Morse Code Converter

Cinay Dilibal & Claire Yu

ENGS 31 / COSC 56

Dartmouth College - Spring 2024

Abstract

For our ENGS31 final project, we developed a Morse Code Converter that uses the Basys3 FPGA board to generate Morse code signals by lighting a LED and producing sounds. The objective of the project is to create a large, complex digital system, through a top-down design with bottom-up implementation strategy, which resulted in five different components to get our implementation to function: SCI Receiver, Queue, Morse LUT, and Shift Register. Each component was tested individually before combining them in a Topshell to be tested all together in simulation and then hardware. The hardware was confirmed through manual testing and comparing signals to the oscilloscope waveforms. Regarding its functionality, a user will type letters/numbers into a keyboard, and the characters will take the form of 8-bit ASCII characters. The SCI Receiver checks that the data is an ASCII character, which is then sent to the Queue to be stored, and popped out to be read into Morse code alphabet, which are signals (dots/dashes) sent back to the user. The dots (1) and dashes (111) are demonstrated through how long the output is HIGH for. The lookup table holds each corresponding character with 22 bits and the length associated with the character. These are the values that are used in the Shift Register to output the correct morse code. This project demonstrated successful interfacing, signal processing, and provided a practical application of ENGS31 digital design principles.

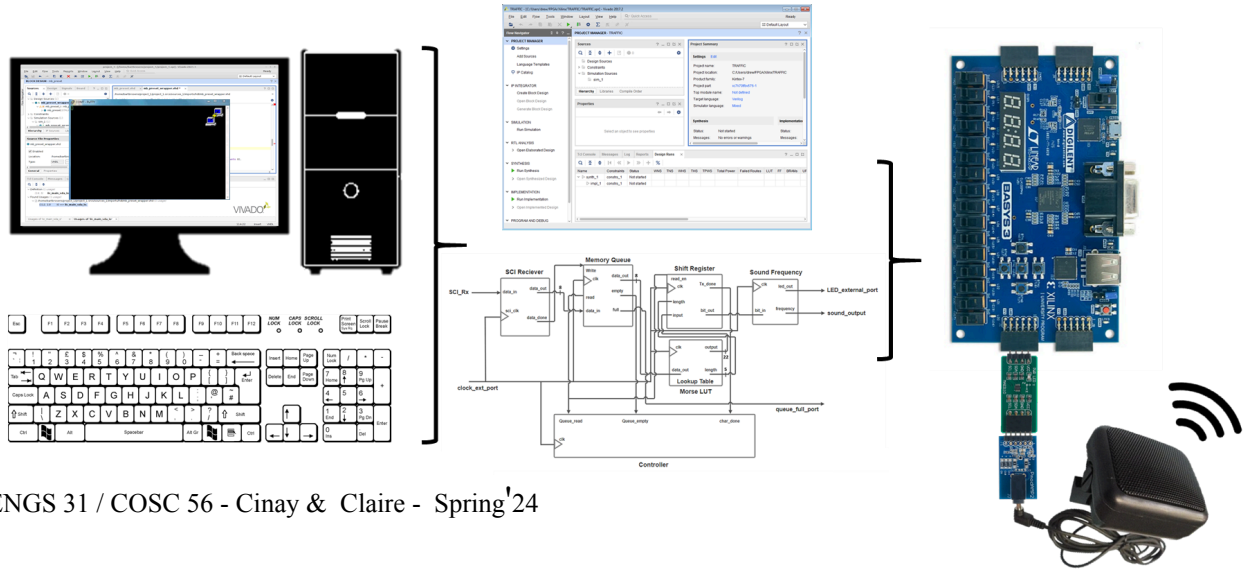
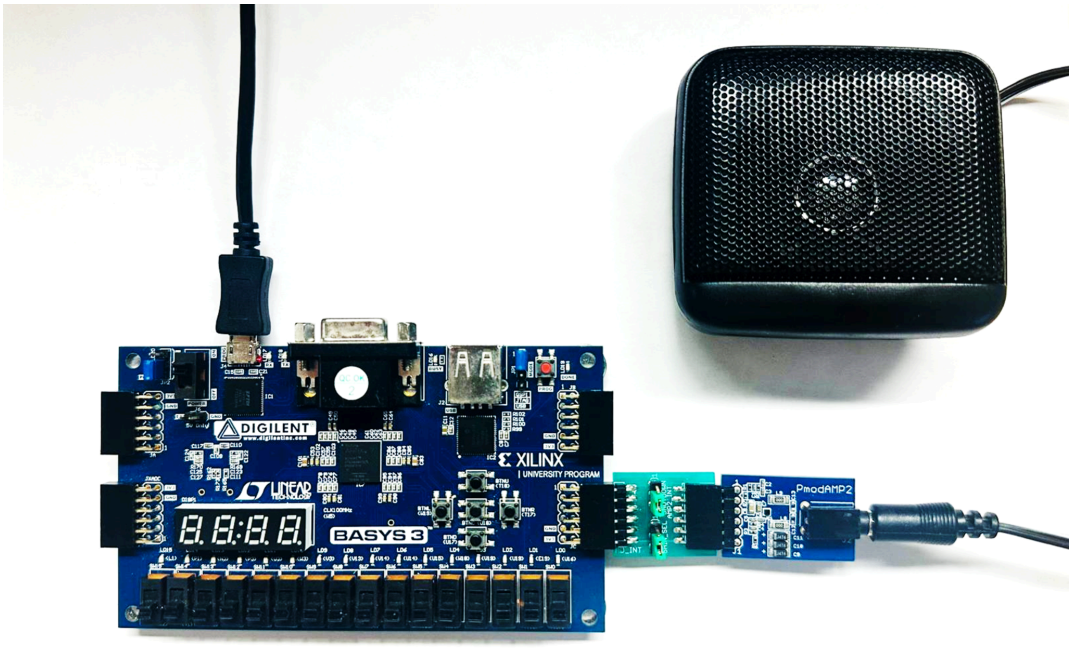


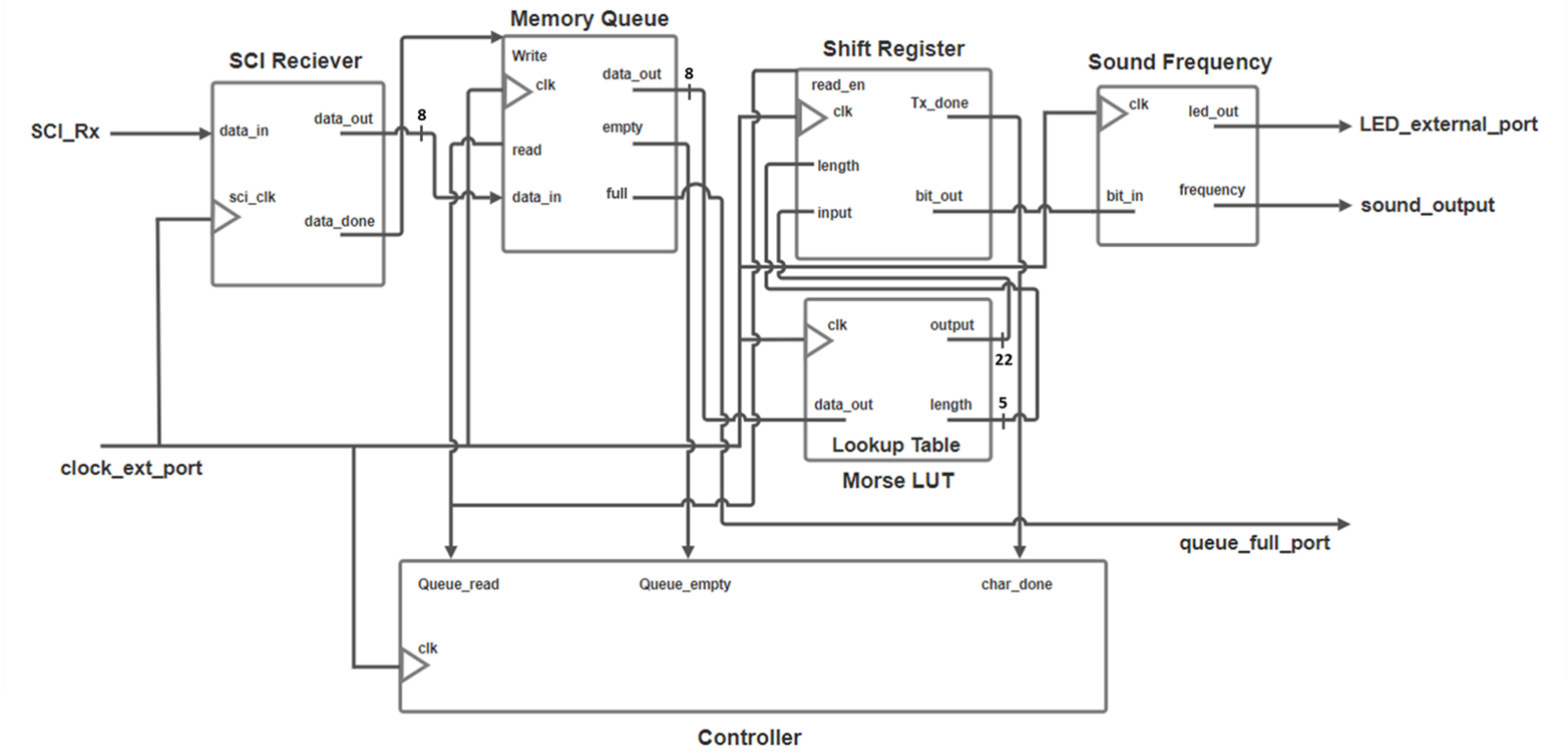
Table of Contents

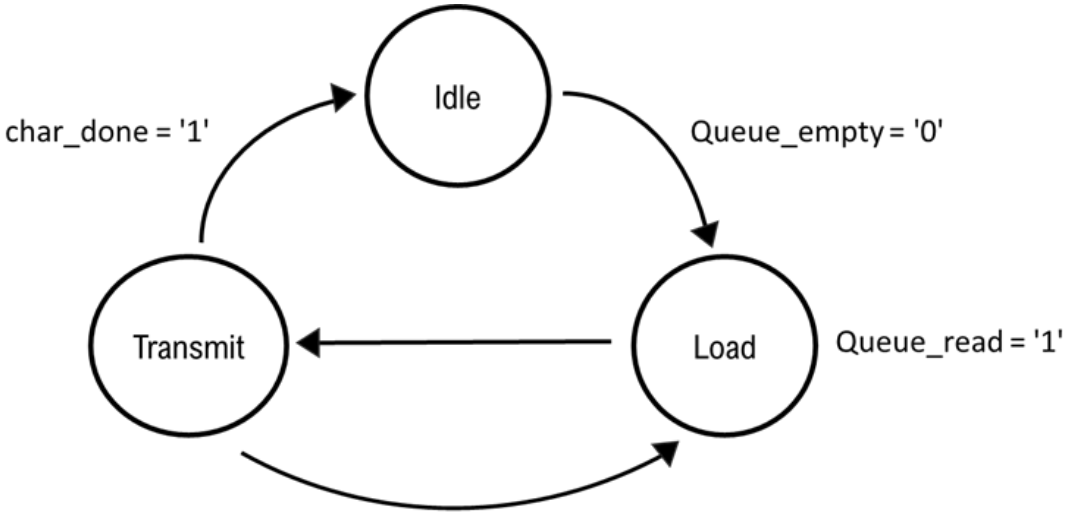
1. System Overview

The overall goal of the design was to implement a Morse Code Converter from ASCII characters that are sent into a PUTTY screen. The intended behavior of the design is depending on the characters that are being typed, dots and dashes will be represented out by a sound output and a blinking of a LED from the Basys3 FPGA Board. Therefore, when a user enters characters into a PUTTY screen, these characters will be converted into Morse Code signals that are timed out correctly. For the overall speed of the Morse code, a dot is on for T and dash is on for 3T. Specifications the circuit achieved that made it a successful design were the performance speed (Morse code response) and functionality testing. Testing each component before they were all combined helped us obtain a well-functioning Morse Code implementation.



1.1 Top-level Block Diagram





The top-level block diagram and state machine for our Morse Code Converter project encapsulate the components and their interactions within our system. In the block diagram, each component plays a crucial role in transforming ASCII characters into Morse code. The process begins with the SCI Receiver, which accepts serial ASCII input and converts it into parallel format. This data then moves to the Memory Queue, a FIFO system that manages the flow of characters to ensure the system processes them as typed by the user. Characters are then passed from the queue to the Morse LUT, which converts each character into a Morse code equivalent represented by a 22-bit vector and a length. This Morse code data is then fed into the Shift Register, which sequentially outputs bits for visual and auditory signaling via the Sound Frequency component. The Sound Frequency component then generates the corresponding sound waves and controls the LED blink patterns to represent Morse code visually.

The state machine controls the data flow through these components, with three primary states: Idle, Load, and Transmit. In the Idle state, the system awaits new characters, moving to Load when the queue indicates data is available. In Load, the system reads a character from the queue and prepares it for transmission. Once loaded, the state transitions to Transmit, where the character is converted to Morse code and output until complete, after which the system returns to Idle to await the next character. This cycle is repeated to ensure efficient and orderly processing of input to output, following the Morse code timing conventions for signal representation.

1.1.1 Pseudocode

1. Idle State: The system remains in this state until there is data available in the Memory Queue. When data is detected, it transitions to the Load state.

***Idle State*:**

if not Memory Queue is empty:

Transition to Load State

else:

Stay in Idle

2. Load State: In this state, a character is read from the Memory Queue, and is converted to Morse code using the Morse LUT. Loads the Morse code data and its length into the Shift Register. Transitions to the Transmit state for outputting the Morse code.

***Load State*:**

character = removed from Memory Queue

morse_code, length = Morse LUT Convert character

Load morse_code and length into Shift Register

Transition to Transmit State

3. Transmit State: The final state handles the bit-by-bit transmission of Morse code through the Shift Register. The LED and sound outputs are controlled for each bit. Continues until all bits of the Morse code have been transmitted. Returns to Idle State to wait for the next character.

***Transmit State*:**

while not all bits transmitted:

bit = Shift Register outputs next bit

Send out the Morse signal for bit

Wait appropriate Morse code timing interval

Transition to Idle State

This high-level state machine (HLSM) demonstrates that the Morse Code Converter operates efficiently, managing the flow of data from the point of input to the visual and auditory signaling of Morse code.

1.2 Description of Ports

Component	Inputs	Description	Outputs	Description
SCI Receiver	sci_clk data_in	The system clock that is running on 100MHz Signifies HIGH when a bit has been sent in and needs to be processed for a character	data_out data_done	The 8-bit ASCII character that needs to be processed into Morse Code One bit signal that goes HIGH when all 8-bits for a character have been shifted out
Memory Queue	Clk Write read Data_in	100MHz Lets the queue know there is a character waiting to be read The queue is ready to read in a character The character that has been sent in	Data_out Empty Full	FIFO logic, where the initial character that has entered will be processed If the queue currently has no characters stored When the queue is full and cannot take anymore characters
Morse LUT	clk data_out	100MHz Taking in the 8-bit character to be processed into Morse Code logic	Output	22-bit standard logic vector in order to account for the dots and dashes depending on

				the character. This is where the character turns into Morse Code form Store the actual length of the character, such that it will make the shift register easier to shift the correct number of bits out
Shift Register	Clk read_en length input	100MHz Lets the shift register know there is data to be processed The length of the character that is currently needing to be shifted The character inputted in Morse-code form (22 bits)	Tx_done bit_out	The shift register has completed all of its shifts for the character The bits that are represented out for the LED and sound one bit at a time
Frequency Sound	clk bit_in	100MHz Take in the bits that are outputted	frequency LED_out	To get the sound waves that correspond to the character Just takes in the bit_in from what it is

1.3 Description of Components

Components	Description
SCI Receiver	Receives the ASCII characters that are sent in from a user. Such that the data is entered serially but will be outputted as parallel from

	this component.
Memory Queue	FIFO mechanism, where the ASCII characters sent from the SCI receiver are stored in the Queue to handle when a user types in characters that are too fast for the system to process.
Morse LUT	In order for the ASCII characters to be converted into Morse code formatting. Takes in the 8-bit characters and outputs them into 22-bits form.
Shift Register	The 22-bit data is inputted into the shift register, which is used to demonstrate the dots and dashes back to the user. The register sends a bit out one at a time.
Frequency Sound	Following the bits that are coming out from the shift register, this component reads in if the bit is HIGH or LOW, and sends out the corresponding sound wave.

