply gets nothing in response. The figure below illustrates this user interface in more detail. Importantly, our entire interface is built using lowercase ascii characters.

A Visual Demonstration of the User Interface Differences between WORDLE and our WORDLE REPLICA

Step 1: The user gets 6 guesses, both in the original game and our replica.

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

Step 2: The user types a word—in our replica, the user hits ENTER right after entering the word.

D R I N K

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

Step 3: The word gets evaluated (below is a comparison of what the user might see in the actual game versus our replica).

D R I N K

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

D R I N K

D ?

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

A Visual Demonstration of the User Interface Differences between WORDLE and our WORDLE REPLICA

Step 1: The user gets 6 guesses, both in the original game and our replica. The solution is FLASK.

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

Step 2: The user enters a word.

G A S S Y

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

Step 2: The word gets evaluated (below is a comparison of what the user might see in the actual game versus our replica). As the figure shows, the first S shows up in gray in the game while the second one shows up as green. So, in the case that we have a repeated letter and the user types it in the correct spot, it turns green where the user wrote it, while the other letter takes the color gray. To emulate this behavior, we show a question mark and the letter itself in the correct spot.

G A S S Y

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

G A S S Y

? S

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

_ _ _ _ _

Methods

I. Methodology Description

1. Deconstructing the Problem: High Level Block Diagram

We started out by designing a high level block diagram, which allowed us to visualize the different functional blocks necessary to have a fully working program. The diagram consisted of three main functional blocks:

a. A Serial Communications Interface **(SCI) Receiver**

b. A **SCI Transmitter**:

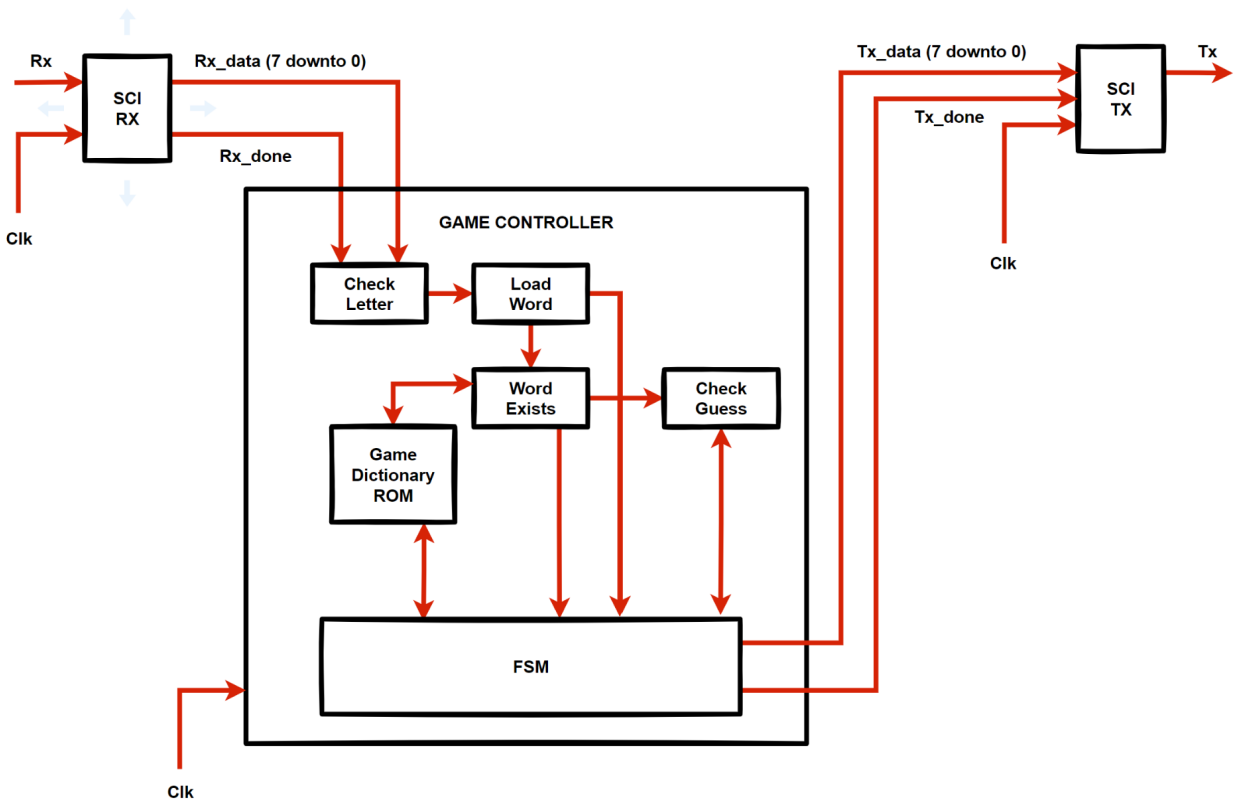
c. The **Game Controller**: The game logic is the block that runs the game. In other words, now that the program has received user input via the SCI Receiver and is ready to output data via the SCI Transmitter, the only functional block left to think about is the one controlling the game itself. Within this parent block we have 5 sublevel block: **Check Letter, Load Word, Word Exists, Check Guess.** These blocks interact with the **Game Dictionary ROM**, that contains all the valid wordle words, finally, we have a **Top Level FSM** that listened for signal from these blocks, and send signals to the Transmitter on what to output to the use.

With this logic, we could create a game flow for our wordle:

- When the user enters a character, **Check Letter** checks the nature of the signal that is, whether is a valid ASCII alphabetic character, a backspace, or a delete character and sends a signal to **Load Word**. If it is valid ASCII, it normalizes the character to lowercase and send it into **Lord Word.**
- **Load Word** keeps an array of characters, so when a new character comes in, it saves it, and send a signal to the **FSM**, so that that character can be sent out. If it receives a backspace character, is deletes the most recent character and send a signal so that the **FSM** can tell the **SCI Transmitter** to do the same on the teletype. It only tells the **FSM** to trigger sending an enter when the word is full, that is, it has 5 valid alpha characters. Then, it also passed the word to **Word Exists.**

- **Word Exists** checks that the word is a valid dictionary word. If the word is invalid, it tells the **FSM** to send an enter so that the user can enter a different word. Otherwise, it passes the word to **Check Guess.**
- **Check Guess** check the guess word against the solution word that comes from the **FSM** and computes two vectors representing the output for the current guess. It sends this to the **FSM** and the **FSM** uses the vectors to figure out what to tell the transmitter to send. That is ‘an alphabetic character’, ‘a question mark’, or ‘a blank space’ depending how the users guess matches the solution word.
- **Game Dictionary ROM** holds all the words in the game. It is communicates with **Word Exists** to check if a word is in the dictionary, and with the **FSM** to select a solution word for the game.
- **The FSM** keeps track of all the states in the game and uses control signals to determine what the transmitter should send. It also keeps a continuously going counter that selects the solution words for each game round, and a count of the number of words entered during a game round.
- Importantly, the clock goes to all blocks, apart from the check letter block.

This design led to the following high level block diagram:



2. **Top Level Design: Block Diagrams**

   Once we had a top level, we could design each separate unit details for each of these blocks is describes in the block diagram section below: Block Diagrams

3. **Functionality: RTL Diagrams**

   The RTL designs for each block are in this section below:

4. **Designing the Process: The Finite State Machine**

   This is the controller that makes the game work. Details about the design and iteration process are detailed in the finite state machine section below:

5. **Building and Connecting the Dictionary: Game Dictionary ROM**

   We accomplished this process by first converting the WORDLE dictionary, which contains 12,972 words, into ASCII binary code. Building and connecting the WORDLE dictionary was done through the following steps:

   - Importing the dictionary onto an EXCEL file
   - Converting all letters in the file into lower case letters
   - Splitting the rows of words into 5 column, each containing one letter
   - Converting each of the letters into ASCII decimal numbers (e.g., a is 97, b is 98..etc). Since everything was lower case, the numbers were between 97 and 122
   - Converting each decimal number into binary code. This gave us 40 bits of data for each word
   - Concatenating the columns to form one single column containing all 40 bits for each word
   - Uploading the rows of binary data into a COE file

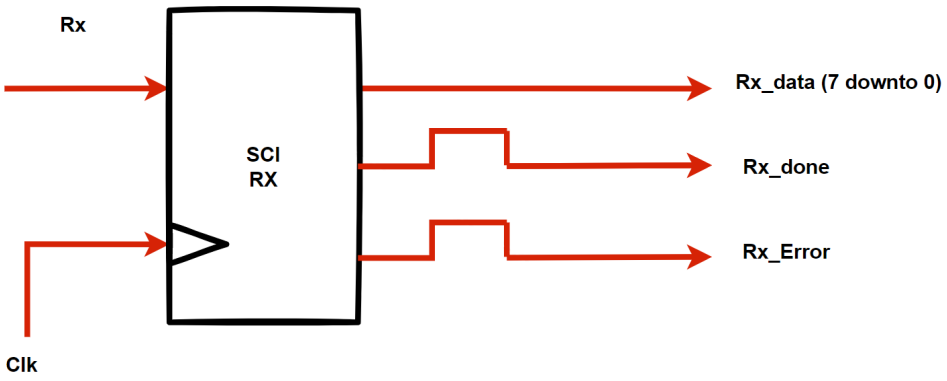
   Finally, we used the IP cores in Vivado to create a block memory unit and we uploaded the COE file to create the **Game Dictionary ROM.**

