ENGS 31 / COSC 56
Final Report:
Morse Code Decoder w/ VGA Display

Justin Sapun, Arun Guruswamy, Khanh Le, Sam Peter

Abstract

This document presents the design and implementation of a digital system using a Field Programmable Gate Array (FPGA) platform for a Morse code input and decoding game. The project focuses on user interaction through button input on the FPGA, real-time display of the decoded letters on a VGA monitor, and immediate feedback based on the accuracy of the entered Morse code.

The digital system consists of an FPGA board, a VGA display, and an audio output. The FPGA serves as the controller, decoding Morse code, and generating and sending corresponding letters to the VGA monitor. The VGA display is central to the design, mapping ASCII characters to pixel bitmaps or patterns. The button interface on FPGA enables Morse code input, while the audio output produces sound effects.

The implementation involved designing and programming the FPGA board to handle the Morse code decoding algorithm and mapping ASCII characters to the VGA display's pixel bitmap or pattern. State machines and single-port ROMs were utilized to identify Morse code patterns and map them to their respective ASCII characters. Rigorous testing and verification using machine-generated code and a 7-segment display were performed to ensure the accuracy and reliability of the decoding process and VGA display mapping.

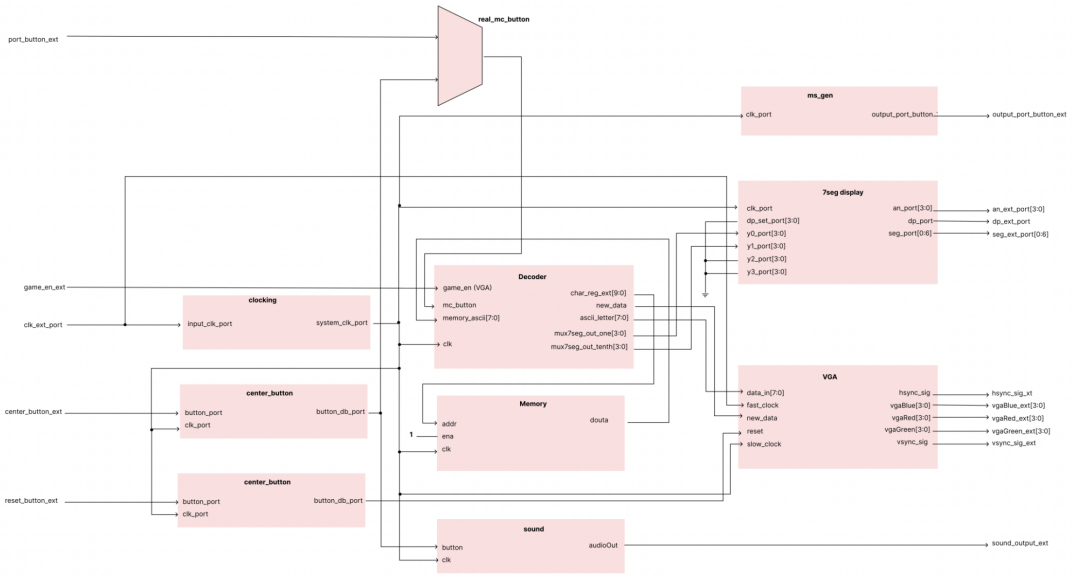
During gameplay, users input Morse code sequences by pressing buttons on the FPGA. If the entered Morse code matches the intended letter, the VGA display updates to show the corresponding letter in real-time and turns green, indicating a correct input. The system then progresses to the next letter. In the case of an incorrect Morse code entry, the VGA display turns red, signaling an error. The interactive nature of the game, coupled with the visual and audio feedback, offers an engaging and educational experience for users to learn and practice Morse code skills effectively.