2. Technical Description

The components that were demonstrated in this project include SCI Receiver, Memory Queue, Morse Code Lookup Table, Shift Register, and Frequency Sound. The clock is an input to all components where it runs on 100 MHz.

2.1 SCI Receiver

The SCI Receiver takes in ASCII characters that are sent by a user. A serial-to-parallel conversion happens here in order for the data to be processed into Morse Code formatting. The Receiver also takes in a start and stop bit, with an implementation of a controller to tell when a data is ready to be shifted.

2.1.1 Description of Ports

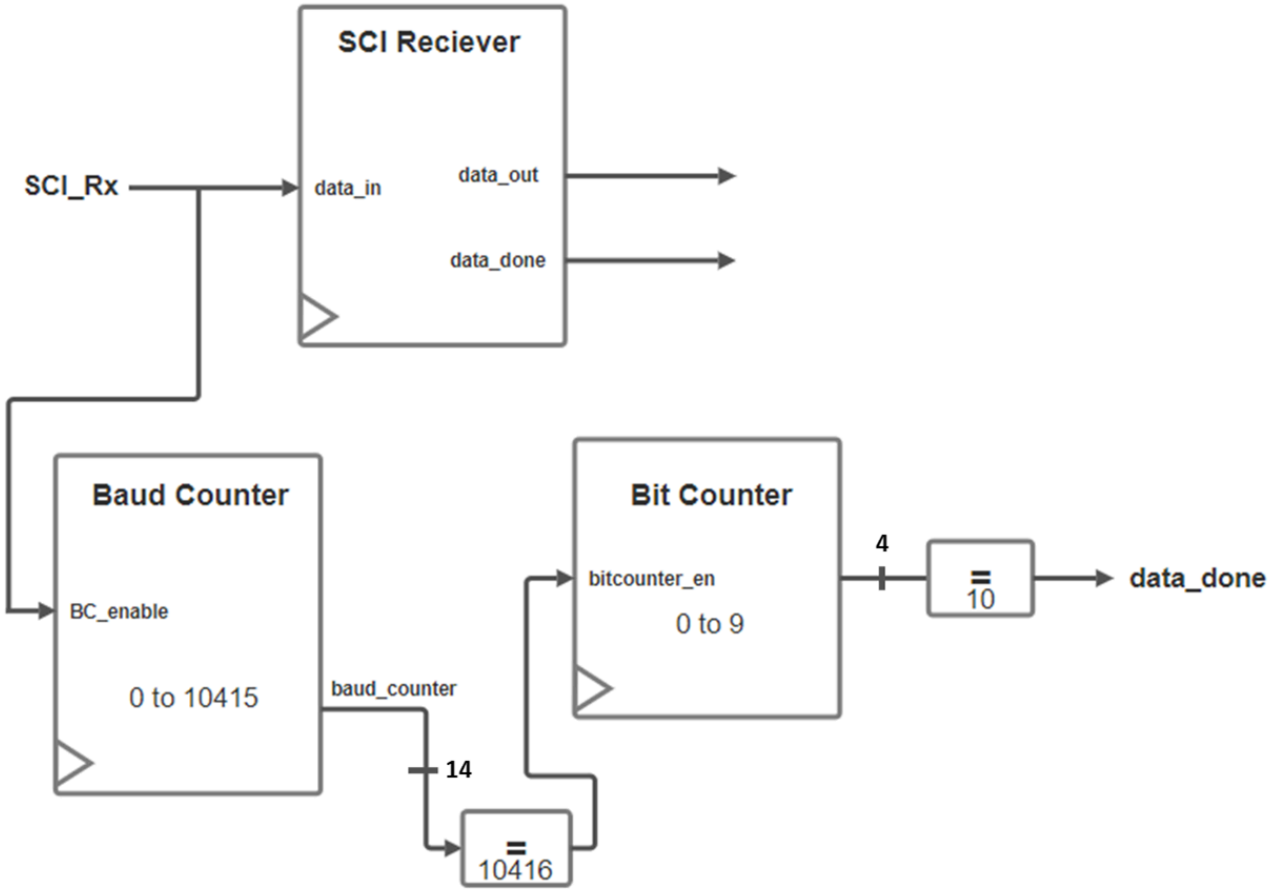
Inputs:

- **data_in:** The ASCII bit that is entered by a user. While ASCII characters are 8-bits, the data takes in one bit at a time.

Outputs:

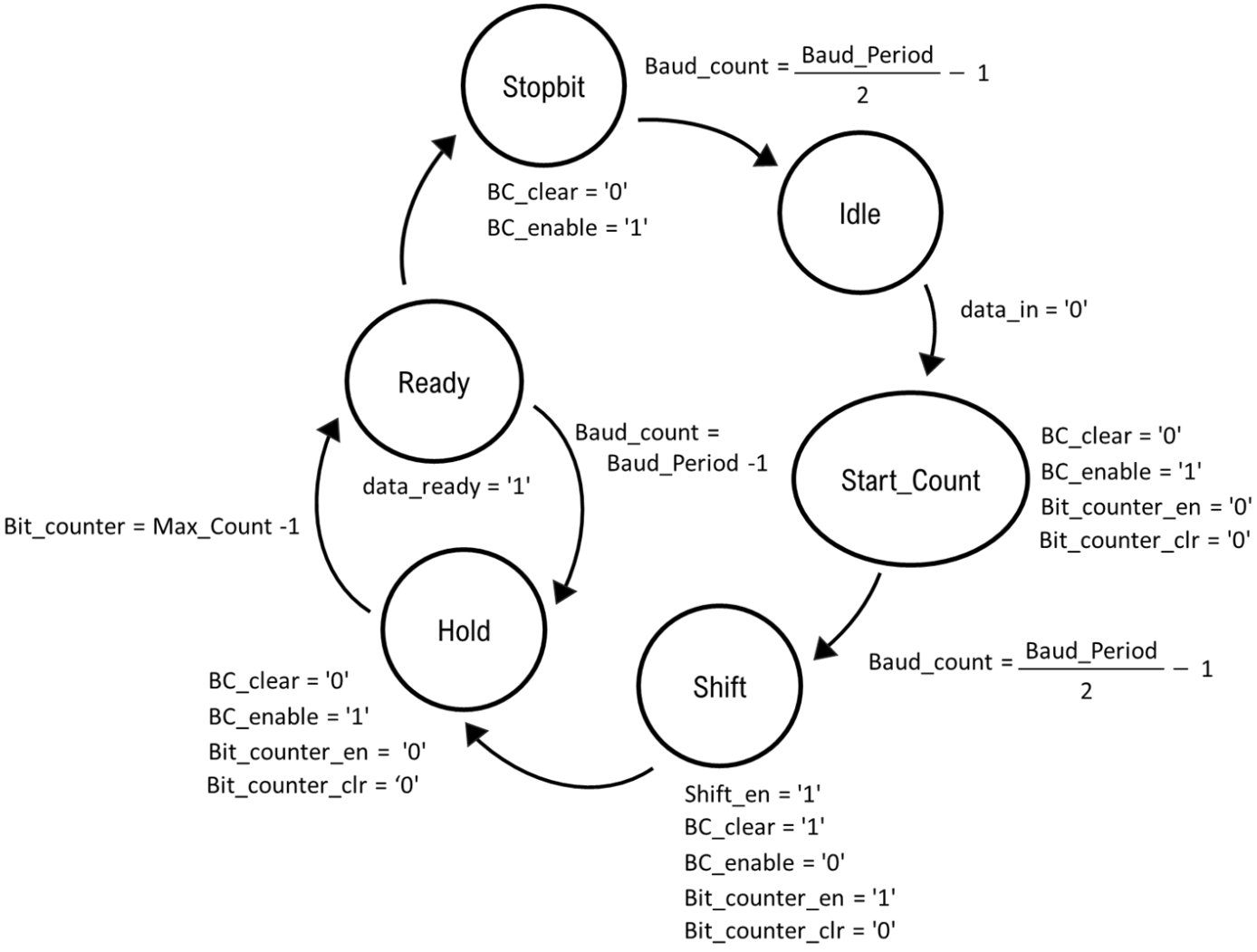
- **data_out:** 8-bit parallel data that is sent out of the SCI Receiver in ASCII character form
- **data_done:** A signal that goes HIGH when all the data associated with a character has been shifted out

2.1.2 Register Transfer Level (RTL) Diagram



The RTL diagram contains a shift register, controller, and two counters: baud and bit. The baud counter is used to regulate the timing of when ASCII values can be read and added to the shift register to be shifted out, in order for the system to maintain the same sample rate for each data. The bit counter helps keep track of how many bits have been entered to reach a terminal count of 10 bits, which signifies that the character inputted can be pushed out to the memory queue. The shift register is just used to store and shift the data. In this case, to convert the serial stream of ASCII values to parallel for all the bits to be sent out simultaneously to represent a character.

2.1.3 Finite State Machine (FSM) Diagram



The finite state machine for the SCI Receiver, which takes in ASCII characters and performs serial-to-parallel conversion for Morse Code formatting, operates through several states to manage the reception and processing of serial data. Here is a detailed walkthrough of our state diagram for the SCI Receiver:

- **Idle:** This is the default state of the receiver. The FSM remains in the Idle state until a valid start bit is detected (data_in = '0'). During this state, the BC_clear signal is set to '0', and BC_enable is set to '1' to prepare the bit counter for the upcoming data bits.

- **Start_Count:** Once the start bit is detected, the FSM transitions to the Start_Count state. In this state, the baud counter (Baud_count) is set to half of the baud period minus one, which ensures that the sampling of incoming data bits is centered within the data bit window. This state helps synchronize the receiver with the data stream.

- **Shift:** After the start bit is confirmed, the FSM moves to the Shift state. Here, the shift register begins to receive the incoming bits. The Shift_en is set to '1' to enable the shifting process. The bit counter (Bit_counter_en) is also enabled to count the number of bits received. Once the shift register receives a total of 8 data bits (since ASCII characters are 8 bits), the FSM transitions to the next state.

- **Hold:** In this state, the FSM checks if the bit counter has reached its maximum count (Max_Count - 1), which indicates that all bits of the character have been received. The Hold state is crucial for validating that the full character has been captured before proceeding. The Bit_counter_en is set to '0' to prevent any further counting during this validation phase.

- **Stopbit:** After verifying the reception of the entire character, the FSM expects a stop bit to confirm the end of the character transmission. This state ensures that the character frame is correctly terminated with a stop bit. If the stop bit is received correctly (Baud_count = Baud_Period - 2), the FSM transitions to the Ready state.

- **Ready:** The character is now fully received, and the data_ready signal is set to '1'. This signal indicates that a complete character in parallel form is ready to be sent out from the SCI Receiver. The data_done output would also go HIGH at this point to signal the completion of the character reception.

- **Return to Idle:** After completing the character transmission, the FSM returns to the Idle state to wait for the next character's start bit, resetting necessary counters and signals (BC_clear, BC_enable) to prepare for the next incoming data.

Our FSM efficiently handles the reception of ASCII characters, ensuring that each character is correctly framed, synchronized, and converted from serial to parallel format. The transitions between states are designed to manage different aspects of the data reception process, including synchronization, bit counting, validation, and readiness for data output.

2.2 Memory Queue

The memory queue takes in the 8-bit ASCII characters, and stores them, such that when a user inputs multiple characters at once, the system can use the queue to manage what values must be read in. Since it is a queue, the register has a first-in, first-out mechanism, such that it maintains the order of the letters that have been sent in.

2.1.1 Description of Ports

Inputs:

- **data_in:** Receives the data from the SCI Receiver and is stored into the queue to be read from the memory

- **read:** The read is an enable bit to let the system know that another ASCII character is ready to be processed. It goes high one character at a time

- **Write:** The write is an enable bit to let the system know that there is data that is waiting to be stored, such that a new piece of memory can be added in.

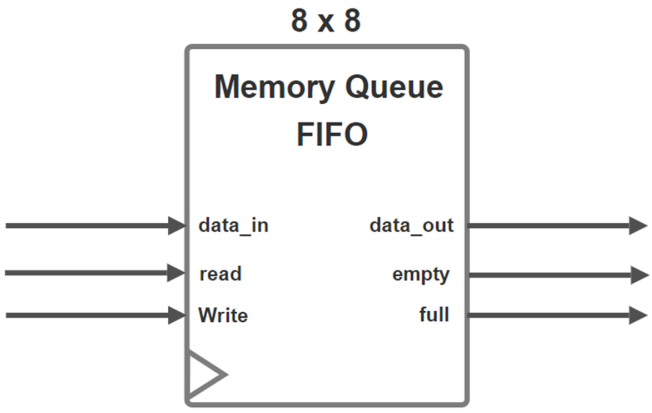
Outputs:

- **data_out:** 8-bit parallel data that is sent out of the SCI Receiver in ASCII character form for a character

- **empty:** An enable bit that tells the system when the queue is currently empty. When the queue is not empty, it will take in the characters and start popping them out. If it is, it will stay idle and wait to load data.

- **Full:** Lets the user know when the queue is full and cannot take in any more characters. It will go high when the read and write addresses are pointed at the same location in the memory.

2.1.2 Register Transfer Level (RTL) Diagram



The RTL diagram just has the queue, however, the queue does use the top-shell controller in order to handle the empty and full ports. Please refer back to 1.1 to see the FSM for the top-shell.

2.1.3 Description of Memory

The queue is an 8x8 array, which means that there are eight available locations in the memory block for 8-bit ASCII characters (8 deep, 8 wide). While it is currently at 8, the length of the queue can always be adjusted accordingly. To create the memory queue, a register file signal, read, and write address signals were used, in order to maintain what data has been entered with also what data has been processed. In other words, these address signals were used to manipulate the memory. Other control signals that were used include full and empty signals to keep track of the number of elements in the queue.

2.3 Morse Lookup Table

The lookup table is a read-only memory component, where it converts the ASCII characters into Morse code formatting. Reading in an 8-bit ASCII character, the lookup table outputs a 22-bit code to accommodate the dots and dashes.

2.3.1 Description of Ports

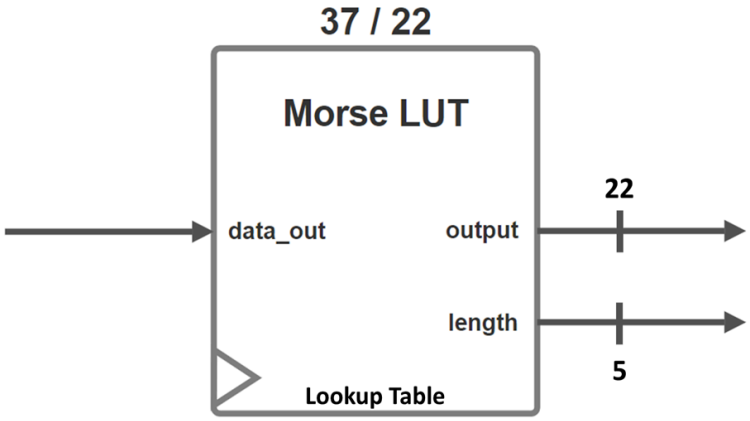
Inputs:

- **data_out:** Takes in the 8-bit ASCII character to find out what it should be converted into for Morse Code

Outputs:

- **output:** Sends a 22-bit code out to the shift register that corresponds to the morse code formatting of dots and dashes
- **length:** Since the data is turned into 22-bit, we store the actual length of the character in this variable to be used for knowing how many bits should be shifted in the shift register

2.3.2 Register Transfer Level (RTL) Diagram



The Morse LUT (Lookup Table) depicted in the Register Transfer Level (RTL) diagram translates 8-bit ASCII characters into Morse code representations, specifically designed to output a 22-bit code that encapsulates the series of dots and dashes for each character.