II. Block Diagrams

A. The SCI Receiver:

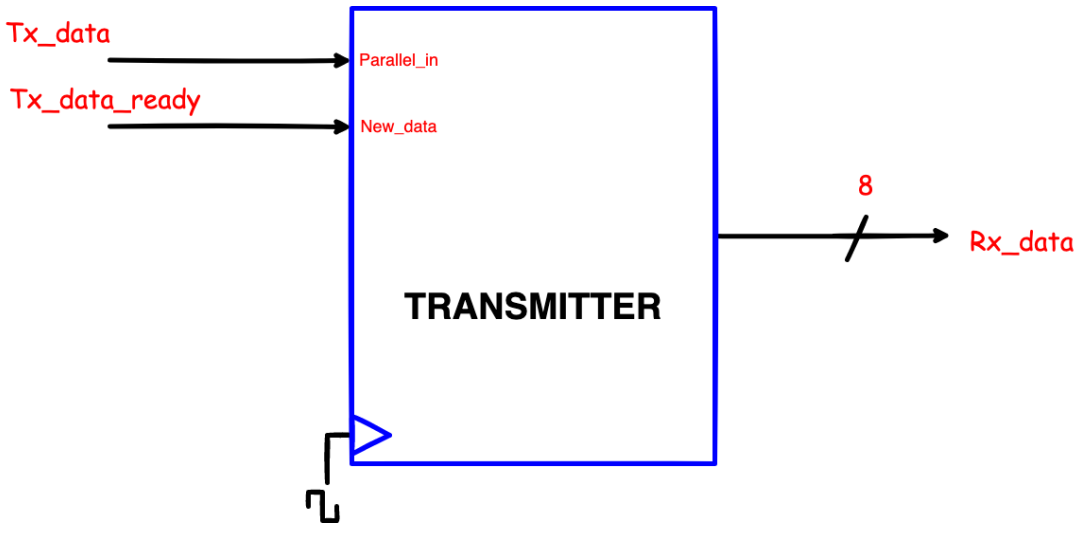
The SCI Receiver takes in user input over a serial line **Rx**, loads the bits into a shift register and afterward outputs signals: **Rx_data**: the 8-bit ASCII character that was entered, **Rx_done (monopulse)**: signifying that the character is ready, **Rx_error (monopulse)**: that goes high if there was error in receiving. The final block diagram was:

Figure 2: High-level Workflow Block Diagram of WORDLE REPLICA



B. The SCI Transmitter:

The transmitter takes in 8-bits of data to transmit as parallel input, **Parrallel_In**, as well as a **New_Data, monopulse,** and it shifts out the bits one by one on the **Tx** output line.

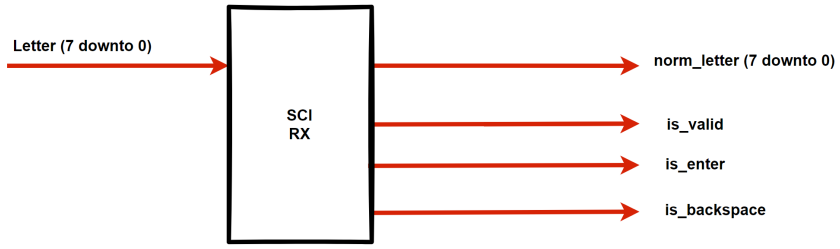


C. The Game Controller:

For the five blocks in the game controller, we had the following block diagrams:

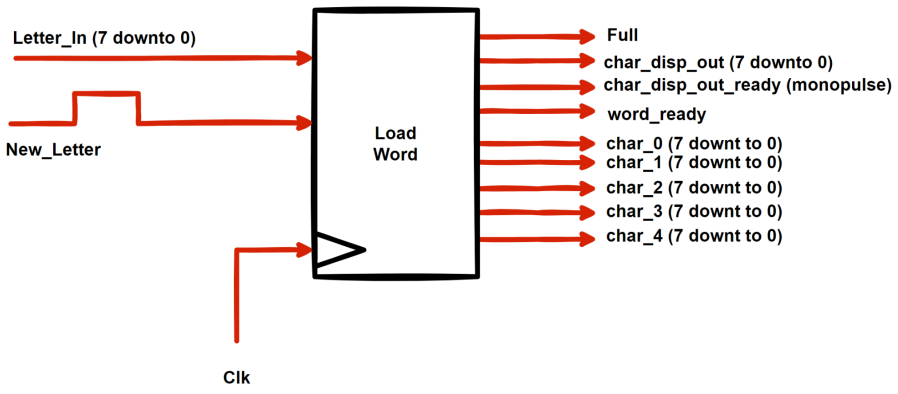
- **Check Letter:**

  This block takes in the a **Letter** that has been entered by the user. It evaluates it and outputs signals depending on whether it is a valid alphabetic character, **is_valid**, and then it outputs the lowercase version of the word on the **norm_letter** line. Similarly, if the character is an enter character, in this case, carriage return, **is_enter**, goes high, and if it is a backspace character, **is_backspace** goes high.



- **Load Word:**

  The **Load Word** block takes in a letter when the **New_Letter, monopulse**, goes high. The outputs **Full**, goes high when we have loaded in 5 valid characters, **char_disp_out, 8-bits**, is the signal to the **FSM** on what character to send out when **char_disp_out_ready, monopulse**, goes high. **word_ready, monopulse**, goes high when we have loaded in 5 valid characters, and the 8-bit output **char_0** to **char_4** hold the characters for the 5-letter word.

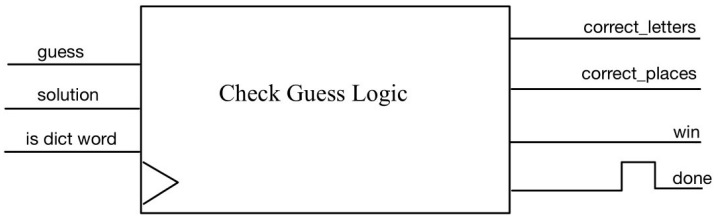


- **Word Exists:**

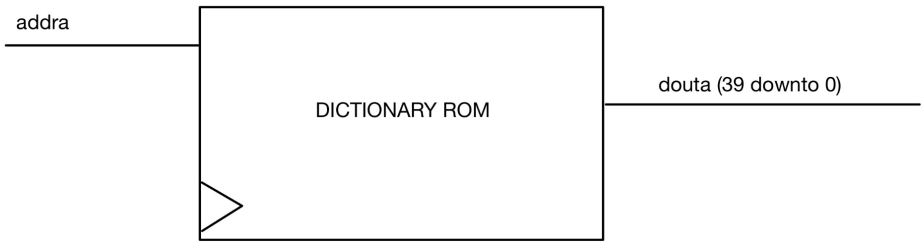
  Word Exists takes in the users **guess,** and a signal, **word_ready, monopulse**, that says the guess is ready to be evaluated. It checks if the word is in the dictionary and then set **is_dict_word** and **not_in_dict** accordingly.

- **Check Guess:**

  Guess is the users guess, while **Solution** is the game word. This block takes in the **Guess** once we validate is as a word in the dictionary. Then it outputs the **correct_letter(5 downto 0)** and **correct_place(5 downto 0)** vector. For correct letters, the output at an index is **1** if the letter at that index in the guess word is the same as in the solution word, while correct place goes to **1** if the letters are in the correct positions. **Win** goes high if the guess matches the game word and **Done** is a monopulse to let the **FSM** know that the guess evaluations has been finished.



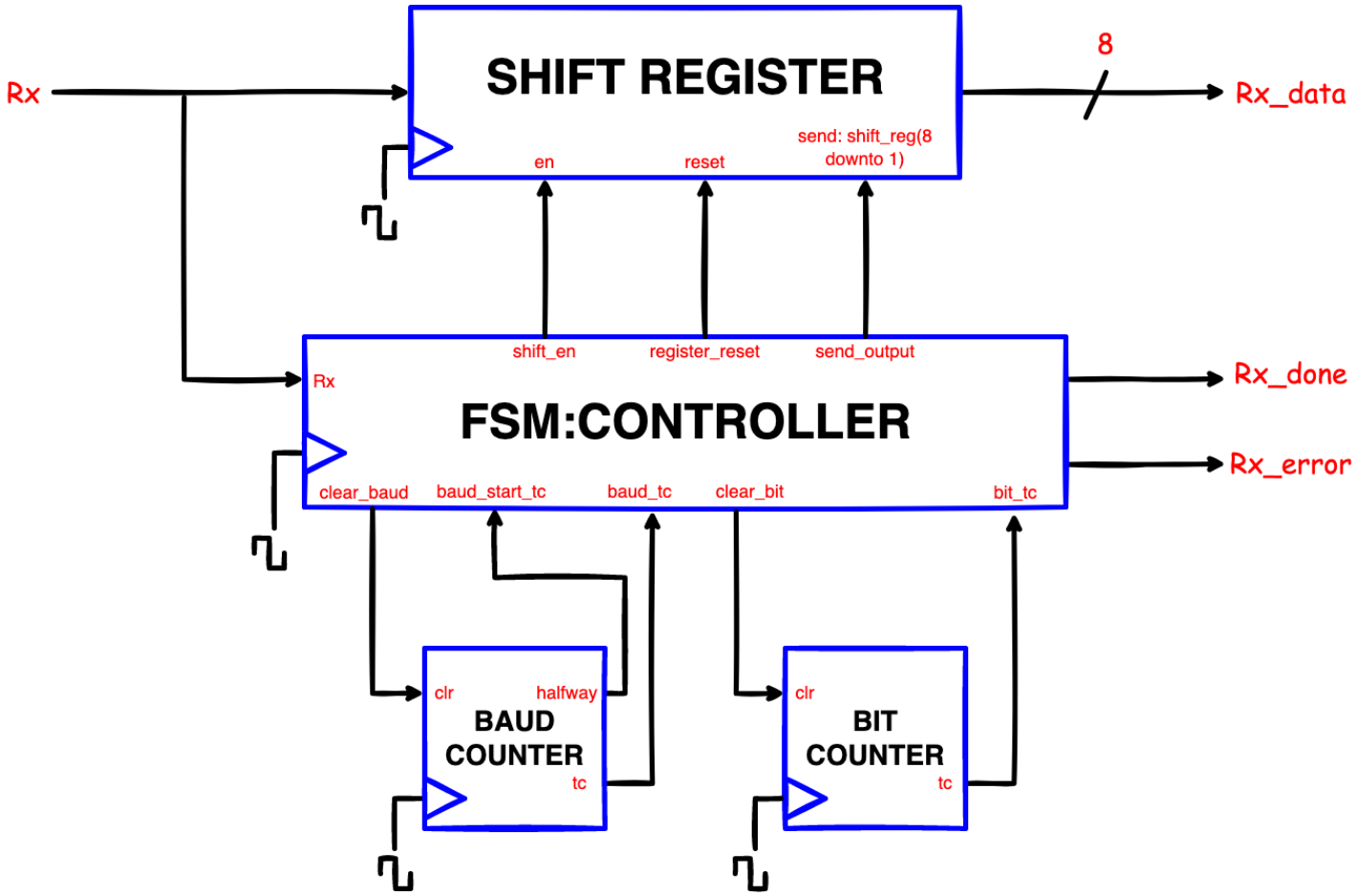
- **Game Dictionary ROM:** The purpose of this block is to hold read only memory (ROM) of all 12,972 WORDLE words in ASCII binary code. Each letter consists of 8 bits of code, and each word consists of 40 total bits.