1. System Overview

The overall goal of the design is to create a system that enables users to play a Morse code decoder game using an FPGA platform and VGA display. The intended behavior of the system is to allow users to see the letter cue on VGA, and input Morse code by pressing buttons on the FPGA, with the decoded letters displayed in real-time on a VGA monitor. The system provides immediate feedback based on the accuracy of the entered Morse code. The successful design achieves several specifications. It accurately decodes Morse code input, ensuring that the displayed letters correspond to the user's input. The real-time VGA display allows for instant visual feedback, showing the decoded letters as they are entered. The design also incorporates color changes on the VGA display, turning the letter green for correct input and red for incorrect input. Additionally, the system progresses to the next letter upon correct input, and the user can also move to the next letter by pressing the bottom button.

1.1) Top-level Block Diagram

Figure 1: Top-Level Block Diagram



1.2) Description of Ports

The following is a table of the inputs and outputs of our top-level shell that includes all user inputs and consequent outputs to the VGA display, Seven Seg display, Speaker, and LEDs.

I/O	Port	Description
Input	clk_ext_port	Expect a 100MHz clock defined in the constraint file.
Input	game_en_ext	Game enable logic switch (SW15) for the MC decoder.
Input	Input_switch_ext	Switch (SW14) to change between user input or machine MC. Off (0, down) is defined as center_button_ext, and On(1, up) is defined as port_button_ext.

Input	mux7seg_switch_ext	Seven Seg display enable switch (SW15).
Input	center_button_ext	User input for MC, operated on the center button.
Input	reset_button_ext	An Input signal to change the character being displayed on the screen. Operated on the bottom button.
Input	port_button_ext	Input for machine MC signal generator. Connected via wire.
Output	game_en_led	LED light (15) is enabled when the game_en_ext switch is ON. Visual feedback to know the state of the switch.
Output	mux7seg_switch_led	LED light (14) is enabled when the mux7seg_switch_ext switch is ON. Provides visual feedback.
Output	input_switch_led	LED light (13) is enabled when the Input_switch_ext switch is ON. Provides visual feedback.
Output	sound_output_ext	PWM Output for Amplifier and Speaker. Tuned to __ Hz.
Output	output_port_button_ext	The output of the machine MS signal generator. Outputs AB with a space repeatably. Can be connected to port_button_ext via wire.
Output	seg_ext_port	Segment control for Seven Seg Display. Will be changed when mux7seg_switch_ext is ON.
Output	dp_ext_port	Decimal point control for Seven Seg Display. Note that decimals were not used, so the output will always be 1111.
Output	an_ext_port	Anode/digit control for Seven Seg Display. Will cycle through each digit very fast.
Output	vsync_sig_ext	Single-bit output ports that carry the vsync signal to the VGA.
Output	hsync_sig_ext	Single-bit output ports that carry the hsync signal to the VGA.
Output	vgaRed_ext	4-bit output ports that carry the red components of the output color to the VGA.
Output	vgaGreen_ext	4-bit output ports that carry the green components of the output color to the VGA.
Output	vgaBlue_ext	4-bit output ports that carry the blue components of the output color to the VGA.

1.3) Description of Components

General

Clk Divider
- Divides the clock frequency (100MHz) to generate a lower frequency (25MHz) required for precise timing and synchronization within the system.

Button Input Conditioning
- Filters and stabilizes the button input signals to eliminate any noise or bouncing effects caused by the physical button contacts.

Morse Code

Sound Generator
- Generates audio signals based on the button inputs to produce sound effects in synchronization with user interactions.

Machine Morse Code Signal Generator
- Simulates button presses to generate Morse code signals instead of relying on user input from physical buttons
- Provide a controlled and predetermined Morse code input for testing, simulation, and demonstration purpose

Morse Code Decoder
- Analyzes and decodes the received Morse code signals into corresponding letters or characters for display on the VGA monitor.

Mux Seven Seg
- Multiplexes and selects the appropriate seven-segment display segments to represent the decoded ASCII letters

VGA

Bitmap ROM:
- Rom file that stores the 64-bit representation of each character that will be displayed on the screen. For a given ASCII character, it outputs a 64-bit std_logic_vector.