2.1.3 Description of Memory

As mentioned above, the lookup table is read-only memory. The memory array is 37 deep since we can read in letters A-Z, 0-9, and spaces while being 22 bits wide. For instance, “A” is

associated with 65 in ASCII value, which is “0100001.” From there, we convert the character in terms of its Morse Code format. For “A,” in the Morse code signal, it is one dot and one dash, where we get “10111000.” A dot is represented with a ‘1,’ while three ‘111’s represent a dash. However, we also account for 3T between each character by ending the value with three zeros. With the Morse Code for the letter, we add zeros to the data vector to make the output 22 bits, such that the input into the shift register gets the same length of a vector every time.

A couple of other examples are demonstrated below:

- ‘B’ has one dash and three dots, therefore "111010101000" & "0000000000”
- ‘E’ has one dot, therefore "1000" & "0000000000000000000" (where zeros are appended in the end)

2.4 Shift Register

The shift register takes the 22-bit and length inputs from the lookup table and sends them out to the sound frequency component to be demonstrated to the user. In this case, there is also a controller that is used along with the shift register. There are two counters used for the shift register. The dip counter helps to regulate the timing of when each data bit can be shifted, while the bit counter controls how many bits have been shifted out taking in consideration of the wait times.

2.4.1 Description of Ports

Inputs:

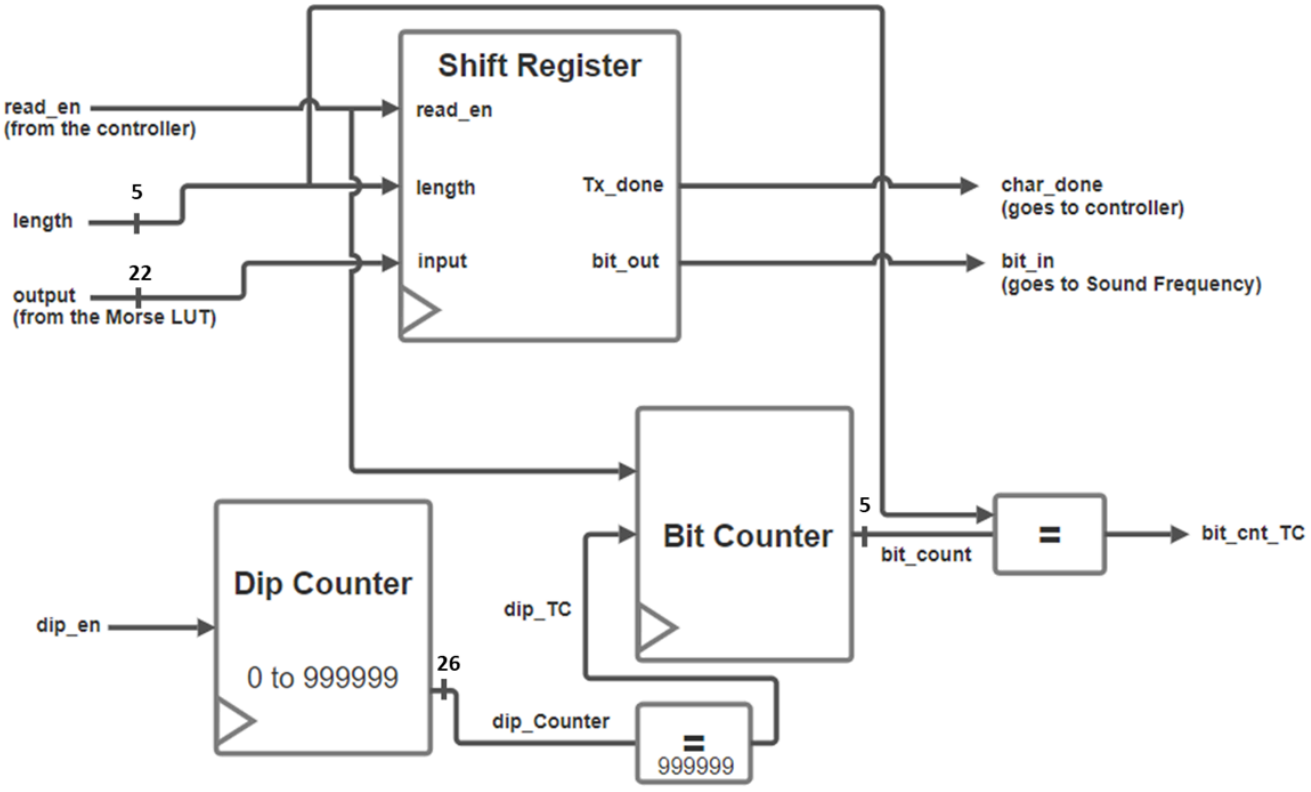
- **read_en:** Is an enable bit to indicate when there is data needing to be shifted out

- **length:** Takes in the length of each character, such that for bit counting, it will end at the right number. However, we also have a length register as an internal signal that increments the vector by two bits to account for the three zeros at the end.
- **input:** The 22-bit data that comes from the lookup table. This data is used to fully demonstrate the morse code conversion from ASCII characters to Morse Code format.

Outputs:

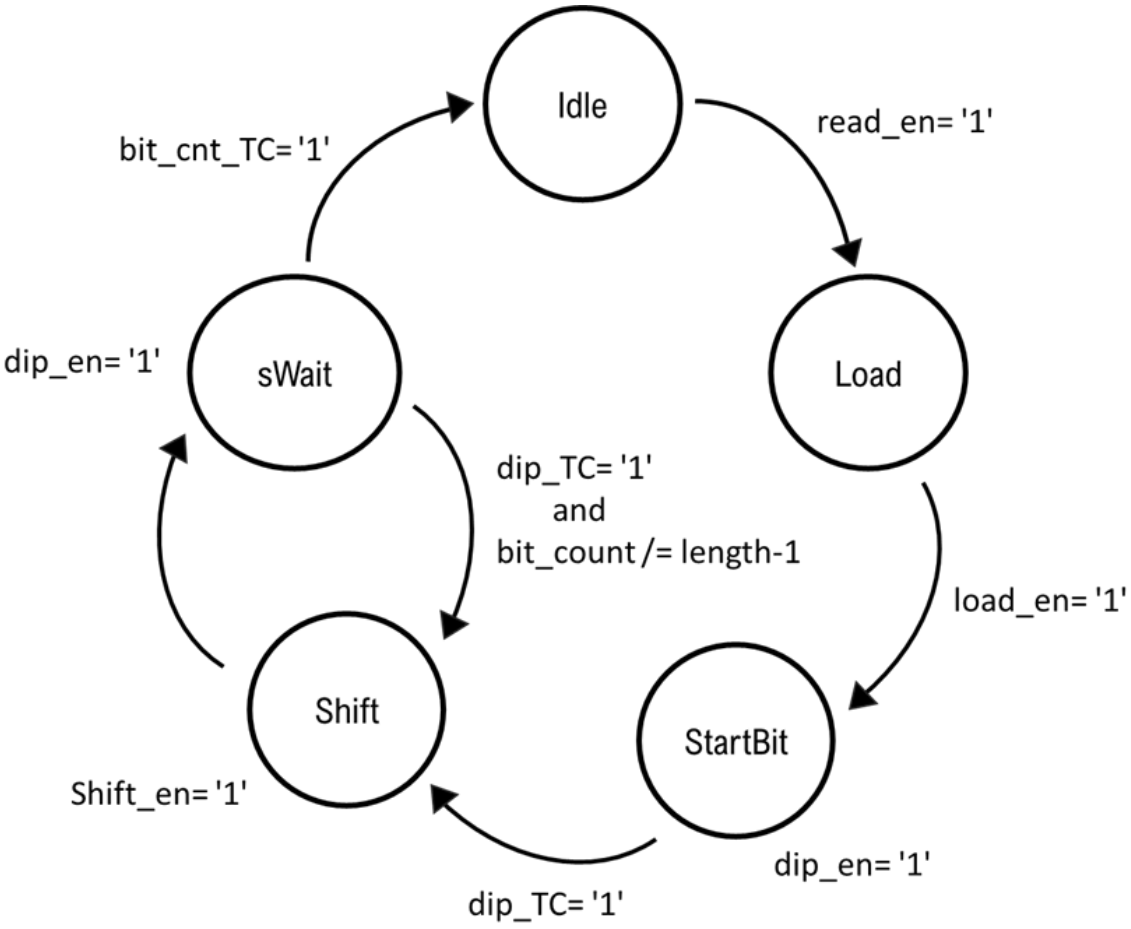
- **Tx_done:** Is an enable bit to indicate that a character has been fully processed and shifted out
- **bit_out:** A one bit value, which is demonstrated from the input being shifted. Whenever input is shifted, the MSB from the shift register will be the bit_out.

2.4.2 Register Transfer Level (RTL) Diagram



The RTL diagram for the "Shift Register" is described above. The shift register includes two counters: dip and bit counter. The dip counter takes in signals from the controller and will count up until the terminal count has been reached. In this case, since the clock is 100 MHz, and we want to read at 10 bits per second, such that 1 baud per bit, resulting in the dip_period to be 10,000,000. The second counter is used for the length of the associated character entered. It increments the count whenever a bit has been shifted up to the total length of the character. The shift register is used to send the data out going from parallel to serial since the data is being sent out a bit at a time.

2.4.3 Finite State Machine (FSM) Diagram



The state diagram starts with the idle state and stays in this state until read_en goes HIGH. In this case, when there is data that can be read into the shift register to be shifted out. Then load From the Load state, the FSM transitions to the StartBit state when the load_en signal is set to '1'. In the StartBit state, the machine initializes the transmission of data bits. This state is important for setting up the initial conditions necessary for data shifting, such as resetting the bit counter and preparing the shift register. Once the initial bit has been processed, the FSM moves into the Shift state. In this state, the data from the shift register is sequentially shifted out one bit at a time. The shift operation continues until all the bits specified by the length have been transmitted. This is controlled by the bit_count, which increments with each shift operation and compares against the length to determine when all bits have been sent.

If the dip_TC signal (Data In Place Terminal Count) is '1', indicating that the end of the data bit sequence has been reached, the FSM then transitions to the sWait state. This state introduces a wait period, which is crucial for ensuring that there is a gap between the consecutive data transmissions. The wait period helps in distinguishing separate data units or characters in the transmission stream.

Finally, after the wait period is over, the FSM can either return to the Idle state if there is no further data to process (read_en is '0'), or it can proceed back to the "Load" state to begin transmitting a new data unit if more data is present (read_en is '1'). This loop continues as long as there is data to be transmitted, with the FSM cycling through the Load, StartBit, Shift, sWait, and back to Idle states as required. The design of this FSM ensures that each character or data unit is properly processed and transmitted with clear delineation, making it suitable for serial data communication where timing and sequence integrity are critical.

2.5 Sound Frequency Output

The sound frequency is just used in order to get the beeping sound for the output. It also takes the bit_out from the shift register, such that the blinking of the LED also happens from this component.

2.5.1 Description of Ports

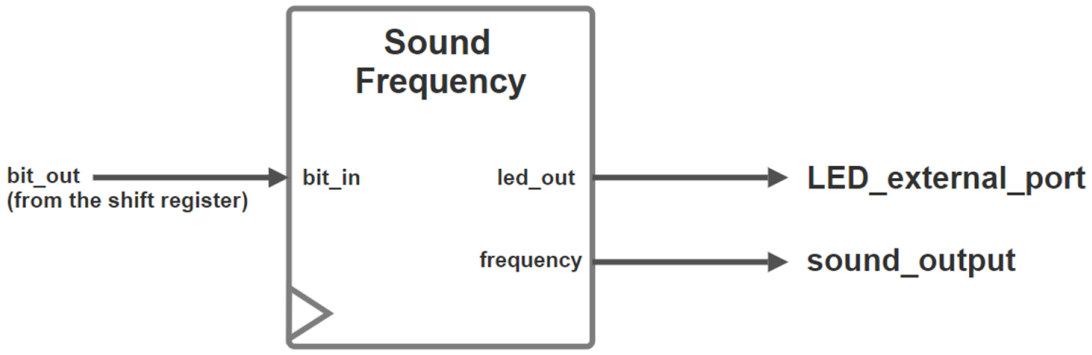
Inputs:

- **bit_in:** Gets the output of the shift_register from the bit_out signal. Although the shift register already sends the characters in Morse Code, this is just used in order for the sound output to be demonstrated as well.

Outputs:

- **frequency:** Sends out the sound waves associated with each character
- **LED_out:** Takes in bit_in to send out to the LED to blink accordingly.

2.5.2 Register Transfer Level (RTL) Diagram



The RTL diagram for the "Sound Frequency" is described above. The module takes a single input, bit_out from a shift register, which is a digital signal indicating Morse code elements. This signal is fed into the bit_in input of the Sound Frequency module. The module has two primary outputs: LED_external_port and sound_output. The led_out signal controls the

LED_external_port, directly reflecting the state of the bit_in. When bit_in is high (indicating a Morse code dot or dash), the led_out goes high, turning the LED on to signal the presence of a Morse element. Conversely, when bit_in is low (indicating a pause between Morse elements), the led_out goes low, and the LED turns off.

The frequency output manages the sound_output and produces sound waves corresponding to the Morse code signals being received. A high bit_in results in the generation of a tone (the specifics of like frequency and duration are determined by the internal settings of the module), while a low bit_in ceases the sound, indicating a pause. This design ensures a synchronized audio-visual representation of Morse code.

3. Design Validation

A few of the screenshots annotated below were done in the EDA playground for validation, however, all the designs were also tested with Vivado to make sure it was giving the same output as demonstrated in EDA.

3.1 SCI Receiver

Success was achieved for our SCI Receiver when each ASCII character value has been correctly converted from serial to parallel. In other words, our testing strategy was to make sure that when one bit was inputted into the Receiver at a time, the output should be an 8-bit vector of the actual character.

3.1.1 Behavioral Simulations

For this waveform, we tested when the characters ‘A’ and ‘F’ were sent into the SCI Receiver, validating the component through manually verifying with a testbench. The simulation

shows proper operation since the inputted values, while going from least to most significant bits, are correctly outputted in the ordering of their associated characters. For instance, data_in sends in 0, 1, 0, 0, 0, 0, 0, 1, 0, which results in the output as "0100001," following the binary number of 'A.' Additionally, it is important to note that the data_ready signal only goes up when all data bits have been shifted, which is also shown in the waveforms.

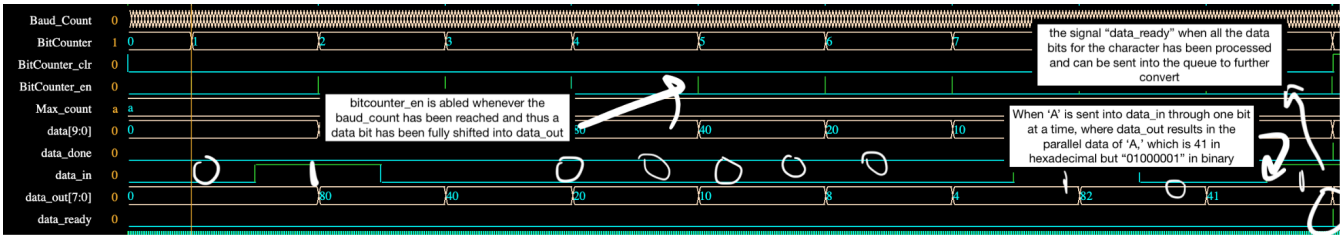


Figure 1: *Waveform for the character 'A'*

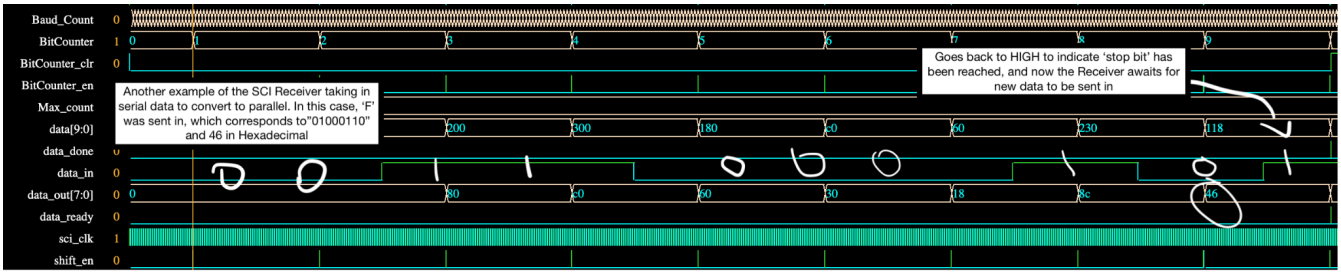
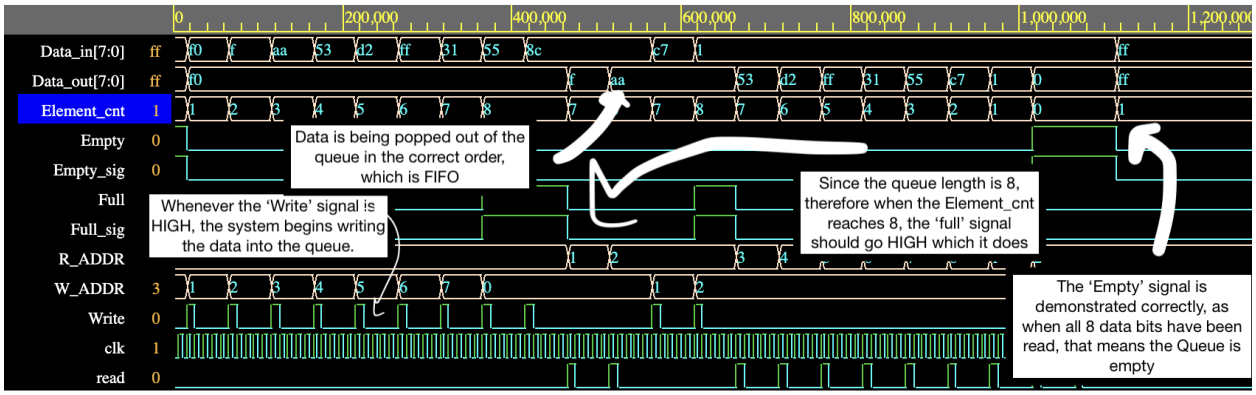


Figure 2: *Waveform for the character 'F'*

3.2 Memory Queue

Success was achieved for our Memory Queue when it was correctly writing and reading in data, since the purpose of this component was to store in data when a user types in various letters at once. Therefore, our testing strategy was to reach the maximum number of data that a queue can hold and see the outcome of it.