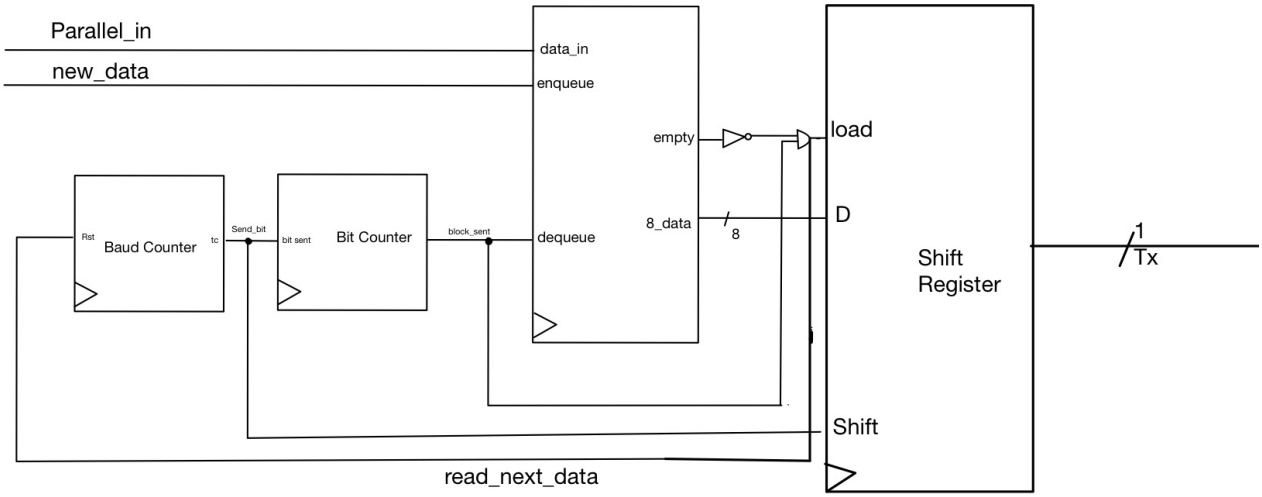
III. Register Transfer Level (RTL) Diagrams

We translated each block to RTL Diagrams and use those to write the code for each block. Beside each block is a link to well-documented code for that block.

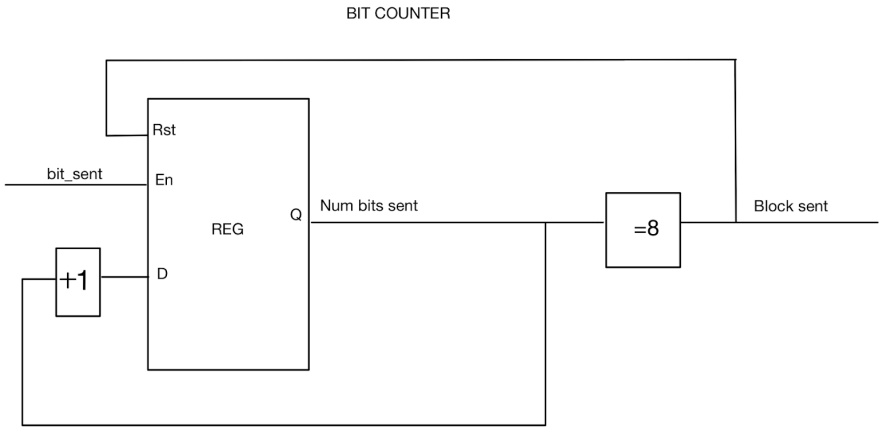
A. SCI Receiver: sci_receiver.vhd



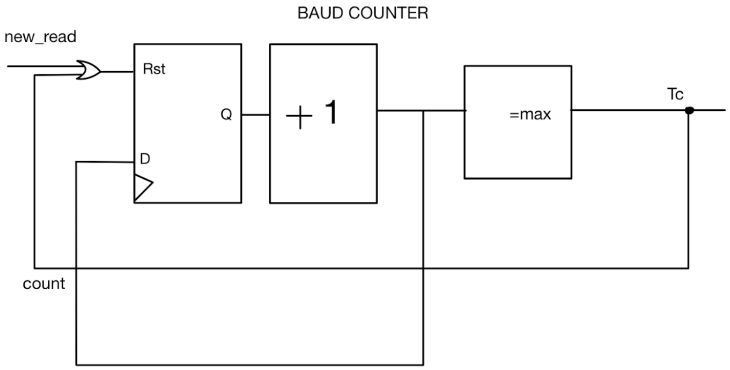
B. SCI Transmitter: sci_transmitter.vhd



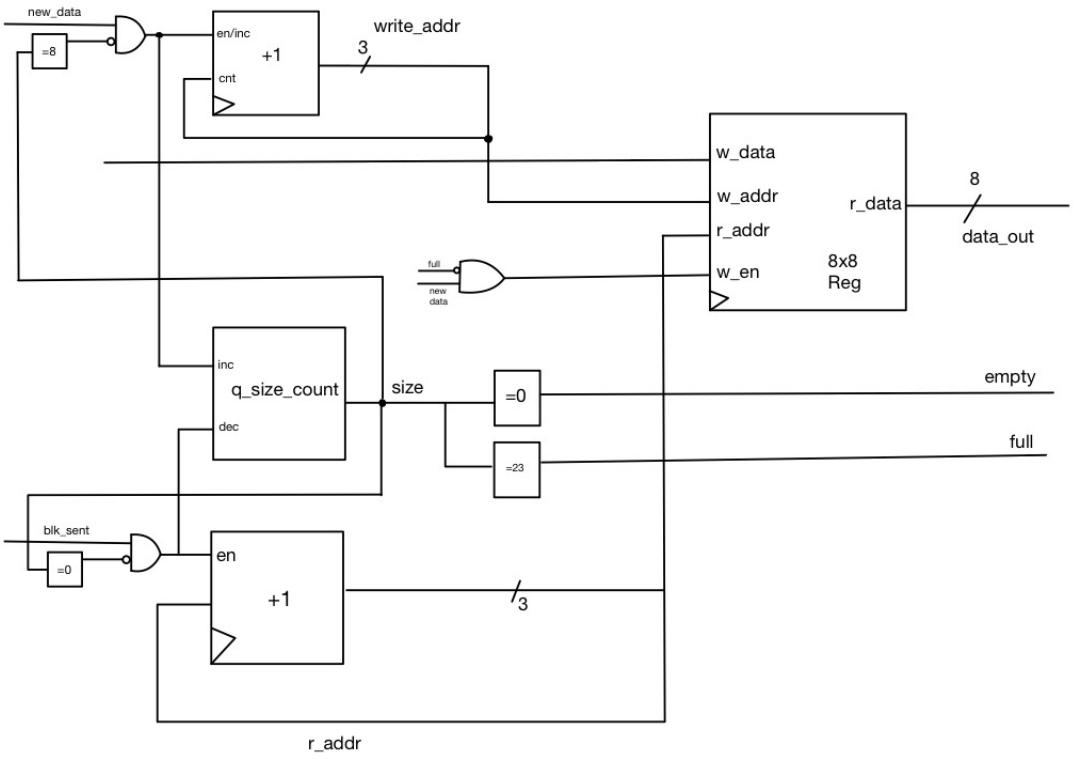
a. Bit Counter:



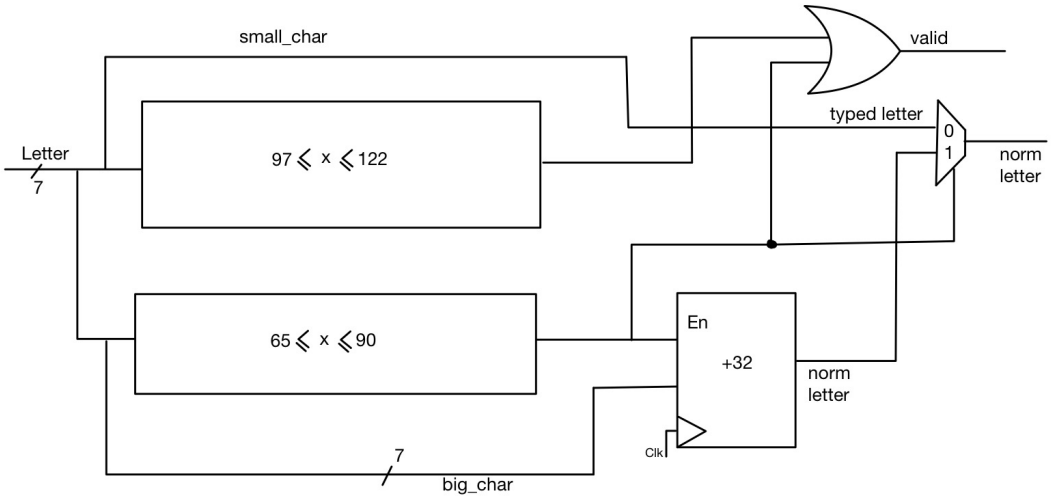
b. Baud Counter:



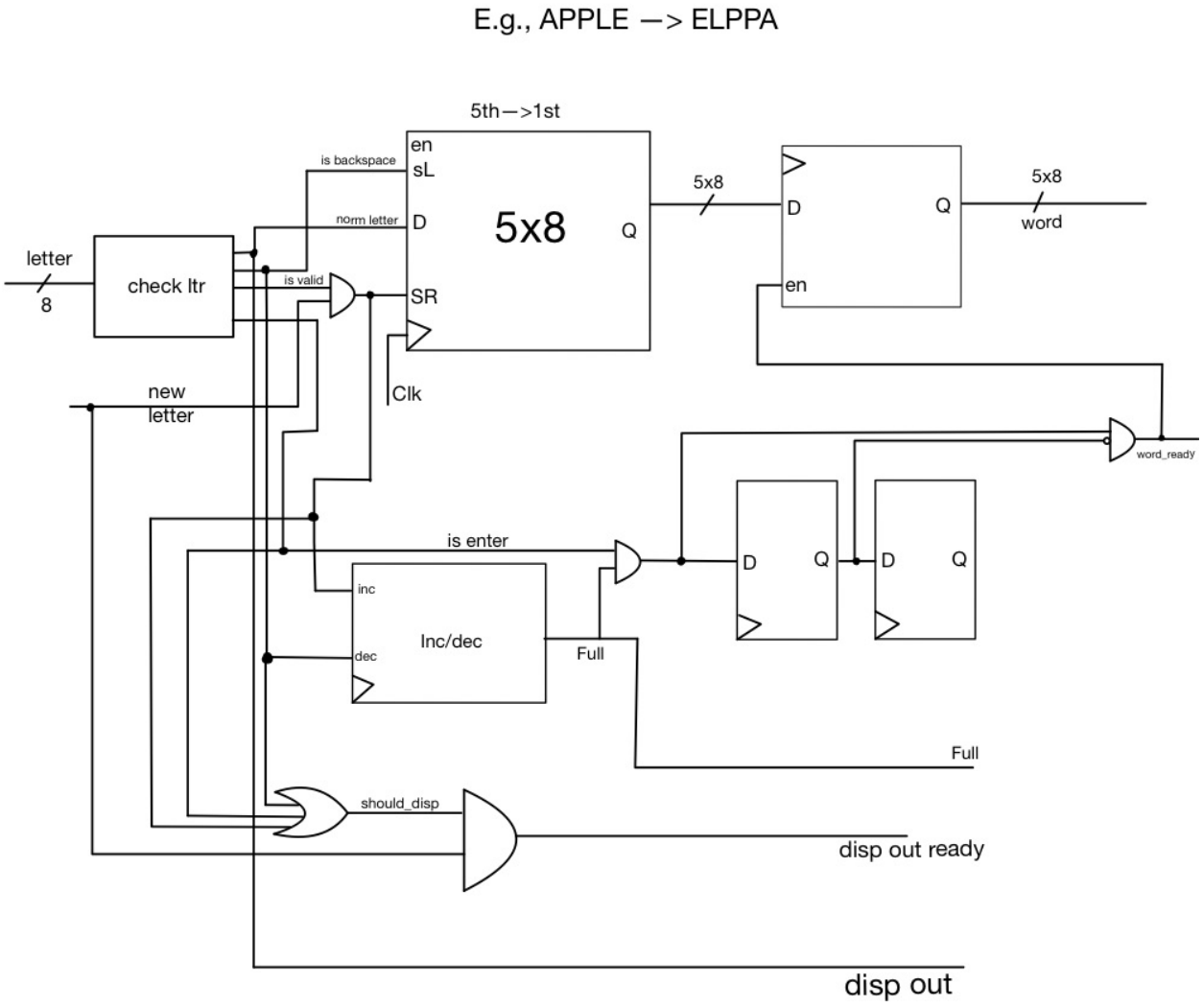
c. Queue:



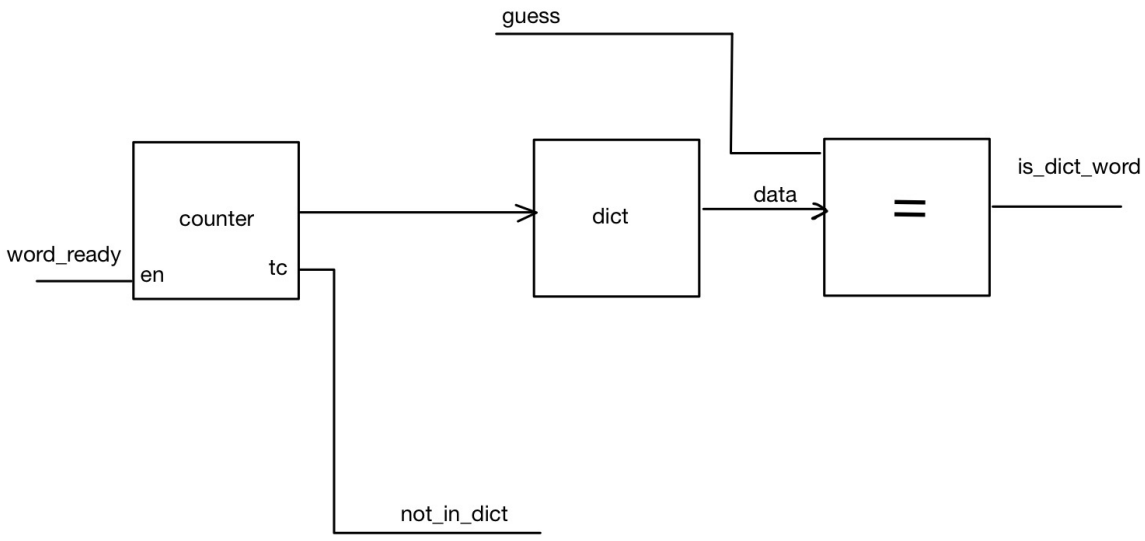
C. Check Letter: check_letter.vhd



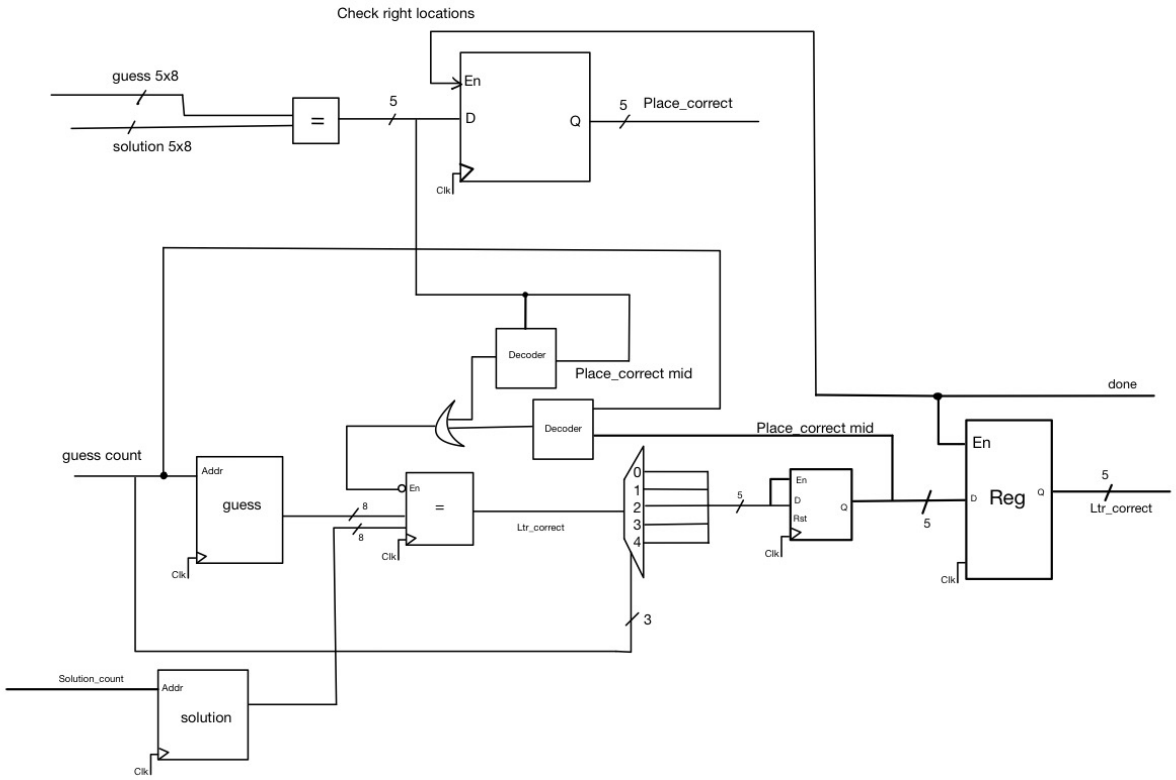
D. Load Word: load_word.vhd



E. Word Exists: word_exists.vhd



F. **Check Guess**: check_guess.vhd

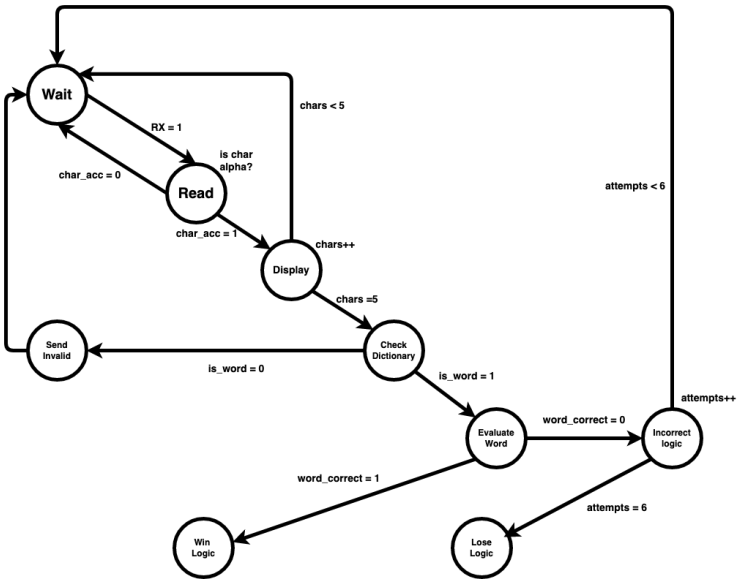


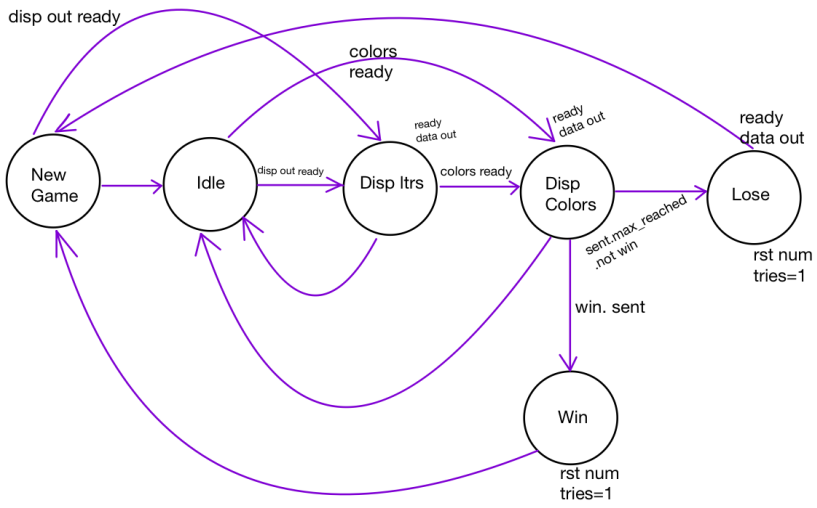
IV. Finite State Machine (FSM)

As we continued to work on the project, the FSM gradually evolved.

Initial Model

Our first model focused more on the initial stages of the game, including the reading, display and checking if a word is in the dictionary.

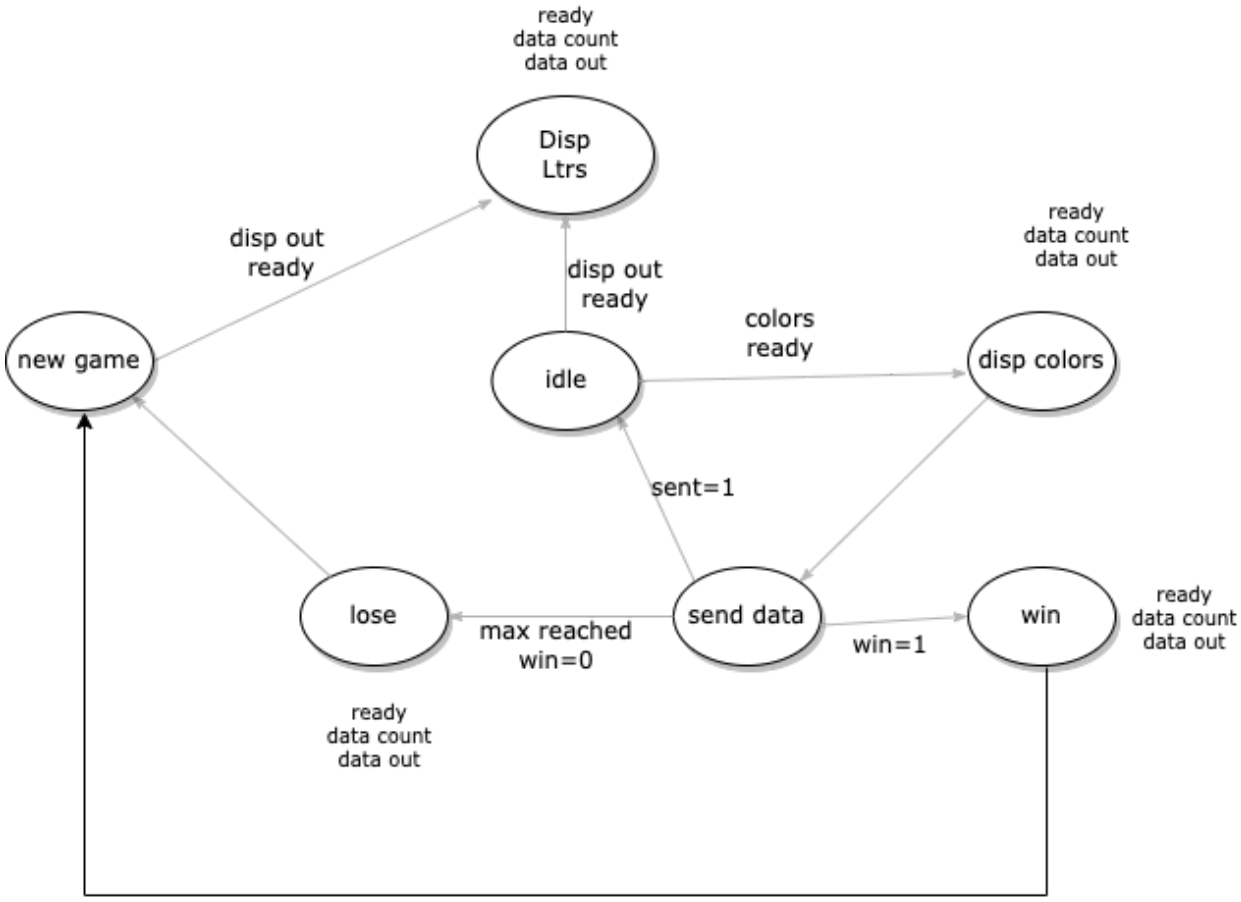




Final FSM Model

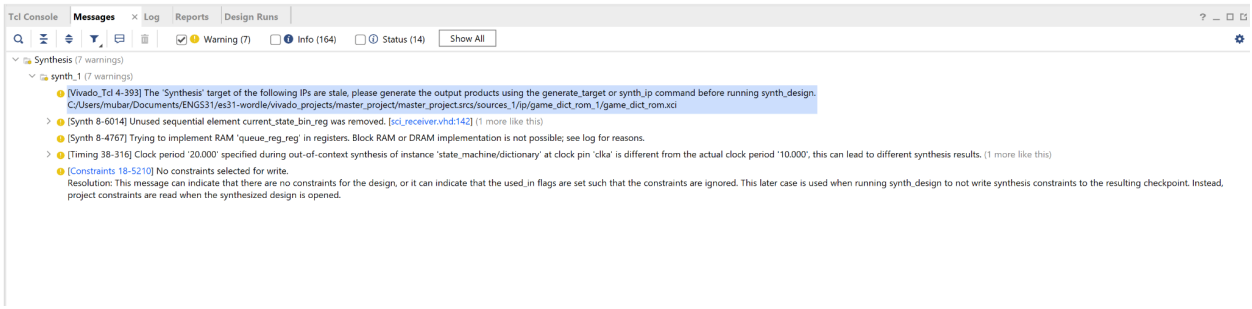
Our final model combined properties from the past two models, which allows us to have more of a comprehensive process without getting lost in states that are unnecessary and avoids timing issues. The game is initially in the new_game state and when a character that can be displayed comes in (disp_out_ready goes high), the system chooses the current dictionary input as the solution signal and proceeds with the game. The new_game state can only be reached thereafter after the win or lose state has been reached.