Display Logic (a.k.a. vga_sub_shell):
- Contains vga_logic component that generates VGA timing signals such as hsync, vsync, video_on, and pixel coordinates
- Contains display_logic component that determines how the 64-bit number from the bitmap ROM is being displayed on the VGA screen by using pixel coordinates to output a color

VGA_Top_Shell:
- File that connects the display logic to ROM.
- Has a random counter that is used to determine the next letter to be displayed on the screen
- Updates status variables to show whether the letter being displayed is a question, or whether the user's answer is correct or incorrect (by comparing ASCII value sent from the Morse code decoder and the ASCII value of the letter being displayed)

2. Technical Description

2.1) Clock Divider

A clock divider is a digital component that divides the frequency of an input clock signal by a specified factor, producing a lower-frequency output signal. Our clock divider obtains an external 100MHz clock signal and outputs a 25MHz clock signal used throughout our design.

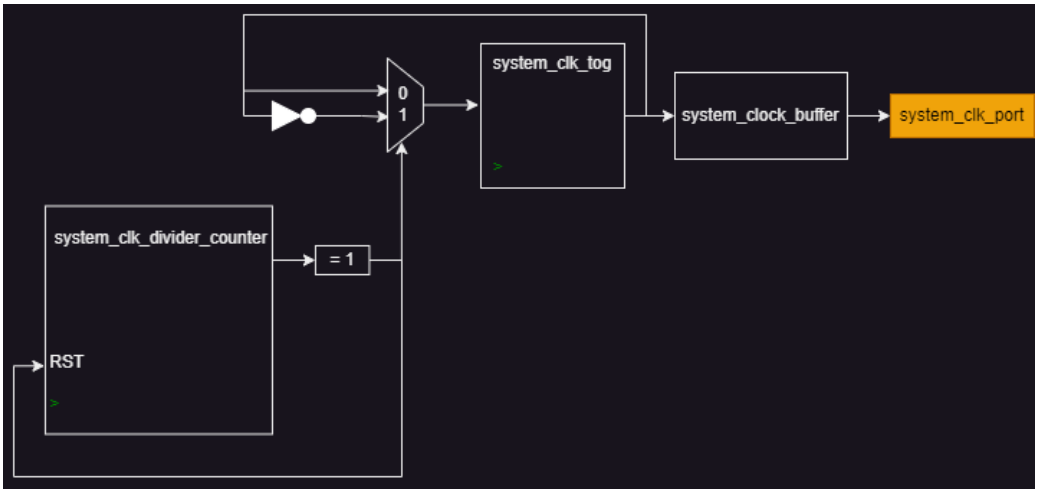
2.1.1) Description of Ports

- Inputs
  - input_clk_port (Default 100Mhz clock signal)
- Outputs
  - system_clk_port (Desired 25Mhz clock signal)
  - button_mp_port (open/not used)

2.1.2) Register Transfer Level (RTL) Diagram

Below is a simple RTL diagram of a clock divider. In our case, we wanted a 25MHz system clk. This meant our system_clk_divider_counter would only count up to 1, before being reset. Simultaneously, the system_clk_tog is alternated between high and low to represent the slower clock. The system_clock_buffer simply takes this alternating signal and puts the new system clock onto the FPGA clocking network. Note that clock forwarding was not used and not represented.

Figure 2: Clock Divider RTL Diagram



2.2) Button Input Conditioning

A button interface component with debouncing and monopulsing functionality. This component ensures reliable and stable input from a mechanical button. It employs a debouncing algorithm that eliminates or reduces the effects of rapid fluctuations or bouncing of the button contacts when pressed or released. This prevents multiple triggering of the button and ensures a clean, stable digital signal output, making it suitable for our Morse code application. The monopulse functionality was not used in this design.