3.2.1 Behavioral Simulations



For this waveform, we tested when various data were entered into the queue one at a time (data_in), until the queue was filled with a maximum length of 8. We tested to make sure that the 'Empty' and 'Full' signals were correctly working, which was verified by writing in data eight times, and then reading in the data eight times where the empty signal became HIGH. Furthermore, we were able to verify that the queue was working as FIFO, since according to the waveform, the first data that was entered was 'f0,' which was also the first data to be read out as shown with data_out.

3.3 Morse Lookup Table

Success was achieved for our Morse Lookup Table when it was correctly converting the ASCII characters into Morse Code formatting, as well as correctly holding the corresponding length for the associated character. In this case, we tested with a letter and number to verify that it was properly operating. Our testing strategy was to make sure that our inputted values of 8-bits were converted to 22-bits with the length of the values being accurate as well.

3.3.1 Behavioral Simulations

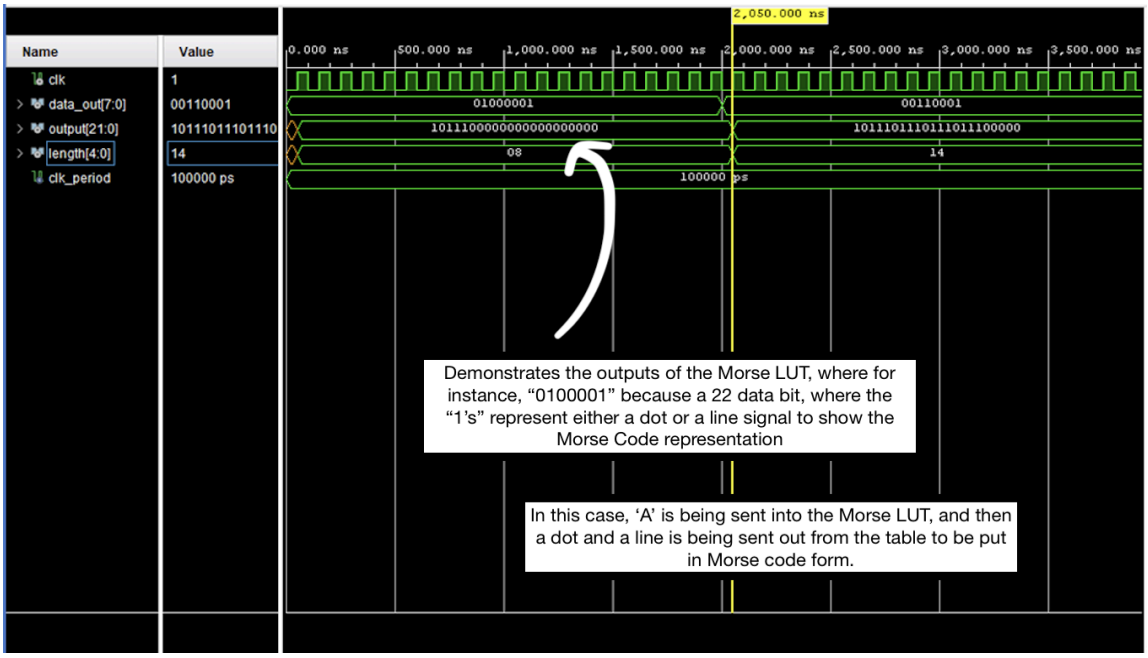


Figure 3: *Waveform for the Morse Lookup Table with ‘A’ and ‘1’ being tested*

In the waveform above, we tested when ‘A’ and ‘1’ were sent into the Morse Lookup Table. ‘A’ corresponds to ‘01000001,’ while ‘1’ is represented with ‘00110001.’ Given that ‘A’ in Morse Code is one dot and a dash, we checked to make sure that there were only four 1’s shown in the output, with the remaining bits as ‘0s.’ This was also the same procedure when we tested with ‘1,’ verifying that the character matched its Morse Code conversion. Although this waveform only showed two characters, we made sure to check every character to ensure we had the correct conversions for all characters.

3.4 Shift Register

Success was achieved for our Shift Register when it correctly obtained the data from the Morse Lookup Table and used the length to accurately send out the correct number of bits one at a time for each character inputted. Therefore, we tested to make sure that the shifting of the data

was demonstrated, as well as the enable bit (Tx_done) that shifting was complete became HIGH at the right time and the FSM associated.

3.4.1 Behavioral Simulations

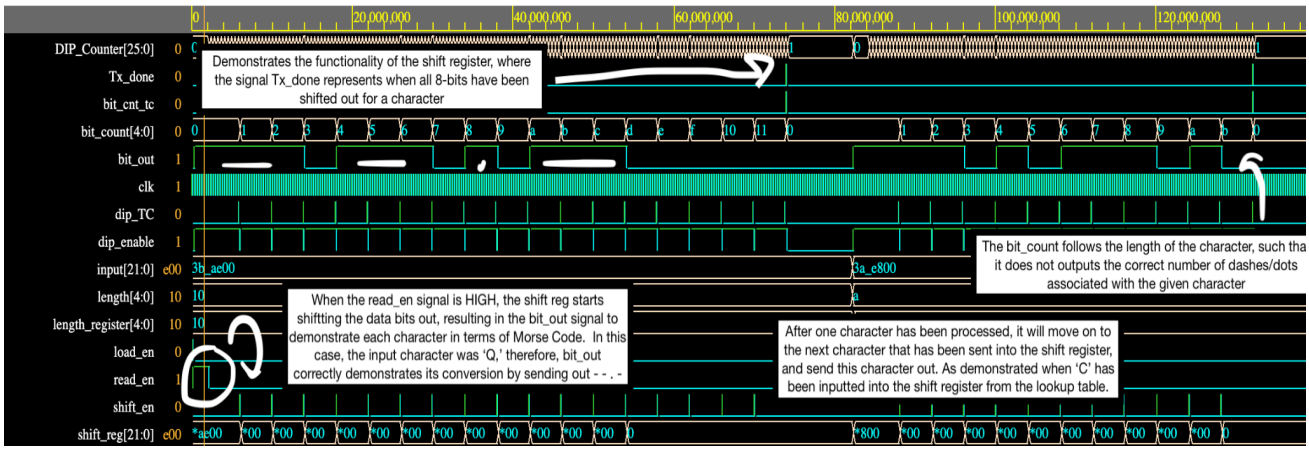


Figure 4: *Waveform that demonstrates the overview of the simulation for the Shift Register.*

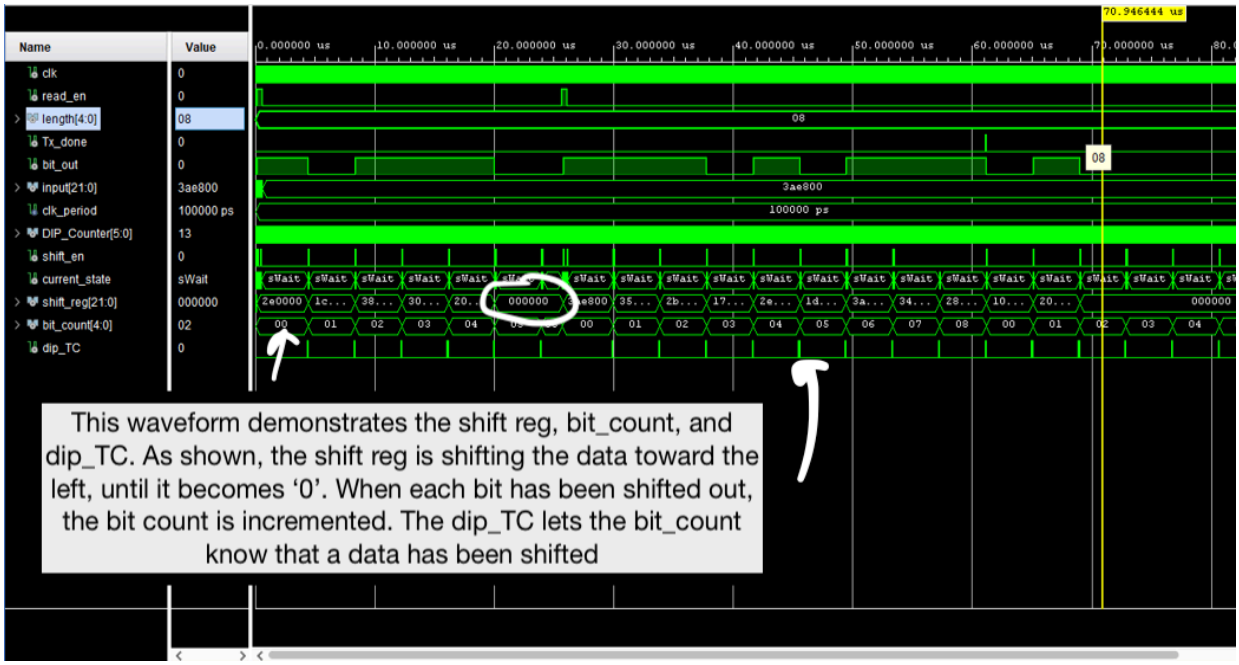


Figure 5: *Waveform that describes the shift register signal in more detail.*

In the waveforms above, we demonstrate what happens when two characters have been sent into the shift register after the inputted values were converted into Morse Code. In this case,

the characters are ‘Q’ and ‘C.’ As described with the annotations, whenever the ‘shift_en’ goes HIGH, the register shifts out one bit of data that corresponds with the character. We verified that the shifting was properly operating by making sure that the shift_reg becomes all ‘0’s when there is no shifting left to complete, while also manually verifying that ‘bit_out’ has the correct number of dots and dashes associated with the character we have tested.

3.5 Sound Frequency Output

For the last component, this was more of a verifying that the data that was sent out from the shift register can also be used to produce a beeping sound. Therefore, the testing strategy for this component was to check when a bit was sent as ‘1,’ we received a HIGH output and vice-versa.

3.5.1 Behavioral Simulations

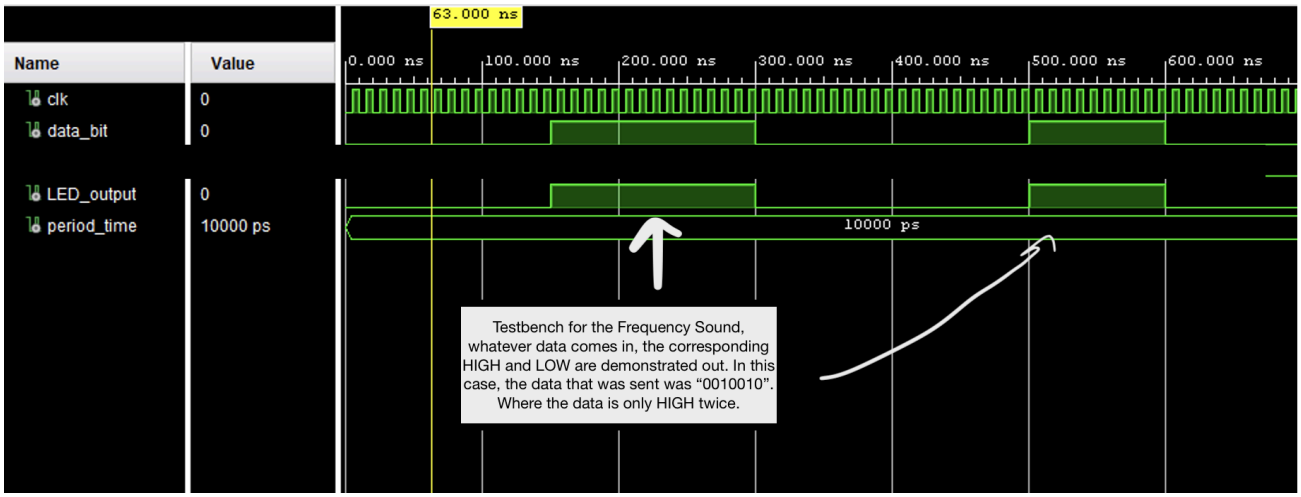


Figure 6: *Waveform that demonstrates the output of the Sound Frequency component. The output should just match the input directly.*

In this waveform above, the simulation shows proper operation because the data that was sent in only goes HIGH twice, which is demonstrated. If we were getting the correct output, we

could verify that the sound coming from our implementation should be beeping at the right moments.

3.6 Overall Design

After verifying that each component had proper operation, we validated that our design was correctly working by connecting our components together through the Topshell VHDL code. Since we could not test PUTTY for simulation purposes, our testing strategy was to just send in the ASCII characters through the testbench and validate that the Morse Code conversion was achieved in the data_bit. While the waveform only shows a couple of characters being tested, all the characters were also tested to confirm that our implementation can handle multiple characters inputted all at once.