The send_data state is used when we have reached a state where data needs to be sent out through the transmitter. When the other states are reached, they send control signals to store what they want to send in the data_to_send register. This send_data state then decides how many bytes of the data_to_send register to "shift out" to the transmitter.



v. VHDL Code Design and Test Benching: Noteworthy Errors

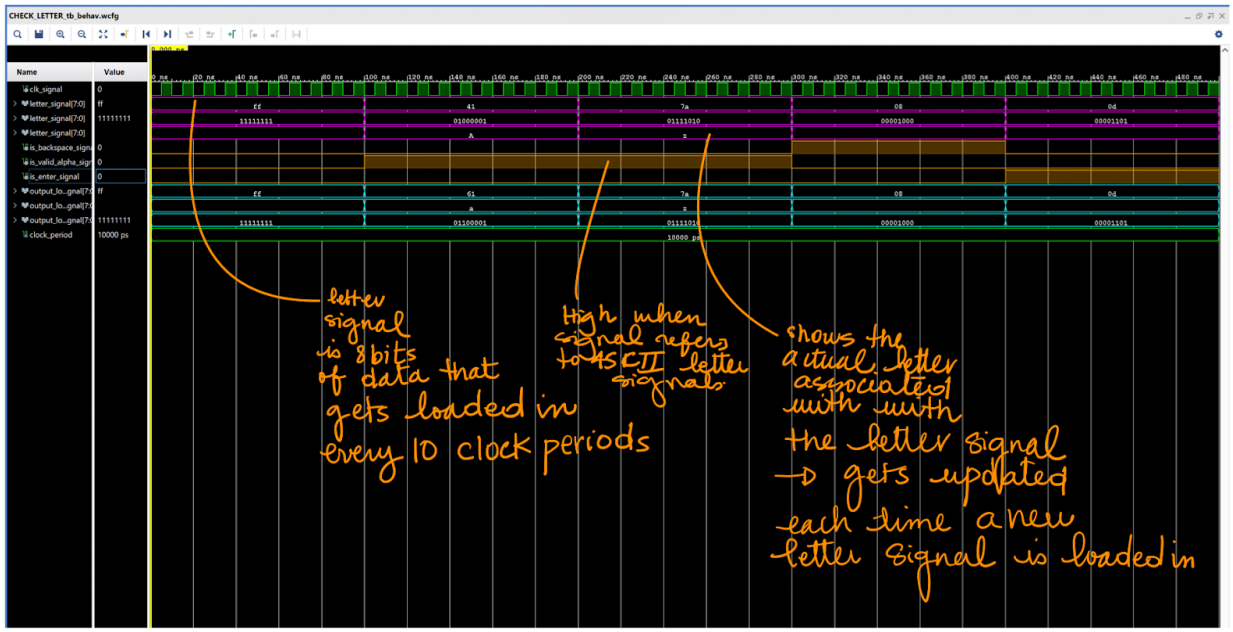
We have no errors and no noteworthy Warning, as shown in the screenshot below:



VI. Results

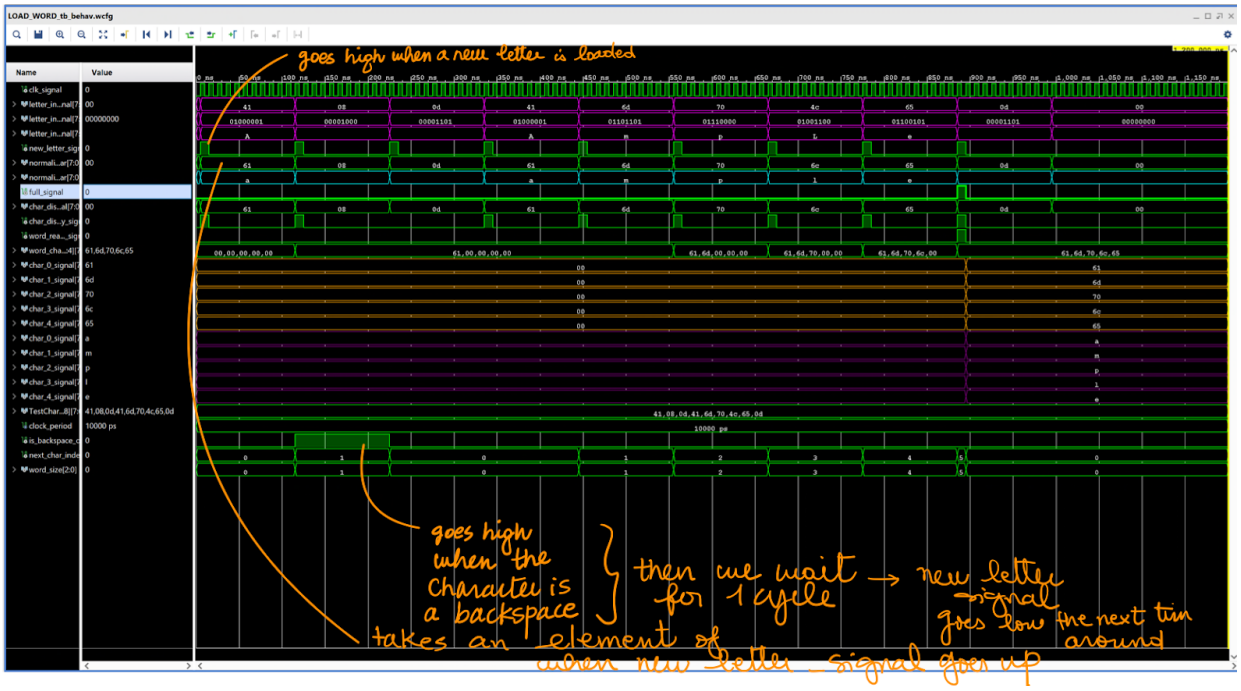
A. Check Letter Simulation: check_letter_tb.vhd,

Detailed descriptions can be found at check_letter_testing_plan.txt



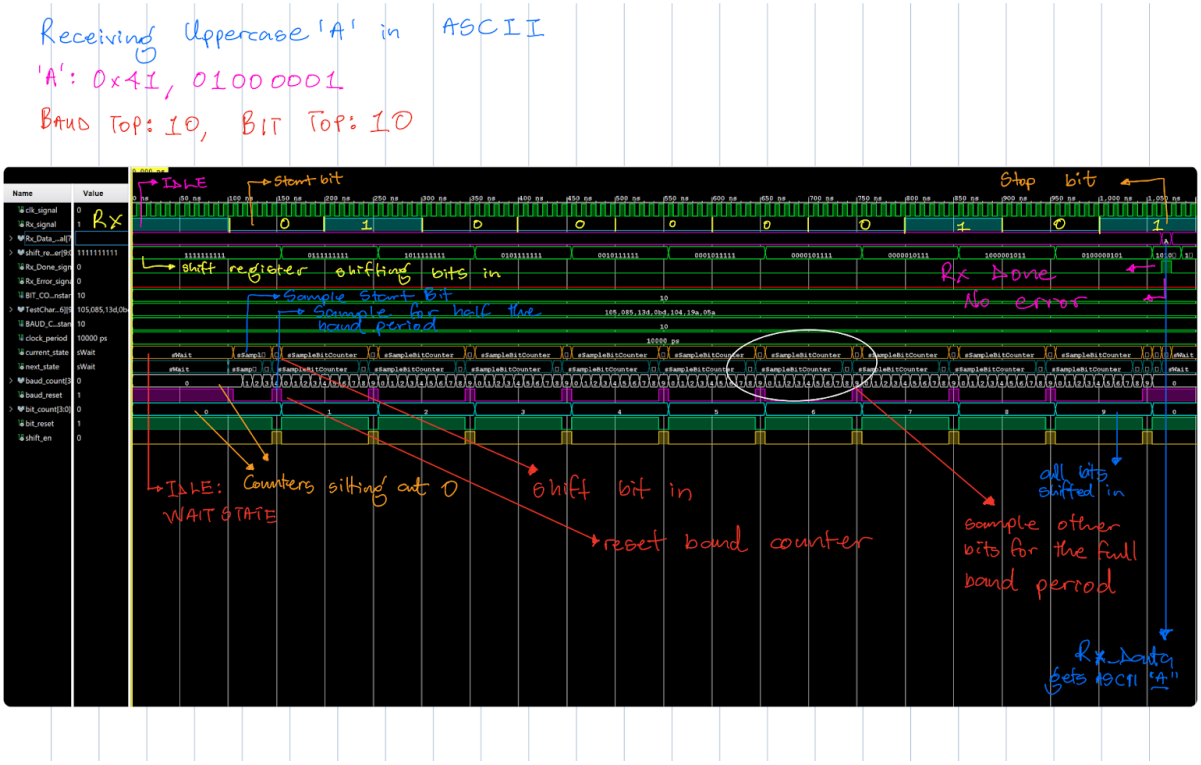
B. Load Word Simulation: load_word_tb.vhd,

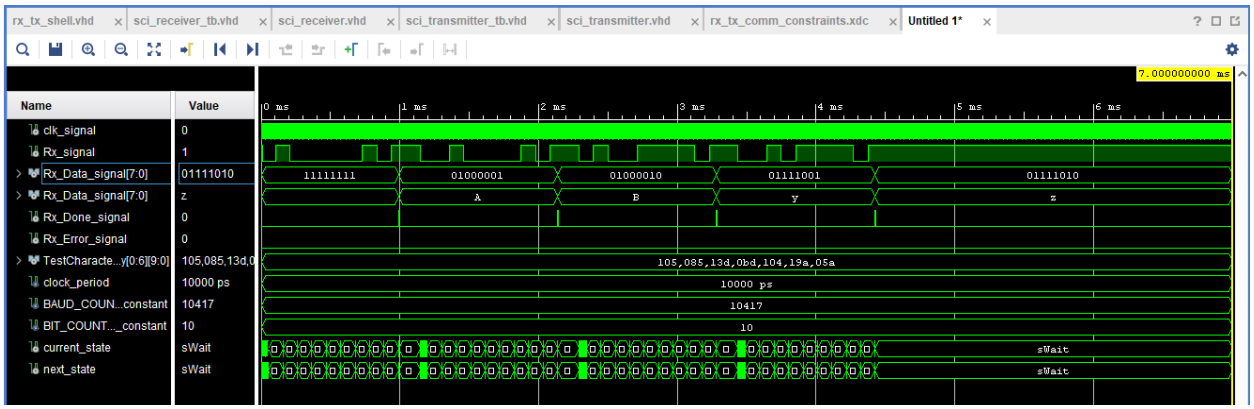
Detailed descriptions can be found at load_word_testing_plan.vhd