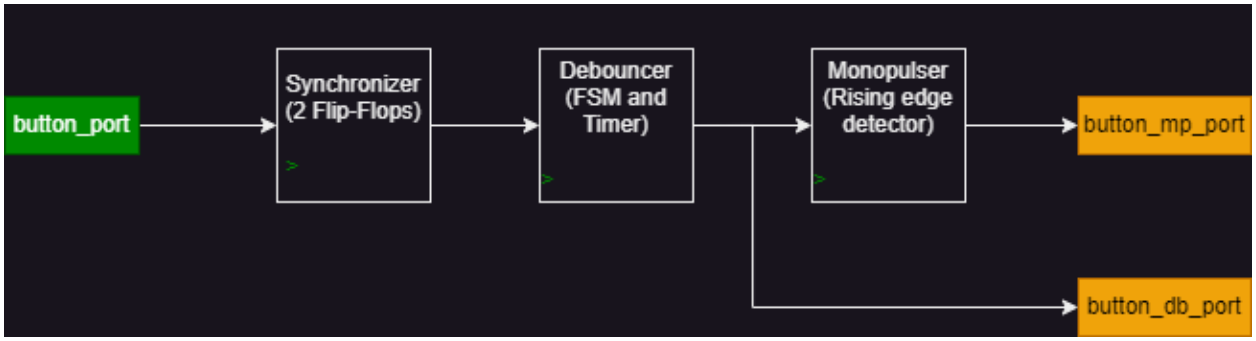
2.2.1) Description of Ports

- Inputs
  - button_port (mechanical input for Morse code and reset button)
  - clk_port (system clk of 25MHz)
- Outputs
  - button_dp_port (debounced button output)
  - button_mp_port (open/not used)

2.2.2) Register Transfer Level (RTL) Diagram

Below is the simplified DataPath diagram for the button interface. Each sub-component performs a special operation to ensure a reliable button press from the user. The synchronizer works to prevent metastability on the input by passing the raw button press through a double flop synchronizer. The bouncer works to remove unwanted input noise from the user inputs button. The monopulser (which was not used in the design) serves to output a debounced signal high for only one clock cycle.

Figure 3: Button Interface Block Diagram & RTL Explanation



2.3) Sound Generator

The sound generator component utilizes a 50Hz pulse width modulation (PWM) signal to produce a soft sound whenever the Morse code button is pressed. By connecting the output of the PWM signal to a speaker, the generated pulses can be converted into an audio waveform, providing useful audio feedback to users. This enables an enhanced user experience by audibly confirming each button press and aiding in Morse code interpretation.

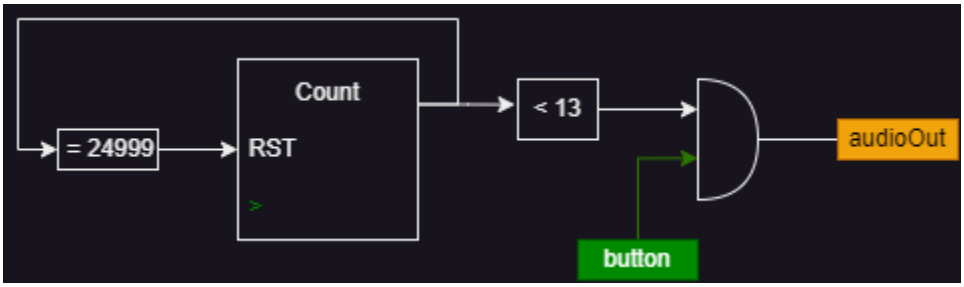
2.3.1) Description of Ports

- Inputs
  - button (port input for the designated Morse code input)
  - clk (system clk of 25MHz)
- Outputs
  - audioOut (output to the audio amplifier and speaker system)

2.3.2) Register Transfer Level (RTL) Diagram

The design is a very simple PWM datapath that sets the frequency to 50Hz by counting to 25000 and setting the duty cycle to (12/25000). The output is only sent to the speaker if the button is pressed as seen by the AND gate.

Figure 4: Sound Generator RTL Diagram



2.4) Machine Morse Code Signal Generator

This component repeatedly generates a Morse code signal of ‘AB’ followed by a space. This is a very simple implementation and only exists for testing purposes. During early hardware validation, it was difficult to know how precise our Morse code presses needed to be so the Machine generator aided in diagnosing suspicions during the debugging process.