3.6.1 Behavioral Simulations

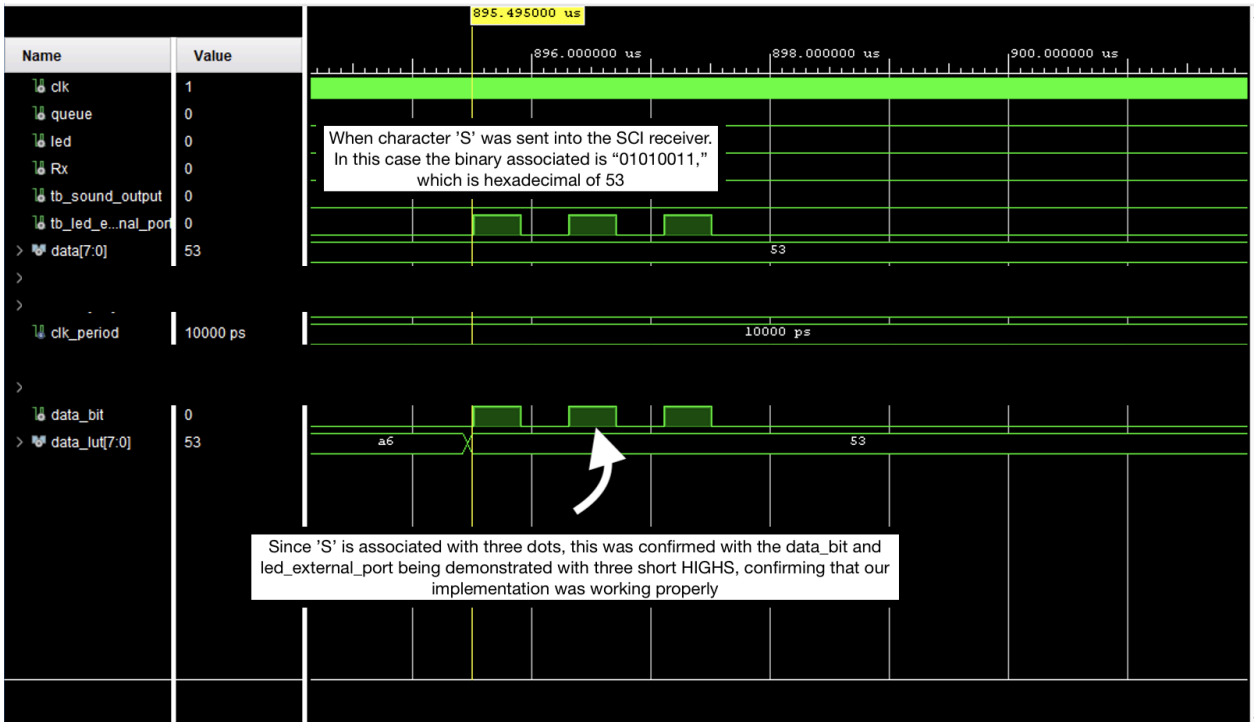


Figure 7: Waveform that demonstrates the output when the character 'S' is inputted

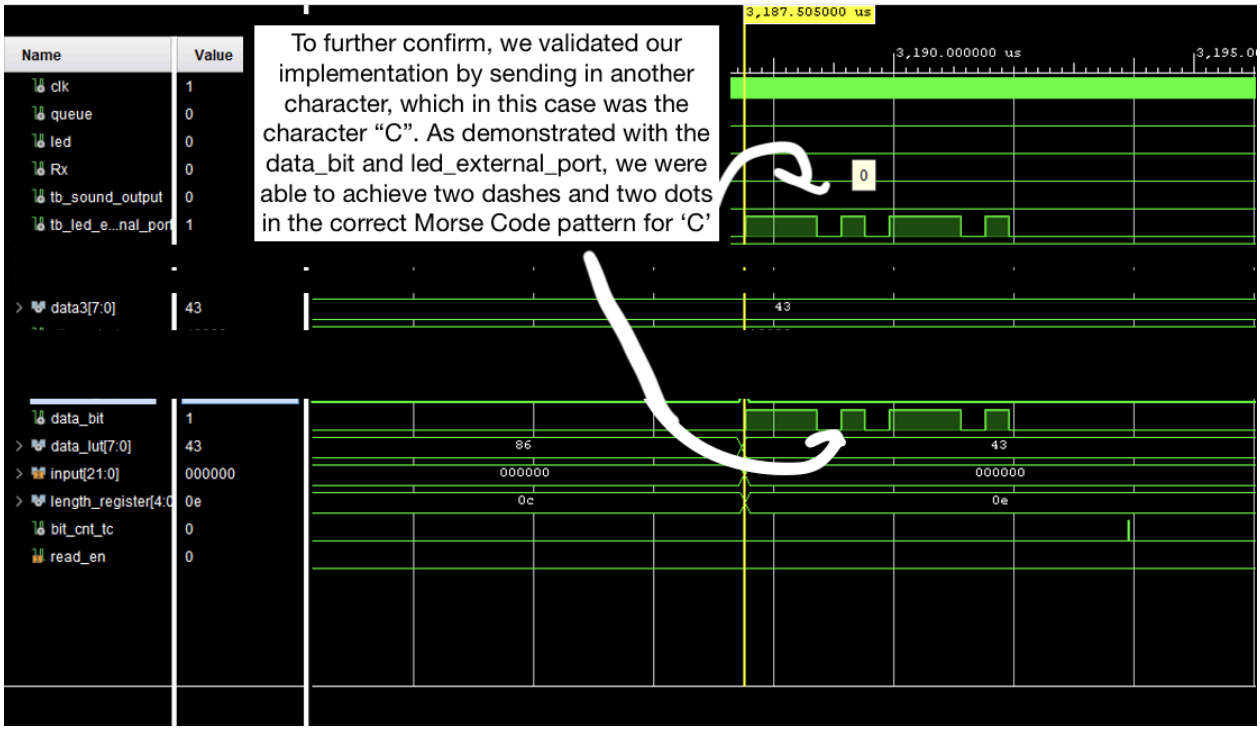


Figure 8: Waveform that demonstrates the output when the character ‘C’ is inputted

In the waveforms above, we tested our design by sending in the characters ‘S’ and ‘C’, respectively. We were able to achieve the correct number of dashes and dots associated with each character as demonstrated with the data_bit signal. As a result, we were able to validate that our design should properly operate correctly. However, to further confirm its functionality, we tested it with hardware.

3.6.2 Hardware Validation

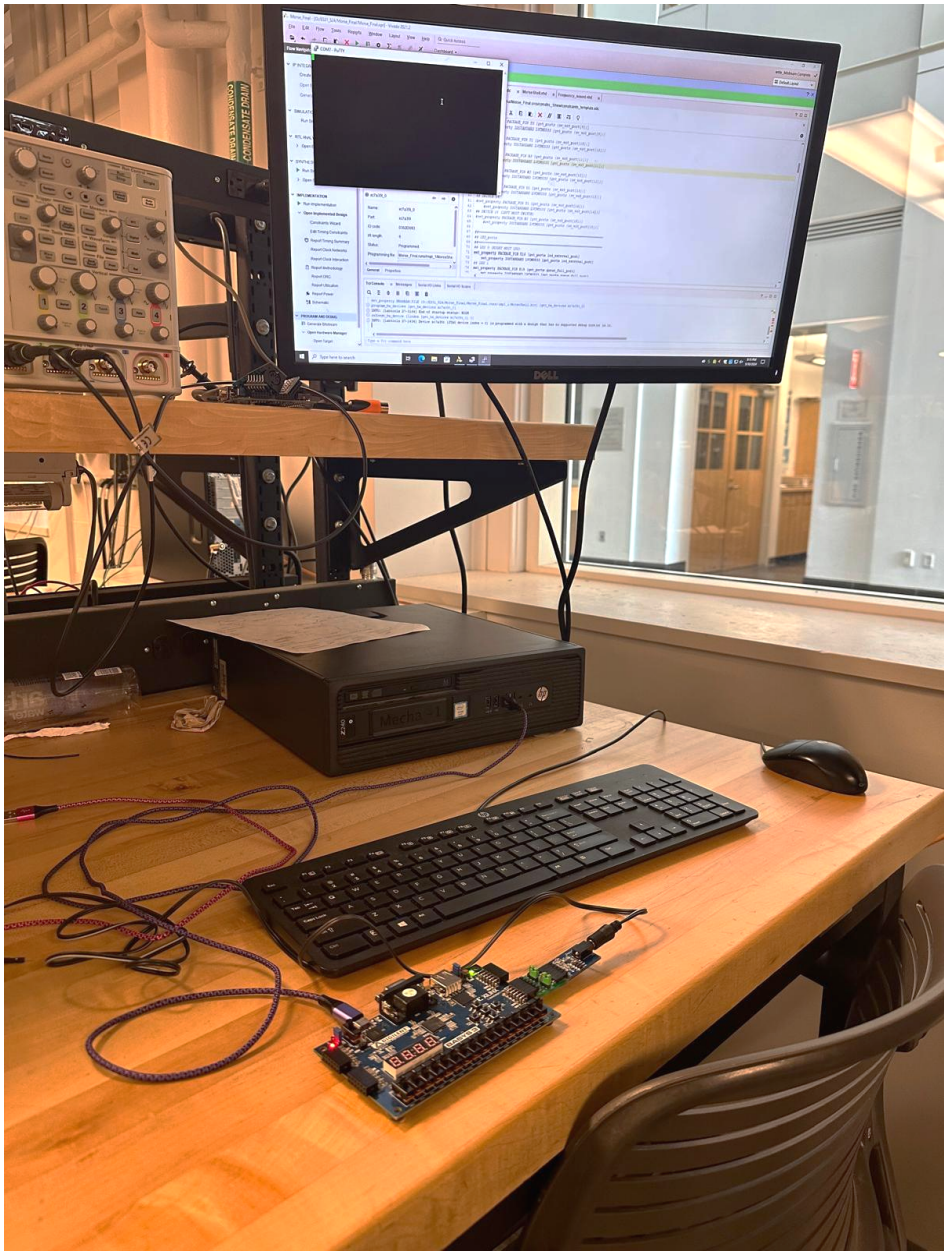
The primary objective of this hardware validation was to confirm the functionality of the Morse code translation system, which had been previously demonstrated in simulation. The validation was performed by integrating the system with real-world interfaces and observing its operation under controlled test conditions. After we had validated our design by checking with

testbenches, we generated a bitstream to validate it on hardware. Additionally, we verified that we were getting the correct signals for each character through using the oscilloscope.

Hardware Validation Result:

Below is a short video showcasing the hardware testing process of single characters.

- **Audio & Visual Testing Video:** https://youtube.com/shorts/1OVOCM7OwTM



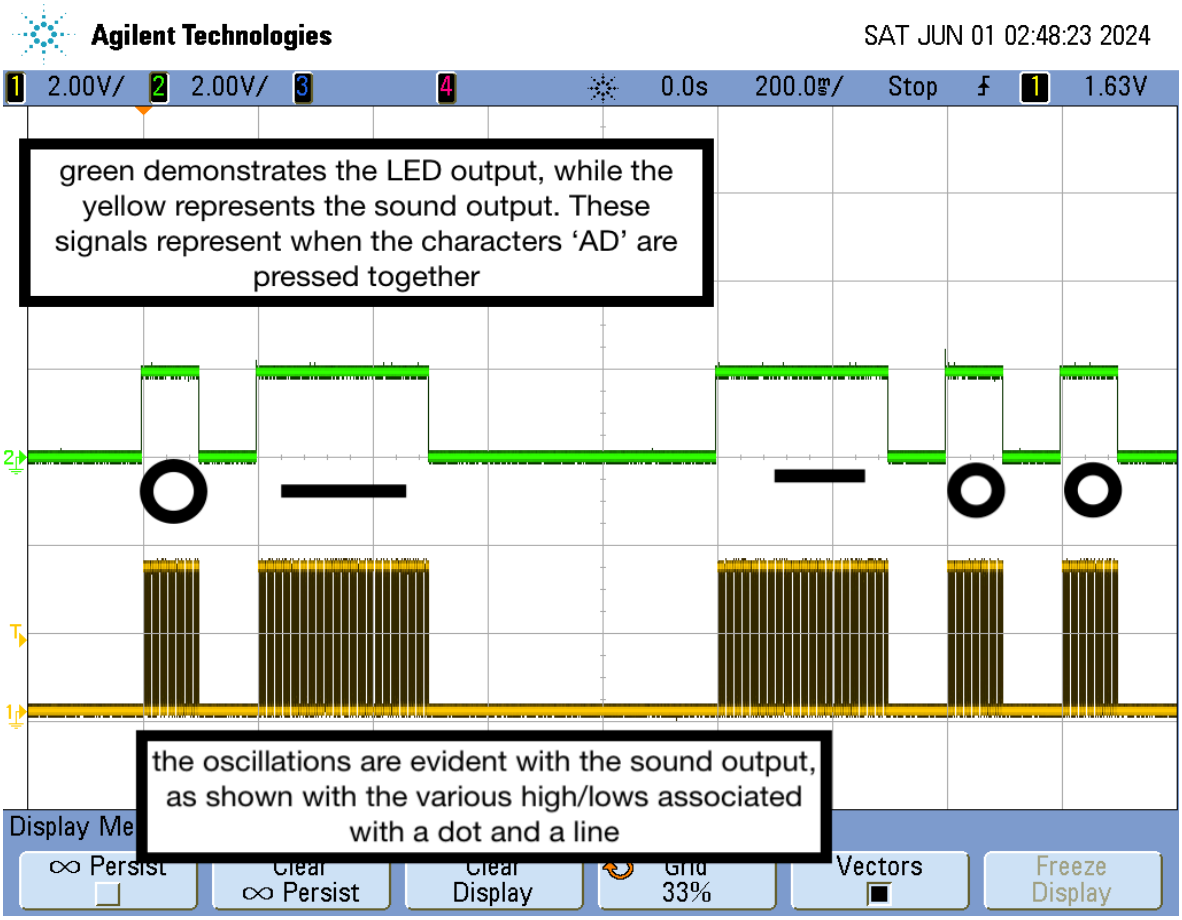


Figure 9: *The scope screenshot demonstrates the signals achieved when 'AD' is pressed on the keyboard. Green represents what the LED would get, while the yellow signal is what the sound output would be.*