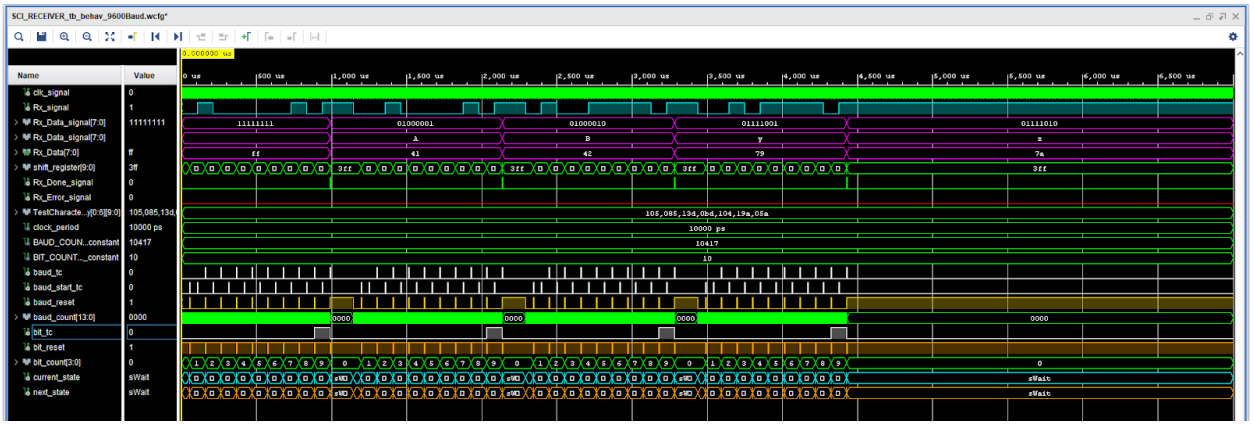


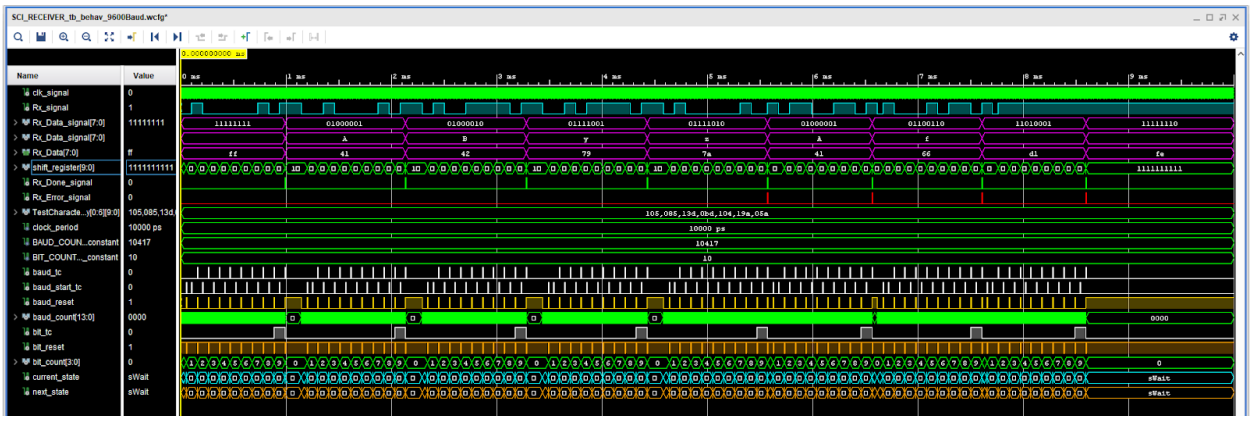
C. Receiver: receiver_tb.vhd

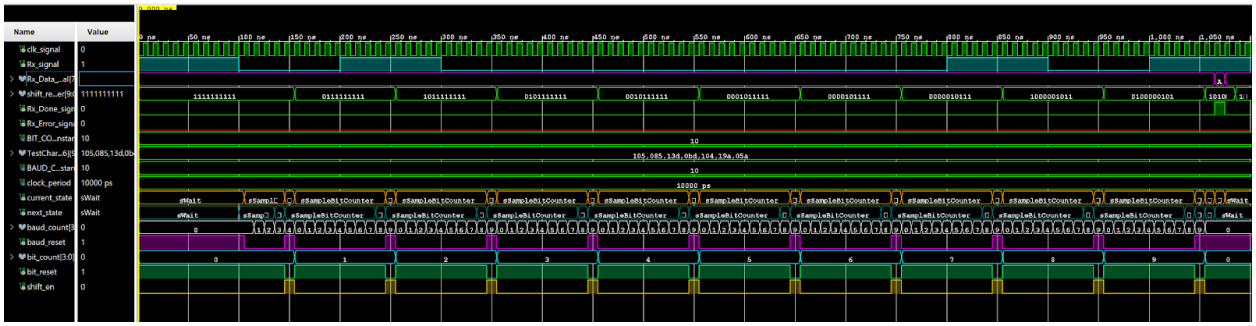
Detailed descriptions can be found at receiver_testing_plan.txt




rx_tx_shell.vhd
sci_receiver_tb.vhd
sci_receiver.vhd
sci_transmitter_tb.vhd
sci_transmitter.vhd
rx_tx_comm_constraints.xdc
Untitled 1*
Name
Value
clk_signal
Rx_signal
Rx_Data_signal[7:0]
Rx_Data_signal[7:0]
Rx_Done_signal
Rx_Error_signal
TestCharacte...y[0:6][9:0]
clock_period
BAUD_COUN...constant
BIT_COUNT..._constant
current_state
next_state
7.000000000 ms
11111111
01000001
01000010
01111001
01111010
A
B
y
z
105,085,13d,0bd,104,19a,05a
10000 ps
10417
10
sWait

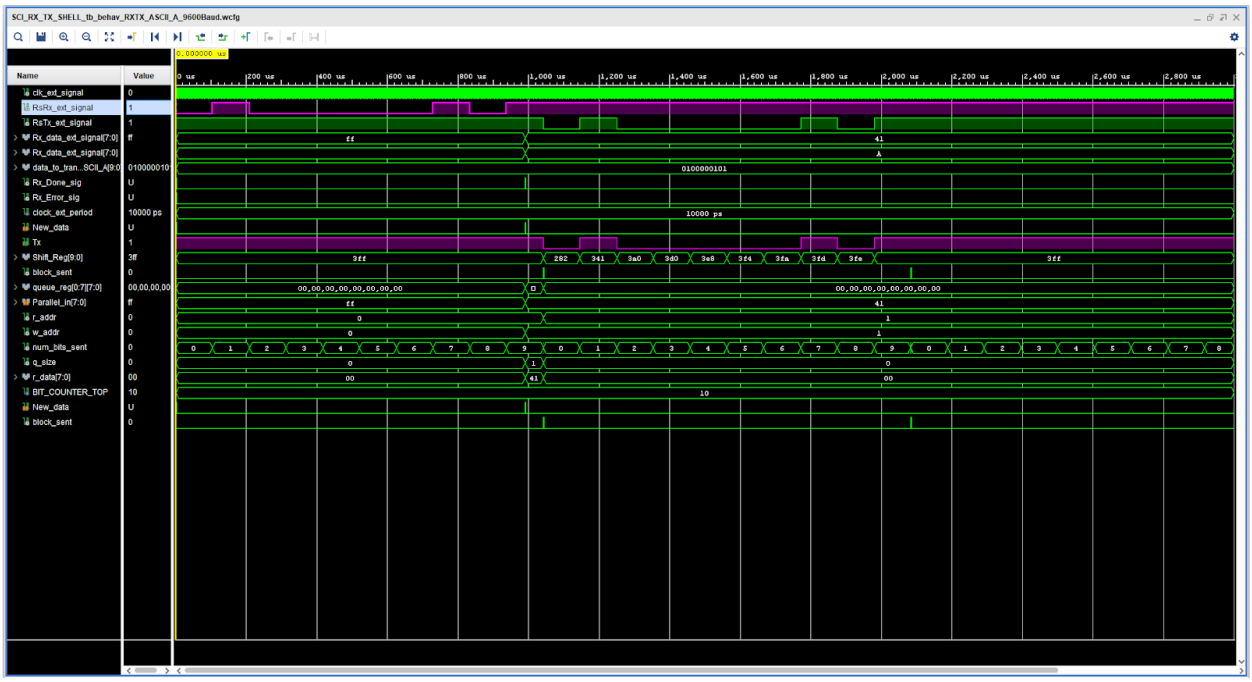

SCI_RECEIVER_tb_behav_9600Baud.wcfg*
Name
Value
clk_signal
Rx_signal
Rx_Data_signal[7:0]
Rx_Data_signal[7:0]
Rx_Data[7:0]
shift_register[9:0]
Rx_Done_signal
Rx_Error_signal
TestCharacte...y[0:6][9:0]
clock_period
BAUD_COUN...constant
BIT_COUNT..._constant
baud_tc
baud_start_tc
baud_reset
baud_count[13:0]
bit_tc
bit_reset
bit_count[3:0]
current_state
next_state
11111111
01000001
01000010
01111001
01111010
A
B
y
z
105,085,13d,0bd,104,19a,05a
10000 ps
10417
10
sWait