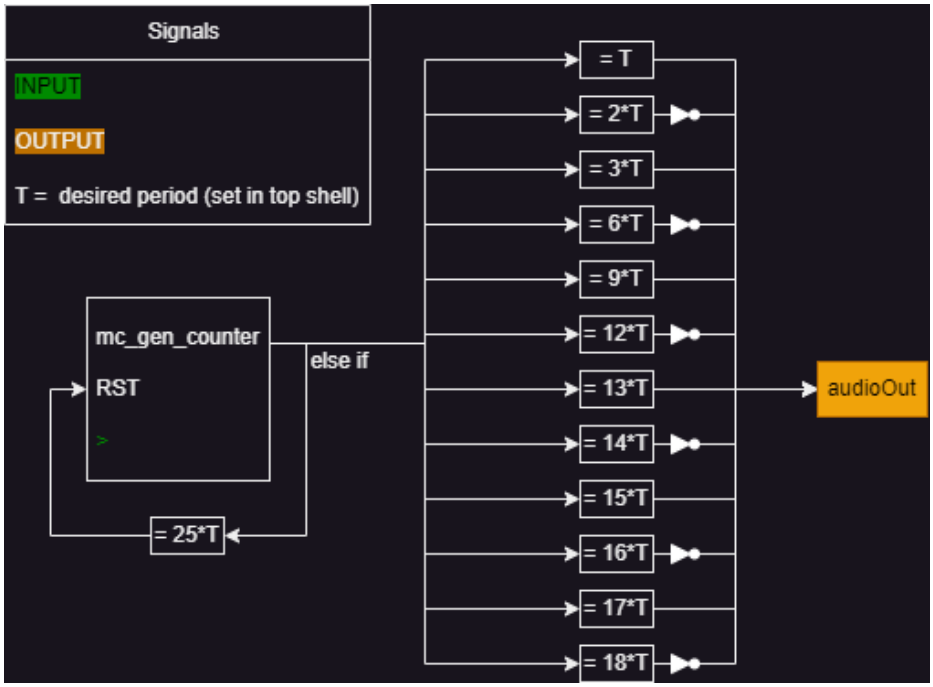
2.4.1) Description of Ports

- Generic
  - T (Morse code time period defined in top shell)
- Inputs
  - clk (system clk of 25MHz)
- Outputs
  - audioOut (output to the audio amplifier and speaker system)

2.4.2) Register Transfer Level (RTL) Diagram

Looking at the figure below, we can see the implementation of the signal generator. It’s produced by a counter that counts and compares to every sequence in its count. Once it reaches a certain threshold, it will change the audioOut output to reflect the relevant change. Once it reaches the end of ‘AB’ and a space, it will reset the counter and begin a new cycle.

Figure 5: Machine Morse Code Signal Generator RTL Diagram



2.5) Morse Code Decoder

This component comprises the majority of the solution for decoding Morse code signals, ensuring smooth and reliable operation. It efficiently interprets inputs from the designated Morse code input and accurately outputs letters based on the universal ASCII binary table. This component is comprised of a FSM and a Datapath to organize components easier in the top shell. It also handles the logic for outputting letters to the seven seg for separate operations from the VGA display.

2.5.1) Description of Ports

- Generic
  - T (Morse code time period defined in top shell)
- Inputs
  - clk_port (25MHz system clk)
  - game_en (enable switch for the decoder)
  - mc_button (designated Morse code input)
  - memory_ascii [8-bit] (ASCII binary for letter obtained from memory)
  - mux7seg_switch (enable switch for seven seg)
- Outputs
  - ascii_letter [8-bit] (8-bit ASCII binary for letter output)
  - char_reg_ext [10-bit] (Morse character for indexing memory)
  - mux7seg_out_one [4-bit] (4th digit of seven seg output)
  - mux7seg_out_tenth [4-bit] (3rd digit of seven seg output)
  - new_data (signal is high when a new letter available)