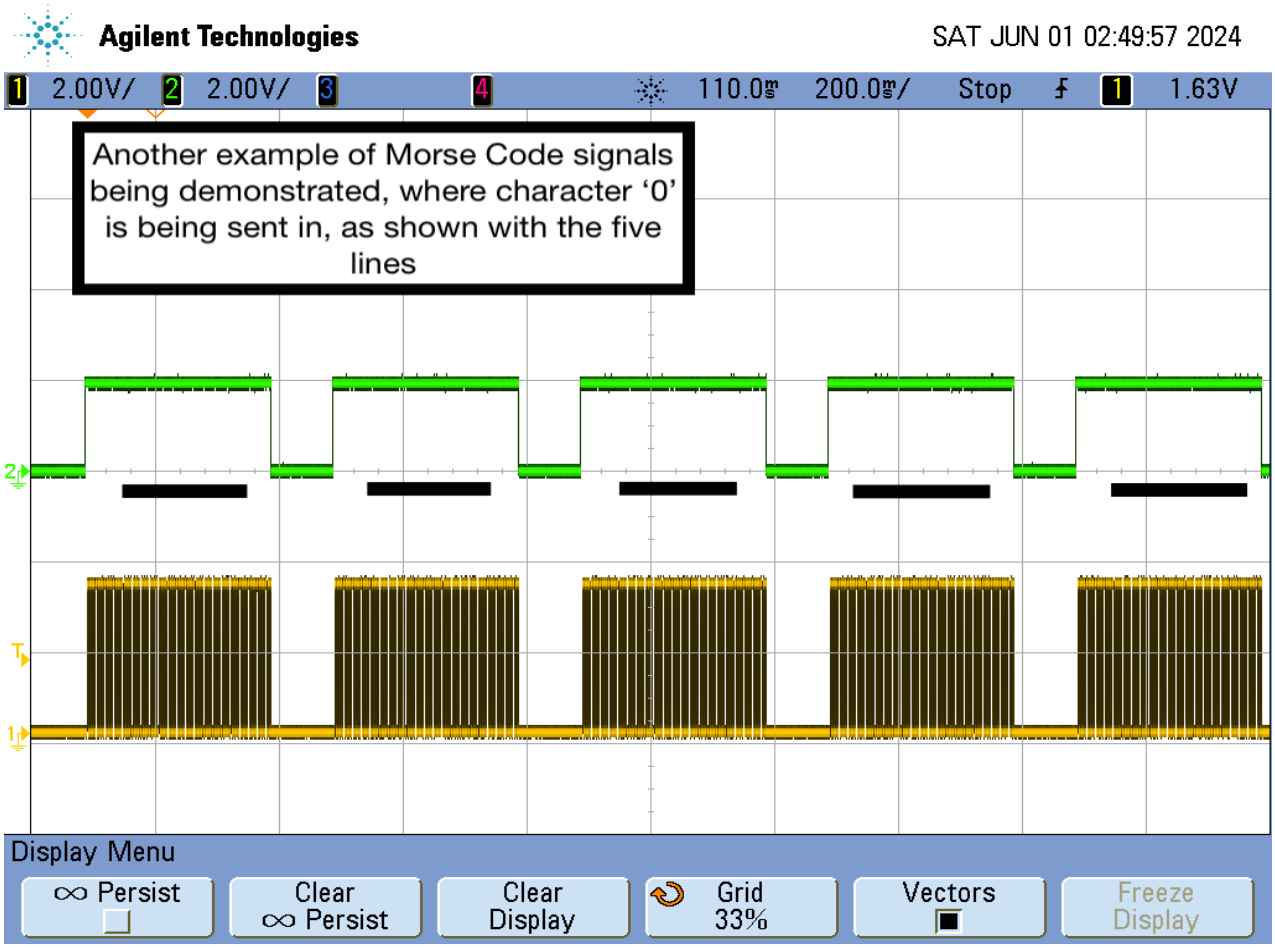


Figure 10: *The scope screenshot demonstrates the output for '0' being sent in*

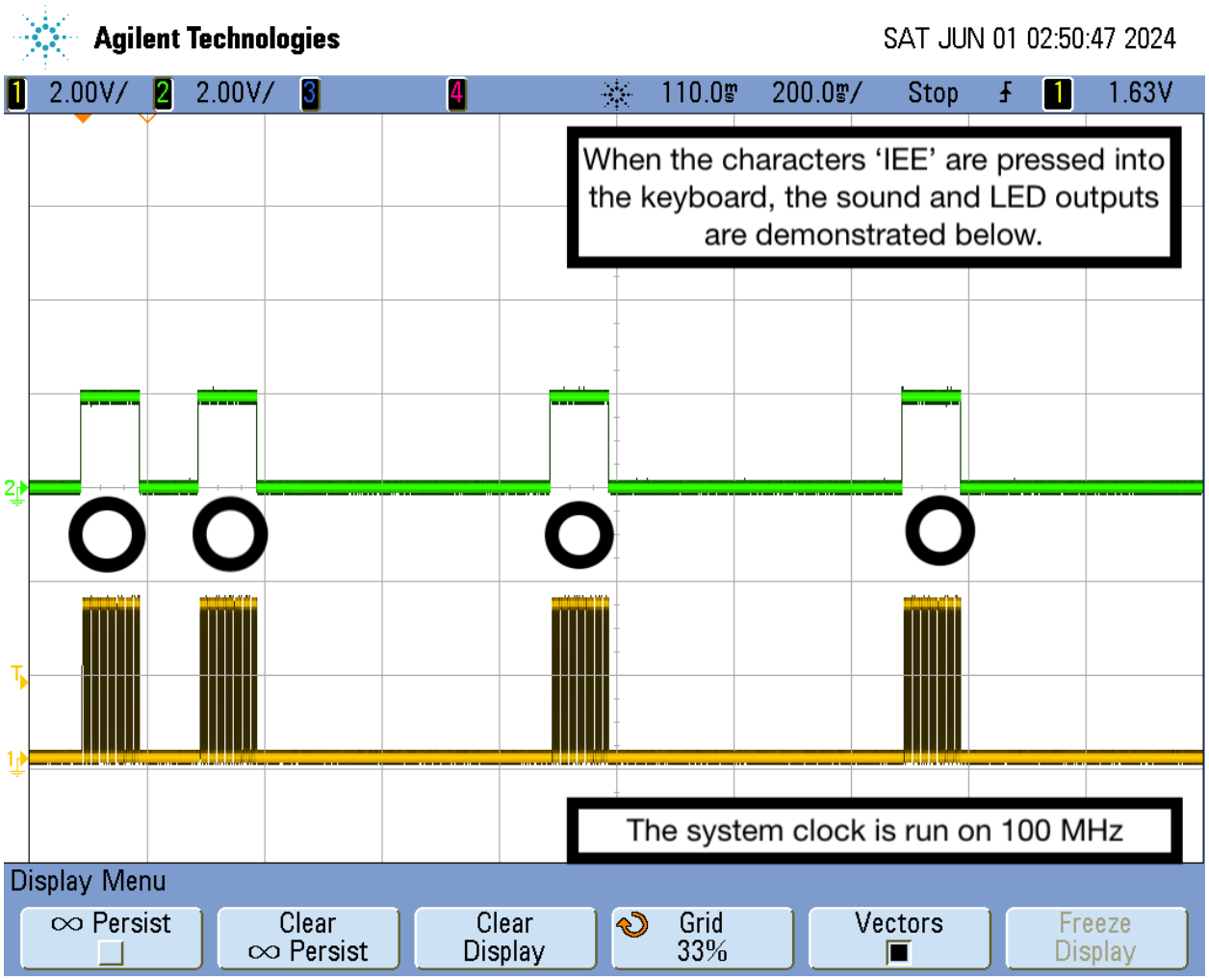
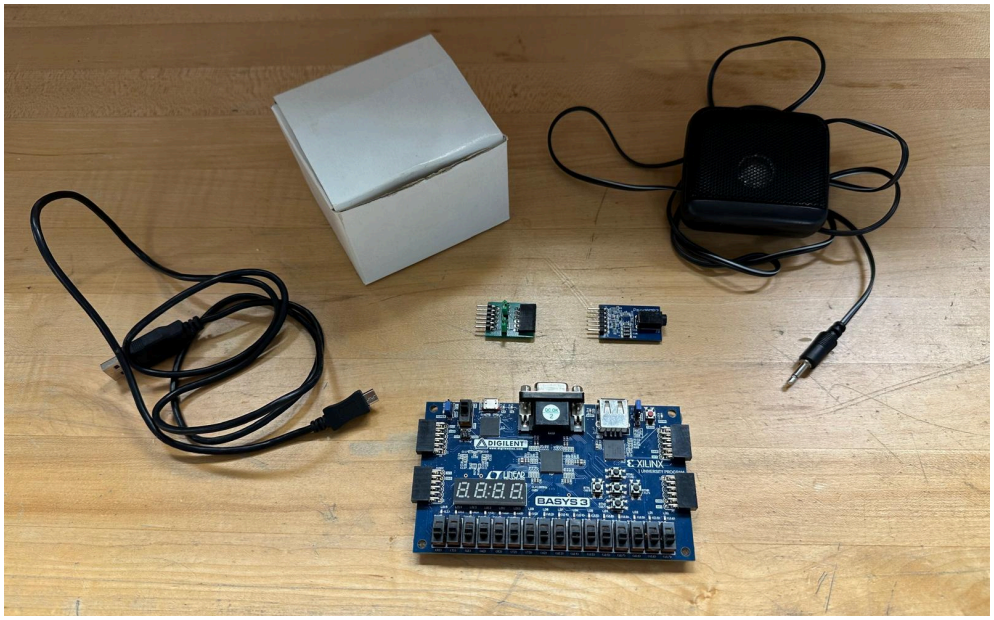


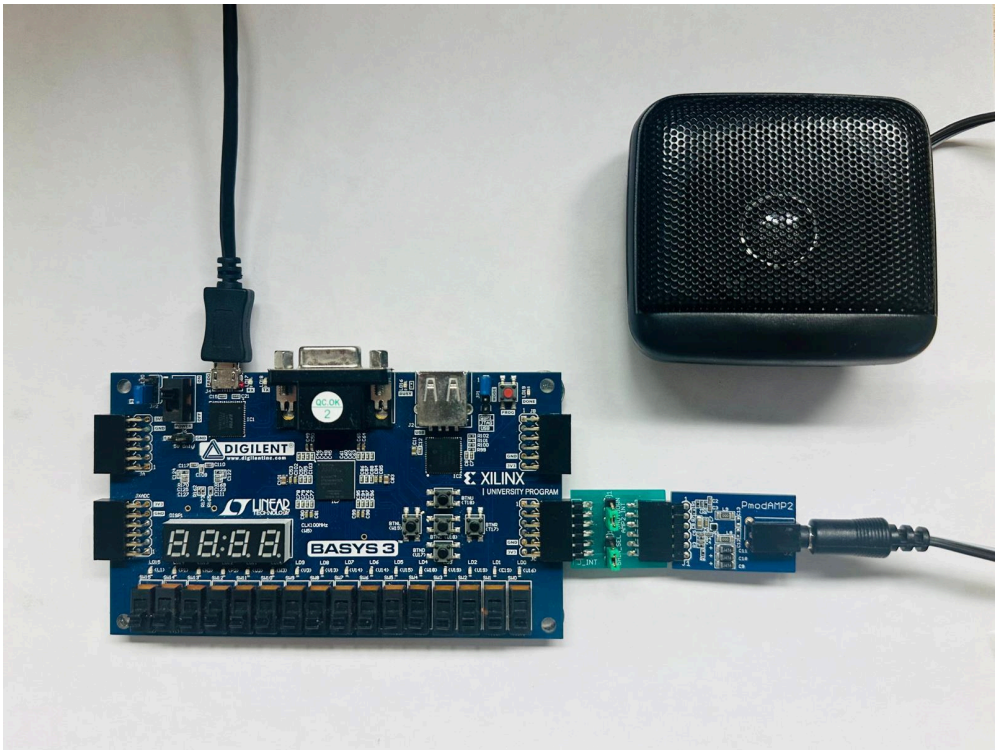
Figure 11: *The scope screenshot demonstrates the output for 'IEE' being sent in*

While the screenshots only demonstrate a few of the characters, we made sure to check for each character to confirm that the Morse Lookup Table had no mistakes and the signals were demonstrated correctly.

After validating the signals, we connected the system to the Putty terminal interface, allowing us to input ASCII characters directly into the system. The Morse code system was configured with the necessary hardware interfaces, including a speaker for sound output and a Basys 3 Artix-7 FPGA Board for visual signaling via the LED. Connections were established between the system's output ports and the speaker and LED, as outlined in the constraints file.



Before the hardware development stage.



After the hardware development stage and validation confirmation.

Interface Configuration:

Putty was set up to communicate with the system, allowing for the input of characters. Once the system was powered on, we programmed the device and started testing to ensure all connections were secure and functioning.

Individual characters were inputted into the system via Putty to test the translation and output accuracy. Each letter or number was entered sequentially, and the corresponding Morse code was observed through both the speaker and LED output. Our first and last names were also entered into the system to test the handling of string inputs and the translation into Morse code over longer sequences.

Hardware Test Results:

- **Single Character Testing:** Each letter from A to Z and number from 0 to 9 were correctly translated into Morse code. The speaker outputted the appropriate sound for each dot and dash, and the LED blinked in sync with the sound, confirming the correct operation of both audio and visual outputs.
- **Name Testing:** The system accurately processed and outputted Morse code for every name tested. (We tested “Cinay Dilibal”, “Claire Yu” and some extra tests were “Tad” and “Hello”) These tests demonstrated the system’s ability to handle varied input lengths without errors and delay.

The Morse code system was successfully validated through testing using both single characters and full names. The functionality observed in the simulation was confirmed in a real-world environment, with both the speaker and LED outputs performing as expected.

4. Analysis of Design

The project was successful due to good time management and teamwork. We completed most of our coding about half a week before the deadline, which allowed for extensive debugging and testing. We coordinate our schedules effectively, ensuring that if one partner had classes, the other would start working early and we would alternate as needed to keep making progress together. We also utilized office hours well, kept in close communication with our project TA, and regularly incorporated feedback from Tad. Along the way, we documented our code and processes thoroughly and helped each other address bugs.

We faced some challenges, especially with the FSM logic of our shift register. While we had a clear state diagram layout in mind, we needed to debug the signals several times to make sure everything worked correctly. Timing was another issue, as sometimes signals asserted a clock cycle too early or too late, which required careful adjustments to ensure accurate timing. Despite these issues, our consistent effort and dedication to the project led to us completing it successfully by the demonstration deadline.

4.1 Resource Utilization

In our project, we leveraged the USB connection of the FPGA to interface with the computer, utilizing the Putty software to interpret physical keyboard inputs as ASCII values. For audio feedback, we connected an amplifier and a speaker to one JC port on the FPGA. Visual feedback was facilitated through an LED connected to the data_out signal on another port. Additionally, to ensure compatibility with the 9,600 baud rate set by Putty, we configured the system's baud period to be 10,416. This setup was crucial for the real-time processing and display of data in our project.

4.2 Residual Warnings

After multiple debugging sessions and overall system checkups, we can confirm that there were no critical warnings related to latches in the project. The remaining warnings were associated with elements that did not influence the project's performance. To further ensure this, we checked in with Tad and he confirmed that these remaining warnings did not affect the system's performance.

4.3 Division of Labor

Both partners in this project, Cinay and Claire, demonstrated good teamwork throughout the development process. Initially, we solidified our understanding of the block diagram to prevent future issues. After detailed discussions with our project TA, Tad, and Professor Luke about the RTL diagram, we began the coding process. Tasks were evenly distributed: Cinay took on the Morse LUT, the SCI receiver testbench, and the sound frequency component and its testbench, while Claire handled the queue and its testbench, and the shifting register testbench. They worked together on the shifting register, the SCI receiver, and the top shell level and its testbench. Each component was individually tested to ensure smooth integration later, minimizing the need for extensive debugging. Despite many challenges in debugging components, adjusting control signals, and checking finite state machines, our effective collaboration and meetings every day minimized time loss and contributed to the project's success. Regular meetings, office hours, and responsive feedback helped us address any issues during implementation and simulation promptly, leading to successful project completion.

4.4 Future Work

In future developments of the project, if we were to design another complex digital system, we would incorporate a visual output on a monitor using Video Graphics Array (VGA) technology. We also discussed the possibility of enhancing the current design by adding the capability to decode Morse code. Currently, our project only converts ASCII characters into Morse code. We would aim to extend this by enabling the system to convert Morse code back into ASCII characters. Additionally, since we both enjoy visual elements, we would like to develop this project into something that displays visual Morse code data flow.

Acknowledgments

We would like to thank Tad, the wonderful TA's and Professor Luke for providing us guidance and helping throughout the design process. We definitely could not have developed a functional Morse Code Converter without their assistance.

Conclusions

All in all, we were able to successfully design a Morse Code Converter through pursuing a top-down design with a bottom-up implementation strategy. Although we definitely faced many issues throughout the process, we also learned a lot from this entire experience. If we were given another chance to design a complex digital system, we would choose to generate a visual output to a monitor using a Video Graphics Array. As we both love playing video games, building a game using our knowledge from ENGS31 would definitely be a lot of fun. In terms of advice for other students, we recommend starting the project early. One big recommendation is getting the block and RTL diagrams checked out with the TAs and Professors before writing code to make sure that the design can be implemented.