SCI_RECEIVER_tb_behav_9600Baud.wcfg*
Name
Value
clk_signal
Rx_signal
Rx_Data_signal[7:0]
Rx_Data_signal[7:0]
Rx_Data[7:0]
shift_register[9:0]
Rx_Done_signal
Rx_Error_signal
TestCharacte...y[0:6][9:0]
clock_period
BAUD_COUN...constant
BIT_COUNT..._constant
baud_tc
baud_start_tc
baud_reset
baud_count[13:0]
bit_tc
bit_reset
bit_count[3:0]
current_state
next_state
11111111
01000001
01000010
01111001
01111010
01000001
01100110
11010001
11111110
A
B
y
z
A
f
105,085,13d,0bd,104,19a,05a
10000 ps
10417
10
sWait


clk_signal
Rx_signal
shift_register[9:0]
Rx_Done_signal
Rx_Error_signal
BIT_COUNT..._constant
TestCharacte...y[0:6][9:0]
BAUD_COUNT...constant
clock_period
current_state
next_state
baud_count[13:0]
baud_reset
bit_count[3:0]
bit_reset
shift_en
105,085,13d,0bd,104,19a,05a
10000 ps
sWait
sSampleBitCounter

D. Receiver Transmitter: rx_tx_tb.vhd,

Detailed descriptions can be found at rx_tx_testing_plan.vhd

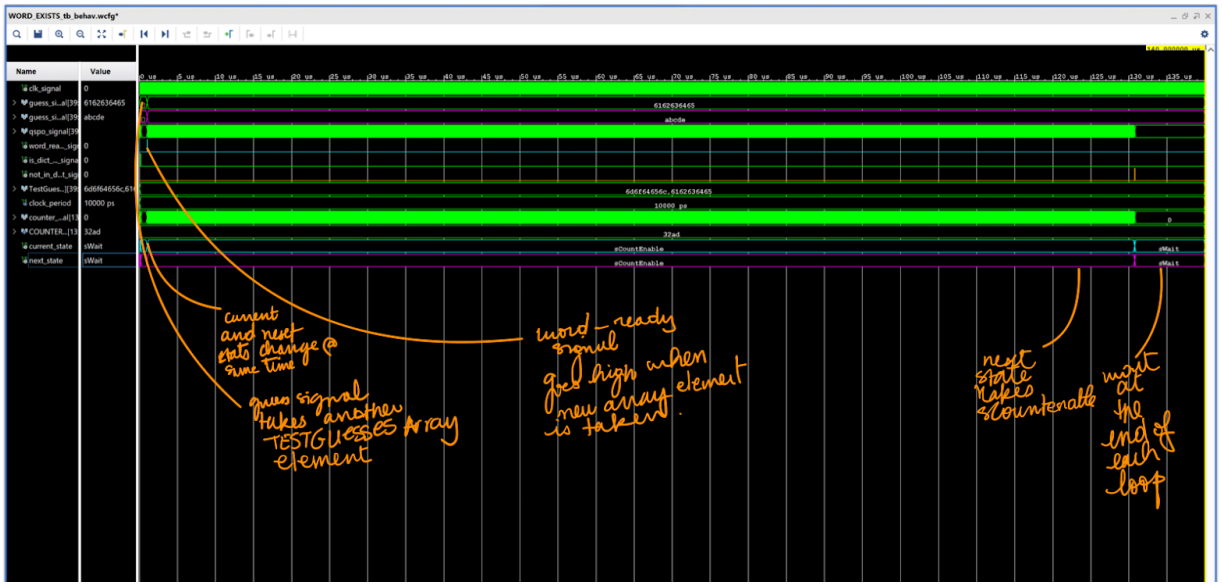


E. Check Uppercase Letter (Receiver Transmitter) check_letter_tb.vhd,

Detailed descriptions can be found at check_letter_testing_plan.txt

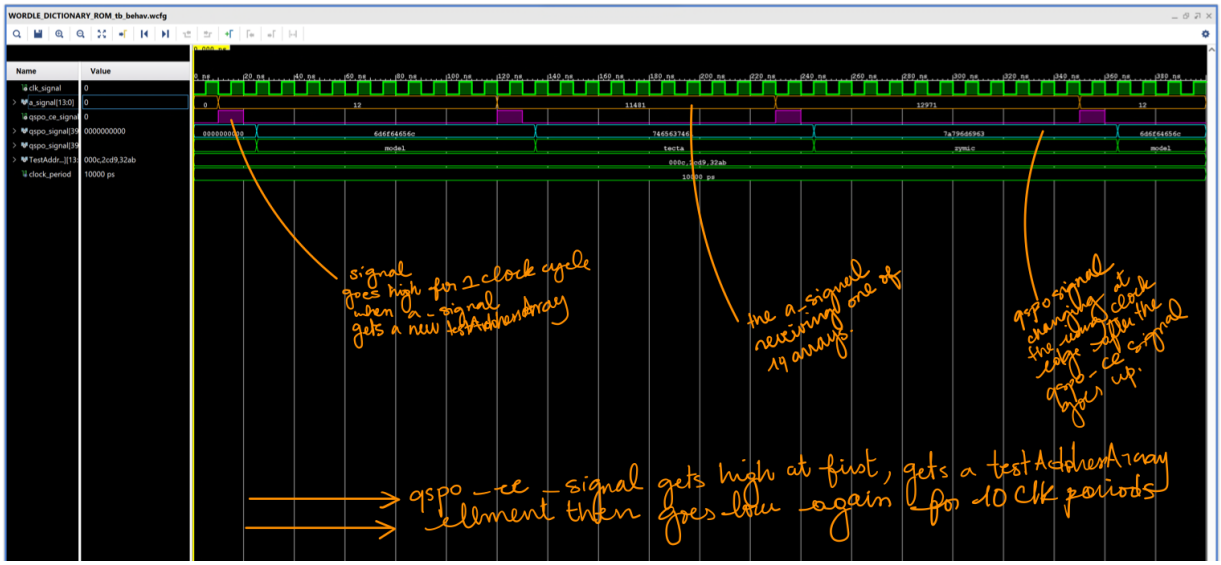
F. Word Exists: word_exists_tb.vhd

Detailed descriptions can be found at word_exists_testingplan.txt



G. WORDLE Dictionary ROM: [wordle_dictionary_rom_tb.vhd]

Detailed descriptions can be found at [dictionary_rom_testingplan.txt]



VII. Behavior in Hardware

The behavior is as described and expected.

VIII. Conclusion

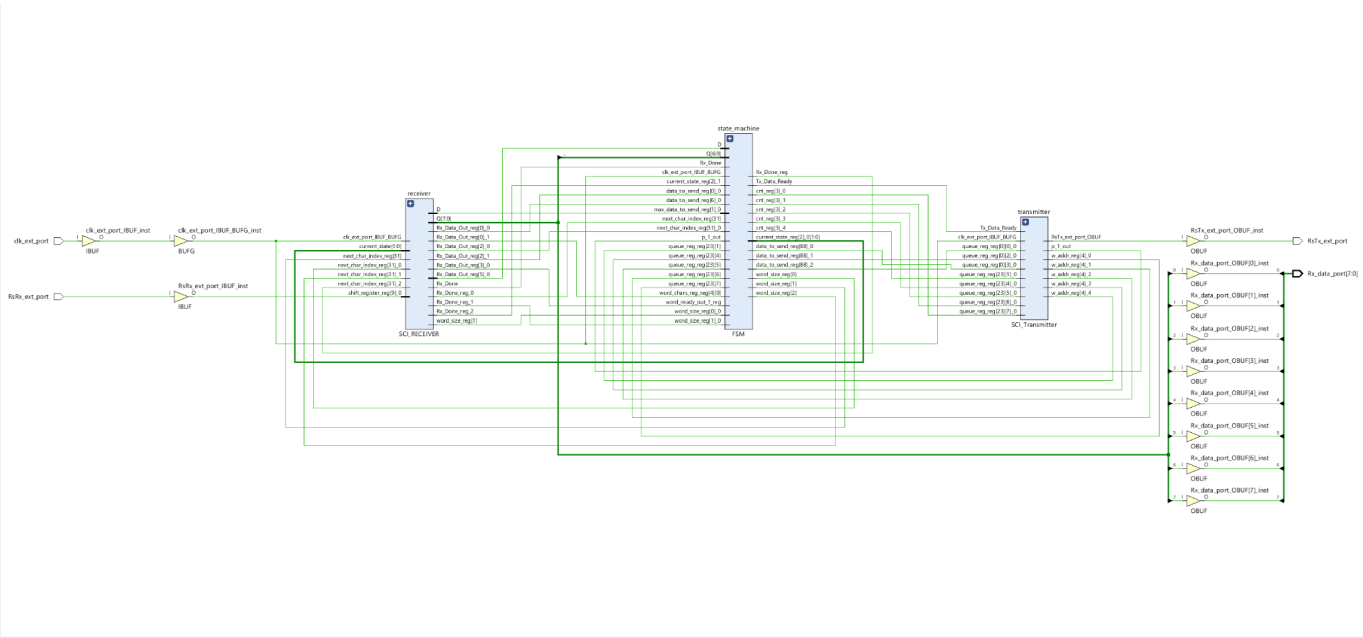
A. Encountered Errors: As we worked through this project, we encountered the following major errors:

- As we built the memory on VIVADO, we made the mistake of making it a distributed memory as opposed to a block memory. This slowed us down toward the end of the project and made debugging somewhat difficult. Once we recognized this as an issue, we found it easier to debug the rest of the errors.
- As we implemented the FSM in VIVADO, we found that it was not interacting well with our transmitter. The reason for this was that the signals connecting the FSM and the transmitter were not working properly in hardware

B. **Overall Conclusion:** Although our implementation successfully worked in simulation, it did not work in hardware. We are currently working on resolving the problem.

C. Final Schematics from Vivado Synthesis:

a. Top Level



b.

IX. Acknowledgements

Wordle dictionary. Wordle Dictionary. (n.d.). Retrieved June 5, 2022, from https://wordledictionary.com/

X. Appendix

Code: https://github.com/mubbie/es31-wordle/tree/main/code