2.5.2) Register Transfer Level (RTL) Diagram

The datapath is the most complex component for the Morse Decoder as there are many different simultaneous processes. It helps to break it down into different sub-components:

Period Definitions (T)

There are rules to help people distinguish dots from dashes in Morse code. We implemented one definition in the top shell accessible by necessary components. For reference, the length of a dot is 1-time unit. A dash is 3-time units. The space between symbols (dots and dashes) of the same letter is 1-time unit. The space between letters is 3-time units. The space between words is 7-time units. In our implementation, we used a T of 0.25s for the best user handling. Please note that T has expressed in terms of clock cycles to simply the datapath.

High and Low Counters

Knowing how Morse code works now, it makes sense to have two counters in the datapath. One for low signals and one for high signals (button presses). Now because T is expressed in terms of clk cycles, both counters count clk cycles and operate inversely of each other. The high_en signal is high only when the Morse code button is pressed and start_en is true. Start_en is high only when the FSM is in the correct state to operate the Datapath. The low_en signal is high only when the Morse code button is not pressed, start_en is high, and first_press is high. First_press is determined by the first ever button press (to prevent premature

signals from the low_counter). Also, note that the prev_high register obtains the last value from the high_counter when low_en allows character detection.

Character Comparison and Register

Once the pre_high_reg obtains the value it instantly compares against CharT (1.5*T) to determine if the value is closer to a dot or dash. From there, it serves as an enable weather to dispatch a ‘01’ or ‘10’. Nothing happens to the char_reg until an enable has been received by the FSM, which will allow the two bits to be concatenated on the right of the current char_reg (shifted left two). The char_reg also has a reset_char signal to help prevent mix-ups when storing previous values. We initially used a 5-bit char_reg but determined we needed a 10-bit register to be able to store default values aside from representations of a dot and dash.

Memory Comparison and Letter Register

After memory comparison with our ROM memory the Decoder receives an 8-bit input in the form of an ASCII binary representation of a letter. From there, the FSM issues a compare_en signal to move this new memory_ascii into the letter_reg. The next clock cycle, ascii_letter is outputted on the data line. If the FSM determines it’s time to send a space (between words), then the load_space_en will become high. When this happens, a value of ‘00100000’ will be loaded automatically into the letter_reg. This was the easiest way to perform this function of loading a space because it could not be held in char_reg and compared in memory. One important consideration is that ROM memory takes two clock cycles to process but we don’t have to change anything because we know that compare_en will not become high until thousands of clock cycles after char_shift_en is high. This is described in period definitions of T and the FSM.