Appendix A: VHDL Source Code

5.1 SCI Receiver

```
----------------------------------------------------------------------------
-- Create Date: 05/22/2024

-- Course: ENGS31/COSC56 - Spring 2024
-- Project Name: SCI Receiver
-- Project Partners: Cinay Dilibal & Claire Yu

-- Description: SCI Receiver that takes in the user inputted data,
and sends it out to the queue
-- to be stored.

----------------------------------------------------------------------------

library IEEE;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_1164.ALL;

entity SCI_Reciever is
     Port (
            sci_clk     : in std_logic;
            data_in     : in std_logic;
            data_out    : out std_logic_vector(7 downto 0);
            data_done   : out std_logic );
end SCI_Reciever;

architecture behavior of SCI_Reciever is

-- Baud Counter Signals
signal BC_enable   : std_logic := '0';
signal BC_clear    : std_logic := '0';
signal Baud_Count  : integer := 0;

constant Baud_Rate : integer := 9600;
```

```
constant Baud_Period : integer := 10416;--10416; -- Baud Period (100
MHz / 9600 = 10416)

-- Internal Signals
signal data : unsigned(9 downto 0) := (others => '0');
signal shift_en : std_logic := '0';
signal data_ready : std_logic := '0';

-- Controller
type State_Type is (idle, start_count, hold, shift, ready, stopbit);
signal current_state, next_state : State_Type := idle;

-- Bit Counter Signals
signal BitCounter_en : std_logic := '0';
signal BitCounter_clr : std_logic := '0';


signal BitCounter : integer := 0;
signal Max_count : integer := 10;

begin

State_Update : process(sci_clk)
begin

    if rising_edge(sci_clk) then

        current_state <= next_state;

        -- baud counter enabler, starts the baud counter
        if BC_enable = '1' then
            Baud_Count <= Baud_Count + 1;
        elsif BC_clear = '1' then
            Baud_Count <= 0;
        end if;

        -- if the shift is high, start shifting data
        if shift_en = '1' then
            data <= data_in & data(9 downto 1); -- shifting logic
```

```
        end if;

        -- bit counter enabler; starts the bit count
        if BitCounter_en = '1' then
            BitCounter <= BitCounter + 1;
        elsif BitCounter_clr = '1' then
            BitCounter <= 0;
        end if;

    end if;

end process State_Update;

-- finite state machine
SCI_FSM : process(current_state, data_in, Baud_Count, BitCounter,
Max_count)
begin

-- defaults
shift_en <= '0';
BC_clear <= '1';
BC_enable <= '0';
data_ready <= '0';
BitCounter_en <= '0';
BitCounter_clr <= '1';

next_state <= current_state;

    case current_state is
        when idle => -- until the start bit is entered
            if data_in = '0' then
                next_state <= start_count;
            end if;

           -- get the start bit to be shifted out
           -- wait for half a baud period
        when start_count =>
            BC_clear <= '0';
            BC_enable <= '1';
```

```
        BitCounter_en <= '0';
        BitCounter_clr <= '0';

        if Baud_Count = (Baud_Period/2) - 1 then
            next_state <= shift;
        end if;

    -- hold and shift for 8 bits
    when hold =>
        BC_clear <= '0';
        BC_enable <= '1';

        BitCounter_en <= '0';
        BitCounter_clr <= '0';

        shift_en <= '0';

        if BitCounter = Max_count-1 then
            next_state <= ready;

        elsif Baud_Count = Baud_Period-1 then
            next_state <= shift;
        end if;

    when shift =>
        shift_en <= '1';

        BC_clear <= '1';
        BC_enable <= '0';

        BitCounter_en <= '1';
        BitCounter_clr <= '0';

        next_state <= hold;

    -- state to let us know that 9 bits have been shifted
    when ready =>
        data_ready <= '1';
```

```
            next_state <= stopbit;

        -- get the stop bit as well
        when stopbit =>
            BC_clear <= '0';
            BC_enable <= '1';
            if Baud_count = (Baud_Period/2) then
                next_state <= idle;
            end if;


   end case;
end process SCI_FSM;

-- output logic
data_done <= data_ready;
data_out <= std_logic_vector(data(9 downto 2)); -- Ignore first and
last bits

end behavior;
```

5.2 Memory Queue

```
----------------------------------------------------------------------
-------------
-- Create Date: 05/22/2024

-- Course: ENGS31/COSC56 - Spring 2024
-- Project Name: Queue
-- Project Partners: Cinay Dilibal & Claire Yu

-- Description: The Queue that stores all the characters and then
pops them off
-- one-by-one

----------------------------------------------------------------------
-------------
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

-- Entity of the queue
ENTITY Queue IS
PORT (     clk         :     in    STD_LOGIC; --100 MHz clock
           Write :     in    STD_LOGIC;
           read  :     in    STD_LOGIC;
        Data_in  :     in    STD_LOGIC_VECTOR(7 downto 0);
           Data_out:   out   STD_LOGIC_VECTOR(7 downto 0);
        Empty    :     out   STD_LOGIC;
        Full     :     out   STD_LOGIC);
end Queue;


architecture behavior of Queue is

CONSTANT QUEUE_LENGTH : integer := 8;

-- memory block that is 8 by 8
type regfile is array(0 to QUEUE_LENGTH-1) of STD_LOGIC_VECTOR(7
downto 0);
signal Queue_reg : regfile := (others =>(others => '0'));

-- internal signals
signal W_ADDR : integer := 0;
signal R_ADDR : integer := 0;
signal Element_cnt : integer := 0;
signal Full_sig, Empty_sig : std_logic := '0';

BEGIN

-- clk process
process(clk, Element_cnt)
begin

     -- rising edge of the clock
     if rising_edge(clk) then
```

```
    --Basic Queue Logic
     if (Write = '1') and (Full_sig = '0') then
           Queue_reg(W_ADDR) <= Data_in;
            if W_ADDR = QUEUE_LENGTH-1 then
                W_ADDR <= 0;
                else
                W_ADDR <= W_ADDR + 1;
            end if;
        end if;

        if (read = '1') and (Empty_sig = '0') then
           Queue_reg(R_ADDR) <= (others => '0');
           if R_ADDR = QUEUE_LENGTH-1 then
                R_ADDR <= 0;
                else
                R_ADDR <= R_ADDR + 1;
            end if;
        end if;


        --Keep track of number of elements to signal Empty or Full.
Note that this could easily be embedded into the if/else statements
above. I just wanted to make it clear that this counter is separate
hardware.
        if (Write = '1') and (Full_sig = '0') then
           Element_cnt <= Element_cnt + 1;
        elsif (read = '1') and (Empty_sig = '0') then
           Element_cnt <= Element_cnt - 1;
        end if;
    end if;

    --Comparitors for Full_sig and Empty_sig
    Full_sig <= '0';
    Empty_sig <= '0';
    if Element_cnt = QUEUE_LENGTH then
      Full_sig <= '1';
    elsif Element_cnt = 0 then
      Empty_sig <= '1';
```

```
    end if;

end process;

Full <= Full_sig;
Empty <= Empty_sig;
Data_out <= Queue_reg(R_ADDR);

end behavior;
```

5.3 Morse Lookup Table

```
----------------------------------------------------------------------------
-- Create Date: 05/22/2024

-- Course: ENGS31/COSC56 - Spring 2024
-- Project Name: Morse Lookup Table
-- Project Partners: Cinay Dilibal & Claire Yu

-- Description: The lookup table for converting from ASCII characters
to Morse Code, in order to be demonstrated on the LED and
-- produce a sound output that follows the Morse Code signals. Each
corresponding character has 22 bits,
-- with the length of each character standardized to 5 bits long.

----------------------------------------------------------------------------

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

-- entity block
entity Morse_LUT is
    Port ( clk          : in STD_LOGIC;
           data_out     : in STD_LOGIC_VECTOR(7 downto 0);
           output       : out STD_LOGIC_VECTOR(21 downto 0);
```

```
            length          : out STD_LOGIC_VECTOR(4 downto 0));
end Morse_LUT;


-- architecture
architecture Behavioral of Morse_LUT is


begin

-- look up table
LUT : process(clk)
begin
    if rising_edge(clk) then

        case data_out is

            -- Letters (A to Z)
            when "01000001" =>
                output <= "10111000" & "00000000000000" ; --A
                length <= STD_LOGIC_VECTOR(to_unsigned(8,5)); -- to
hold the length of the output
            when "01000010" =>
                output <= "111010101000" & "0000000000" ; --B
                length <= STD_LOGIC_VECTOR(to_unsigned(12,5));
            when "01000011" =>
                output <= "11101011101000" & "00000000" ; --C
                length <= STD_LOGIC_VECTOR(to_unsigned(14,5));
            when "01000100" =>
                output <= "1110101000" & "000000000000" ; --D
                length <= STD_LOGIC_VECTOR(to_unsigned(10,5));
            when "01000101" =>
                output <= "1000" & "0000000000000000" ; --E
                length <= STD_LOGIC_VECTOR(to_unsigned(4,5));
            when "01000110" =>
                output <= "101011101000" & "0000000000" ; --F
                length <= STD_LOGIC_VECTOR(to_unsigned(12,5));
            when "01000111" =>
                output <= "111011101000" & "0000000000" ; --G
```

```
            length <= STD_LOGIC_VECTOR(to_unsigned(12,5));
        when "01001000" =>
            output <= "1010101000" & "000000000000" ; --H
            length <= STD_LOGIC_VECTOR(to_unsigned(10,5));
        when "01001001" =>
            output <= "101000" & "0000000000000000" ; --I
            length <= STD_LOGIC_VECTOR(to_unsigned(6,5));
        when "01001010" =>
            output <= "1011101110111000" & "000000" ; --J
            length <= STD_LOGIC_VECTOR(to_unsigned(16,5));
        when "01001011" =>
            output <= "111010111000" & "0000000000" ; --K
            length <= STD_LOGIC_VECTOR(to_unsigned(12,5));
        when "01001100" =>
            output <= "101110101000" & "0000000000" ; --L
            length <= STD_LOGIC_VECTOR(to_unsigned(12,5));
        when "01001101" =>
            output <= "1110111000" & "000000000000" ; --M
            length <= STD_LOGIC_VECTOR(to_unsigned(10,5));
        when "01001110" =>
            output <= "11101000" & "00000000000000" ; --N
            length <= STD_LOGIC_VECTOR(to_unsigned(8,5));
        when "01001111" =>
            output <= "11101110111000" & "00000000" ; --O
            length <= STD_LOGIC_VECTOR(to_unsigned(14,5));
        when "01010000" =>
            output <= "10111011101000" & "00000000" ; --P
            length <= STD_LOGIC_VECTOR(to_unsigned(14,5));
        when "01010001" =>
            output <= "1110111010111000" & "000000" ; --Q
            length <= STD_LOGIC_VECTOR(to_unsigned(16,5));
        when "01010010" =>
            output <= "1011101000" & "000000000000" ; --R
            length <= STD_LOGIC_VECTOR(to_unsigned(10,5));
        when "01010011" =>
            output <= "10101000" & "00000000000000" ; --S
            length <= STD_LOGIC_VECTOR(to_unsigned(12,5));
        when "01010100" =>
            output <= "111000" & "0000000000000000" ; --T
```

```
            length <= STD_LOGIC_VECTOR(to_unsigned(6,5));
        when "01010101" =>
            output <= "1010111000" & "000000000000" ; --U
            length <= STD_LOGIC_VECTOR(to_unsigned(10,5));
        when "01010110" =>
            output <= "101010111000" & "0000000000" ; --V
            length <= STD_LOGIC_VECTOR(to_unsigned(12,5));
        when "01010111" =>
            output <= "101110111000" & "0000000000" ; --W
            length <= STD_LOGIC_VECTOR(to_unsigned(12,5));
        when "01011000" =>
            output <= "11101010111000" & "00000000" ; --X
            length <= STD_LOGIC_VECTOR(to_unsigned(14,5));
        when "01011001" =>
            output <= "1110101110111000" & "000000" ; --Y
            length <= STD_LOGIC_VECTOR(to_unsigned(16,5));
        when "01011010" =>
            output <= "11101110101000" & "00000000" ; --Z
            length <= STD_LOGIC_VECTOR(to_unsigned(14,5));

        -- Numbers (0 to 9)
        when "00110001" =>
            output <= "10111011101110111000" & "00" ; --1
            length <= STD_LOGIC_VECTOR(to_unsigned(20,5));
        when "00110010" =>
            output <= "1010111011101111000" & "0000" ; --2
            length <= STD_LOGIC_VECTOR(to_unsigned(18,5));
        when "00110011" =>
            output <= "10101011101111000" & "000000" ; --3
            length <= STD_LOGIC_VECTOR(to_unsigned(16,5));
        when "00110100" =>
            output <= "101010101111000" & "00000000" ; --4
            length <= STD_LOGIC_VECTOR(to_unsigned(12,5));
        when "00110101" =>
            output <= "1010101010000" & "0000000000" ; --5
            length <= STD_LOGIC_VECTOR(to_unsigned(12,5));
        when "00110110" =>
            output <= "11101010101000" & "00000000" ; --6
            length <= STD_LOGIC_VECTOR(to_unsigned(14,5));
```

```
            when "00110111" =>
                output <= "1110111010101000" & "000000" ; --7
                length <= STD_LOGIC_VECTOR(to_unsigned(16,5));
            when "00111000" =>
                output <= "1110111011101010000" & "0000" ; --8
                length <= STD_LOGIC_VECTOR(to_unsigned(18,5));
            when "00111001" =>
                output <= "11101110111011101000" & "00" ; --9
                length <= STD_LOGIC_VECTOR(to_unsigned(20,5));
            when "00110000" =>
                output <= "11101110111011101110000";       --0
                length <= STD_LOGIC_VECTOR(to_unsigned(22,5));

            when "00100000" =>
                output <= "000000" & "0000000000000000" ; --Space
                length <= STD_LOGIC_VECTOR(to_unsigned(6,5));

            when others =>
                output <= "00000000000000000000";
                length <= STD_LOGIC_VECTOR(to_unsigned(5,5));
        end case;
    end if;
end process LUT;

end Behavioral;
```

5.4 Shift Register

```
----------------------------------------------------------------------
-------------
 -- Create Date: 05/23/2024

 -- Course: ENGS31/COSC56 - Spring 2024
 -- Project Name: Shift Register
 -- Project Partners: Cinay Dilibal & Claire Yu

 -- Description: Shift register conversion after reading in from the
```

```
LUT and sending out
 -- to the LED/Sound


----------------------------------------------------------------------
-------------

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

 -- entity block
 entity Shift_reg is
      Port ( clk          : in STD_LOGIC;
             read_en        : in STD_LOGIC;
             length         : in STD_LOGIC_VECTOR(4 downto 0);
             Tx_done  : out STD_LOGIC;
             bit_out        : out STD_LOGIC;
             input       : in STD_LOGIC_VECTOR(21 downto 0));

 end Shift_reg;

 architecture Behavior of Shift_reg is

 -- internal signals
-- dip counter is similar to the baud counter, for giving enough time
for data to be sent
-- one by one, and then bit count lets us know how many bits have
been shifted.
 signal dip_TC : STD_LOGIC := '0';
 signal bit_cnt_tc : STD_LOGIC := '0';

 signal shift_reg : std_logic_vector(21 downto 0) :=
"0000000000000000000000";

 signal bit_count : unsigned(4 downto 0) := (others => '0');  -- in
order to count up to the correct number of length

 constant DIP_PERIOD : integer := 1000000; --10000000
```

```
-- in order to use the length input, must make an internal signal
 signal length_register : STD_LOGIC_VECTOR(4 downto 0) := (others =>
'0') ;

 signal shift_en : std_logic := '0';
 signal load_en : std_logic := '0';

 signal dip_enable : std_logic := '0';

 signal DIP_Counter : unsigned(25 downto 0) := (others => '0');

 --STATES-- for the FSM logic
 type state_type is (Idle, Load, StartBit, Shift, sWait, bitOu);

 -- control signals
 signal current_state, next_state : state_type := Idle;

 begin

 -- process for the clock
 state_update : process(clk)
 begin
     if rising_edge(clk) then
           current_state <= next_state;
     end if;
 end process state_update;


 -- next state logic process
 next_state_logic: process(current_state, read_en, bit_count, dip_TC,
bit_cnt_tc, length_register)
 begin

 -- default
 shift_en <= '0';
 load_en <= '0';
 dip_enable <= '0';
```

```
 next_state <= current_state;

 case current_state is
      when Idle =>

         -- when data can be read
      if read_en = '1' then
             next_state <= Load;
          end if;
     when Load =>
      -- load in the data and start with the first bit
          load_en <= '1';
          next_state <= StartBit;

     when StartBit =>
        if dip_TC = '1' then
            next_state <= shift;
        end if;

       dip_enable <= '1';

     when Shift => -- shift bits and continue until all bits are
shifted
          shift_en <= '1';

          next_state <= sWait;
     when sWait =>  -- in this state we enable dip count, and have it
reach its max
     -- to start a bit increase
      if dip_TC = '1'and  bit_count /= unsigned(length_register)+ 1
then
             next_state <= Shift;
          end if;
        dip_enable <= '1';

          if bit_cnt_tc = '1' then
             next_state <= idle;
          end if;
```

```
    when others =>
        next_state <= Idle;
end case;

end process next_state_logic;


datapath : process (clk, DIP_Counter)
begin
    if rising_edge(clk)then

        --Dip Counter
        if dip_enable = '1' then -- enable bit for dip count
           DIP_Counter <= DIP_Counter + 1;
        end if;
        if dip_TC = '1' then -- reset when TC is equal to 1
            DIP_Counter <= (others => '0');
        end if;
        if read_en = '1' then -- reset dip count
          DIP_Counter <= (others => '0');

        end if;
    end if;

      --Shift Register
    if rising_edge(clk) then
        if shift_en = '1' then -- start the shift
          shift_reg <= shift_reg(20 downto 0) & '0' ; --shift the
bits and add an idle bit to the MSB

        end if;

          length_register <= length_register; -- remember to set
length to itself so it does not reset
       if load_en = '1' then -- if load is enabled make sure to set
the shift reg for input and length reg to length
         shift_reg <= input;
         length_register <= length;
```

```
        end if;

    end if;


    --asynchronous TC
    dip_TC <= '0';
    if DIP_Counter = DIP_PERIOD-1 then
          dip_TC <= '1';
    end if;
 end process datapath;

 -- bit counter process
 bit_counter: process(clk, bit_count, length_register) -- if not in
clock process, but in asynchronous
 begin
     if rising_edge(clk) then
     if read_en = '1' then
          bit_count <= (others => '0'); --reset to the bit count
       end if;
        if dip_TC = '1' then -- if dip TC is enabled
            if bit_cnt_tc = '0' then -- either increment bit count
                 bit_count <= bit_count + 1;
            end if;
        end if;
            if bit_cnt_tc = '1' then -- bit count TC should reset
               bit_count <= (others => '0');
            end if;


    end if;

  -- asynchronous process
        bit_cnt_tc <= '0';
       if (bit_count = unsigned(length_register) + 2) then
           bit_cnt_tc <= '1';
       end if;

 end process bit_counter;
```

```
 -- output logic
 bit_out <= shift_reg(21); --gets the last bit
 Tx_done <= bit_cnt_tc; -- if weare on the final bit and the
approproate time has elapsed

 end Behavior;
```

5.5 Sound Frequency Output

```
----------------------------------------------------------------------
---------------
--- Create Date: 05/31/2024

--- Course: ENGS31/COSC56 - Spring 2024
--- Project Name: Frequency Sound
--- Project Partners: Cinay Dilibal & Claire Yu
--- Description: The block for the Frequency sound in order to
produce the sound
-- out of the morse code signal
----------------------------------------------------------------------
---------------

----------------------------------------------------------------------
-------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Frequency_sound is
    Port ( clk : in STD_LOGIC;  -- 100 MHz clock
        bit_in    : in     STD_LOGIC;
           frequency : out std_logic;
           LED_out   : out std_logic);
end Frequency_sound;


architecture Behavioral of Frequency_sound is
```

```
-- signal process
signal count : integer := 0;
signal period : integer := 23200; -- 100MHz
signal output : std_logic := '0';
begin

-- clock process
process(clk)
begin
if rising_edge(clk) then
    if count = period/2 then
        output <= not output;
    end if;

    if count = period then
        count <= 0;
        output <= not output;
    else
        count <= count + 1;
    end if;

end if;
end process;

-- set the outputs
frequency <= output;
LED_out <= bit_in;

end Behavioral;
```

5.6 Constraints File

```
## This file is a general .xdc for the Basys3 rev B board for
ENGS31/CoSc56
## 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
## - these names should match the external ports (_ext_port) in the
entity declaration of your shell/top level

##============================================================================
=
## External_Clock_Port
##============================================================================
=
## This is a 100 MHz external clock
set_property PACKAGE_PIN W5 [get_ports clk_ext_port]

     set_property IOSTANDARD LVCMOS33 [get_ports clk_ext_port]
     create_clock -add -name sys_clk_pin -period 10.00 -waveform {0
5} [get_ports clk_ext_port]

##============================================================================
=
## LED_ports
##============================================================================
=
## LED 0 (RIGHT MOST LED)
set_property PACKAGE_PIN U16 [get_ports led_external_port]

     set_property IOSTANDARD LVCMOS33 [get_ports led_external_port]
## LED 1
set_property PACKAGE_PIN E19 [get_ports queue_full_port]

     set_property IOSTANDARD LVCMOS33 [get_ports queue_full_port]
## LED 2
#set_property PACKAGE_PIN U19 [get_ports sound_output]

#     set_property IOSTANDARD LVCMOS33 [get_ports sound_output]

##============================================================================
=
## USB-RS232 Interface
##============================================================================
=
```

```
set_property PACKAGE_PIN B18 [get_ports SCI_Rx]

     set_property IOSTANDARD LVCMOS33 [get_ports SCI_Rx]

##=============================================================================
=
## Implementation Assist
##=============================================================================
=
## These additional constraints are recommended by Digilent, DO NOT
REMOVE!
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]

set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]
```

Appendix B: VHDL Testbenches

5.1 SCI Receiver

```
----------------------------------------------------------------------------
-------------
-- Create Date: 05/22/2024

-- Course: ENGS31/COSC56 - Spring 2024
-- Project Name: Morse Code Encoder
-- Project Partners: Cinay Dilibal & Claire Yu
```

```
-- Description: Testbench for the SCI Receiver
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity SCI_Reciever_tb is
end SCI_Reciever_tb;

architecture testbench of SCI_Reciever_tb is

 -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT SCI_Reciever
    PORT(
        sci_clk     : in std_logic;
            data_in     : in std_logic;
            data_out    : out std_logic_vector(7 downto 0);
            data_done   : out std_logic);
    END COMPONENT;

    -- inputs and outputs
    signal      clk         :     STD_LOGIC; --10 MHz clock
    signal      data_in     :     STD_LOGIC  := '0';
    signal      data_done :       std_logic;
    signal      data_out:  STD_LOGIC_Vector(7 downto 0) :=
"00000000";

    signal letter : std_logic_vector(7 downto 0):= "01000011"; --
receieving the letter C


    -- Clock period definitions
    constant clk_period : time := 10 ns; -- 100 MHz
    constant baud_period : time := 100 ns;--10.4us; -- 9600 baud

BEGIN
```

```
    -- Instantiate the Unit Under Test (UUT)
    uut: SCI_Reciever PORT MAP (
        sci_clk     => clk,
        data_in => data_in,
        data_done    => data_done,
        data_out      => data_out
    );

     -- Clock process definitions
    clk_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin

     -- Scenario 1: reading in
     wait for clk_period;

        data_in <= '0'; -- start bit
        wait for baud_period;

        data_in <= '0';
        wait for baud_period;

        data_in <= '1';
        wait for baud_period;

           data_in <= '0';
        wait for baud_period;

        data_in <= '0';
        wait for baud_period;
```

```
        data_in <= '1';
        wait for baud_period;

        data_in <= '1';
        wait for baud_period;

        data_in <= '1';
        wait for baud_period;

           data_in <= '1';
        wait for baud_period;



        data_in <= '1'; -- stop bit

         -- Wait for transmission to complete
        wait for clk_period * 5;


        -- Add more scenarios as necessary

        -- Complete the simulation
        wait;
    end process;

END testbench;
```

5.2 Memory Queue

```
--------------------------------------------------------------------------------
-- Create Date: 05/22/2024

-- Course: ENGS31/COSC56 - Spring 2024
-- Project Name: Morse Code Encoder
-- Project Partners: Cinay Dilibal & Claire Yu
```

```
-- Description: A testbench for the Queue component.

----------------------------------------------------------------------------

-- libraries
library IEEE;
use IEEE.std_logic_1164.all;

entity Queue_tb is
end Queue_tb;

architecture testbench of Queue_tb is

-- component part of the queue
component Queue IS
PORT (     clk         :     in    STD_LOGIC; --10 MHz clock
           Write :     in    STD_LOGIC;
           read  :     in    STD_LOGIC;
        Data_in  :     in    STD_LOGIC_VECTOR(7 downto 0);
           Data_out:   out   STD_LOGIC_VECTOR(7 downto 0);
        Empty    :     out   STD_LOGIC;
        Full     :     out   STD_LOGIC);
end component;


-- internal signals
signal     clk         :     STD_LOGIC; --10 MHz clock
signal     Write :     STD_LOGIC  := '0';
signal     Read  :     STD_LOGIC  := '0';
signal     Data_in     :     STD_LOGIC_Vector(7 downto 0) :=
"00000000";
signal     Data_out:   STD_LOGIC_Vector(7 downto 0) := "00000000";
signal     Empty :     STD_LOGIC  := '0';
signal     Full  :     STD_LOGIC  := '0';


begin
```

```
-- port mapping for queue
uut : Queue PORT MAP(
          clk  => CLK,
          Read => Read,
       Write => Write,
       Data_in => Data_in,
          Data_out => Data_out);


-- clock process
clk_proc : process
BEGIN

  CLK <= '0';
  wait for 5ns;

  CLK <= '1';
  wait for 5ns;

END PROCESS clk_proc;

-- stimulus process
stim_proc : process
begin

    wait for 15 ns;

    Data_in <= "11110000";--0xF0
    Write <= '1';
    Wait for 10 ns;
    Write <= '0';

    wait for 40 ns;
    Data_in <= "00001111";--0x0F
    Write <= '1';
    wait for 10 ns;
    write <= '0';

    wait for 40 ns;
```

```
    Data_in <= "10101010";--0xAA
    Write <= '1';
    wait for 10 ns;
    write <= '0';

     wait for 40 ns;
    Data_in <= "01010011";--0x53
    Write <= '1';
    wait for 10 ns;
    write <= '0';

    wait for 40 ns;
    Data_in <= "11010010";--0xD2
    Write <= '1';
    wait for 10 ns;
    write <= '0';

    wait for 40 ns;
    Data_in <= "11111111";--0xFF
    Write <= '1';
    wait for 10 ns;
    write <= '0';

    wait for 40 ns;
    Data_in <= "00110001";--0x31
    Write <= '1';
    wait for 10 ns;
    write <= '0';

    wait for 40 ns;
    Data_in <= "01010101";--0x55
    Write <= '1';
    wait for 10 ns;
    write <= '0';

    wait for 40 ns;
    Data_in <= "10001100";--0x8C
    Write <= '1';
    wait for 10 ns;
```

```
    write <= '0';

    wait for 40 ns; -- Read once
    read <= '1';
    wait for 10 ns;
    read <= '0';

    wait for 40 ns; -- Read once
    read <= '1';
    wait for 10 ns;
    read <= '0';

    wait for 40 ns;
    Data_in <= "11000111";--0xC7
    Write <= '1';
    wait for 10 ns;
    write <= '0';

     wait for 40 ns;
    Data_in <= "00000001";--0x01
    Write <= '1';
    wait for 10 ns;
    write <= '0';

    wait for 40 ns; -- Read once
    read <= '1';
    wait for 10 ns;
    read <= '0';

    wait for 40 ns; -- Read once
    read <= '1';
    wait for 10 ns;
    read <= '0';

    wait for 40 ns; -- Read once
    read <= '1';
    wait for 10 ns;
    read <= '0';
```

```
    wait for 40 ns; -- Read once
    read <= '1';
    wait for 10 ns;
    read <= '0';

    wait for 40 ns; -- Read once
    read <= '1';
    wait for 10 ns;
    read <= '0';

    wait for 40 ns; -- Read once
    read <= '1';
    wait for 10 ns;
    read <= '0';

    wait for 40 ns; -- Read once
    read <= '1';
    wait for 10 ns;
    read <= '0';

    wait for 40 ns; -- Read once
    read <= '1';
    wait for 10 ns;
    read <= '0';

    wait for 40 ns; -- Read once --tbe queue is already empty, so
this should do nothing.
    read <= '1';
    wait for 10 ns;
    read <= '0';

    wait for 40 ns;
    Data_in <= "11111111";--0xFF
    Write <= '1';
    wait for 10 ns;
    write <= '0';

    wait;
end process stim_proc;
```

```
end testbench;
```

5.3 Morse Lookup Table

```
----------------------------------------------------------------------------
-- Create Date: 05/22/2024

-- Course: ENGS31/COSC56 - Spring 2024
-- Project Name: Morse Code Encoder
-- Project Partners: Cinay Dilibal & Claire Yu

-- Description: Testbench for the lookup table for MorseCode.
----------------------------------------------------------------------------

-- Libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY Morse_LUT_tb IS
END Morse_LUT_tb;

ARCHITECTURE testbench OF MORSE_LUT_tb IS

    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT Morse_LUT
    PORT(
        clk      : IN  std_logic;
        data_out  : IN  std_logic_vector(7 downto 0);
        output     : OUT   STD_LOGIC_VECTOR(21 downto 0);
        length       : OUT STD_LOGIC_VECTOR(4 downto 0));
    END COMPONENT;

    --Inputs
    signal clk     : std_logic := '0';
```

```
    signal data_out : std_logic_vector(7 downto 0) := (others =>
'0');

    --Outputs
    signal output      : STD_LOGIC_VECTOR(21 downto 0);
    signal length : STD_LOGIC_VECTOR(4 downto 0);


    -- Clock period definitions
    constant clk_period : time := 100 ns; -- 10 MHz

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: MORSE_LUT PORT MAP (
        clk     => clk,
        data_out => data_out,
        output    => output,
        length      => length
    );

    -- Clock process definitions
    clk_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin

        -- Scenario 1: reading in 'A'
        data_out <= "01000001";

        -- Wait for transmission to complete
        wait for clk_period * 20;
```

```
        -- Scenario 2: Reading in '1'
        data_out <= "00110001";

        -- Wait for transmission to complete
        wait for clk_period * 20;


        -- Complete the simulation
        wait;
    end process;

END testbench;
```

5.4 Shift Register

```
--------------------------------------------------------------------------------
-- Create Date: 05/23/2024

-- Course: ENGS31/COSC56 - Spring 2024
-- Project Name: Morse Code Encoder
-- Project Partners: Cinay Dilibal & Claire Yu

-- Description: Testbench for the Shift Register

--------------------------------------------------------------------------------


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

entity Shift_reg_tb is
```

```
end Shift_reg_tb;

architecture testbench of Shift_reg_tb is

-- component for the shift reg
component Shift_reg IS
PORT (     clk          : in STD_LOGIC;
            read_en         : in STD_LOGIC;
            length          : in STD_LOGIC_VECTOR(4 downto 0);
            Tx_done    : out STD_LOGIC;
            bit_out         : out STD_LOGIC;
            input       : in STD_LOGIC_VECTOR(21 downto 0));
end component;


-- internal signals
signal     clk         :     STD_LOGIC; --10 MHz clock
signal     read_en     :     STD_LOGIC  := '0';
signal     length      :     STD_LOGIC_VECTOR(4 downto 0);
signal     Tx_done     :     STD_LOGIC  := '0';
signal     bit_out     :     STD_LOGIC;
signal     input :     STD_LOGIC_VECTOR(21 downto 0) :=
"0000000000000000000000";

constant clk_period : time := 100 ns;

begin

uut : Shift_reg PORT MAP(
           clk  => CLK,
           read_en => read_en,
        length => length,
        Tx_done => Tx_done,
           bit_out => bit_out,
        input => input);


-- clock process
clk_proc : process
```

```
BEGIN

  clk <= '0';
  wait for clk_period / 2;

  clk <= '1';
  wait for clk_period/2;
end process;


-- sitmulus process
stim_proc : process
begin

     read_en <= '0';
     length <= "10000";
    input <= "11101110101110000000000"; -- Q
    wait for clk_period;


    read_en <= '1';
    wait for clk_period*20;

    read_en <= '0';

    wait for clk_period*800;

     length <= "01000";
    input <= "11101011100000000000000"; -- K
    wait for clk_period;

    read_en <= '1';
    wait for clk_period*20;

    read_en <= '0';

    wait for clk_period*20;

    wait;
```

```
end process stim_proc;
end testbench;
```

5.5 Sound Frequency Output

```
----------------------------------------------------------------------------
-- Create Date: 05/22/2024

-- Course: ENGS31/COSC56 - Spring 2024
-- Project Name: Morse Code Encoder
-- Project Partners: Cinay Dilibal & Claire Yu

-- Description: Sound frequency generation testbench

----------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Frequency_sound_tb is
end Frequency_sound_tb;

architecture testbench of Frequency_sound_tb is

component Frequency_sound IS
PORT (
            clk         : in std_logic;
            bit_in      : in std_logic;
            LED_out     : out std_logic;
            frequency   : out std_logic );
end component;

signal clk          : std_logic := '0';
signal data_bit     : std_logic := '0';
signal sound_output : std_logic := '0';
```

```
signal LED_output   : std_logic := '0';

signal period_time : time := 10ns; -- 100MHz

begin

uut : Frequency_sound PORT MAP(
          clk  => CLK,
          bit_in => data_bit,
          LED_out => LED_output,
          frequency => sound_output
);

-- clock process
clk_proc : process
BEGIN

  clk <= '0';
  wait for period_time / 2;

  clk <= '1';
  wait for period_time/2;
end process clk_proc;

-- simulation process
stim_proc : process
begin
    data_bit <= '0';

    wait for 50ns;

    data_bit <= '0';

    wait for 100ns;

    data_bit <= '1';

    wait for 150ns;
```

```
    data_bit <= '0';

    wait for 100ns;

    data_bit <= '0';

    wait for 100ns;

    data_bit <= '1';

    wait for 100ns;

    data_bit <= '0';

    wait;

end process stim_proc;

end testbench;
```