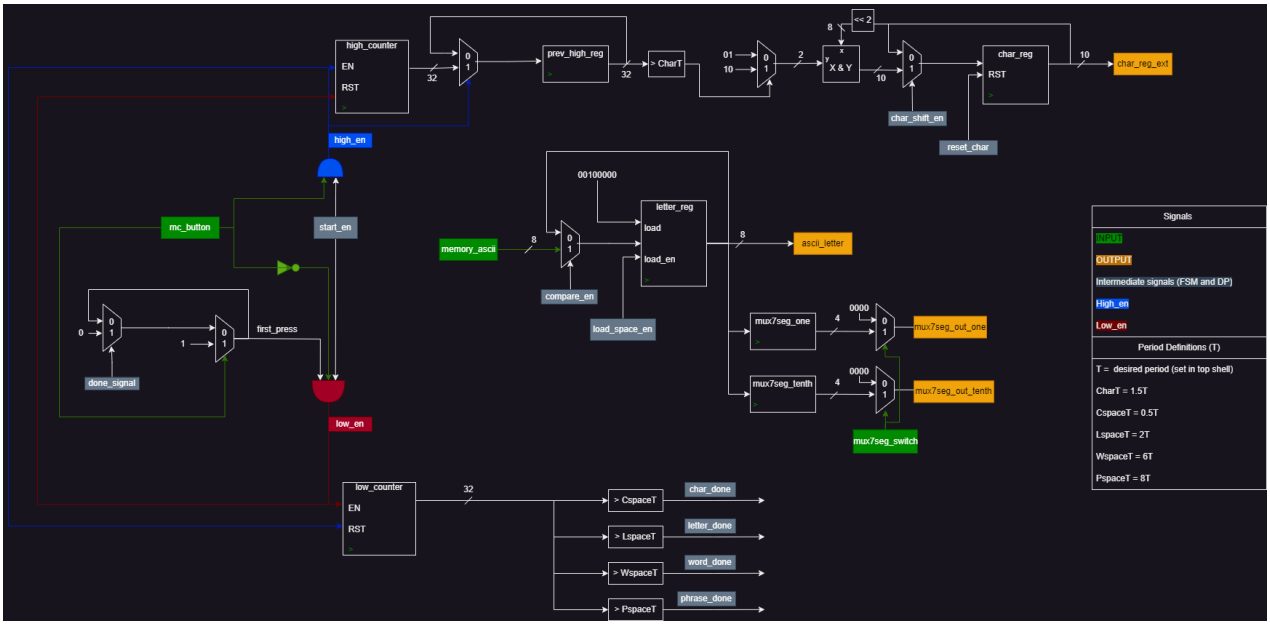
Low Space Signal Logic

This is visualized on the bottom of the RTL diagram and serves to give the FSM certain acknowledgments of the status of spacing. The button presses can be handled with the datapath while spacing logic should be handled by the FSM as it dictates certain operations in the datapath itself. The signals sent are when the low counter has reached a certain value as determined by the values on the right side table.

Mux Seven Seg Logic

Lastly, we have the seven seg logic which incorporates the switch (SW13) to disable or enable the display in the Decoder component. This allows for easier operation and control of the display. The values for the display are pulled directly from the letter_reg.

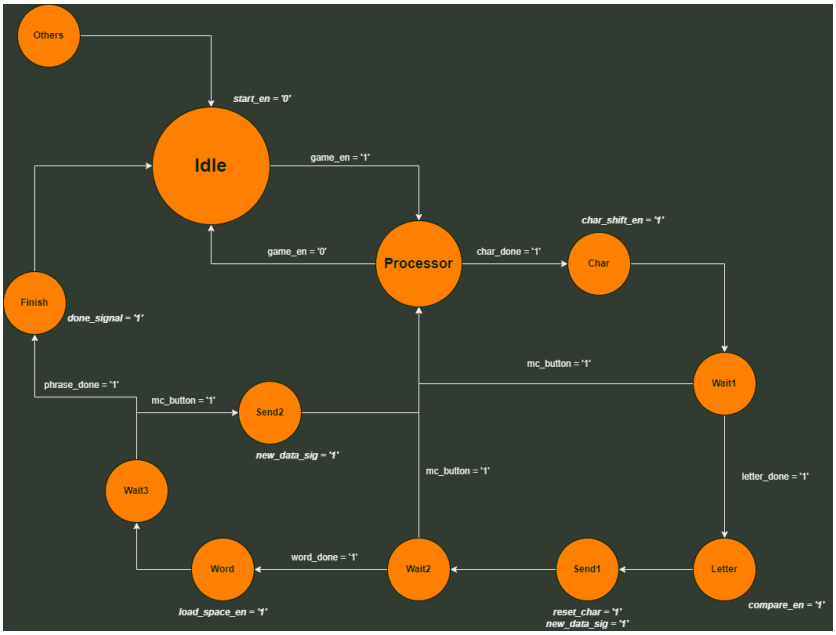
Figure 5: Morse Code Decoder RTL Diagram



2.5.3) Finite State Machine (FSM) Diagram

The Finite State Machine serves to handle all logic regarding the spacing of Morse code characters including characters, letters, words, and the end of a phrase. The FSM receives signals from the datapath when the low signal has reached a certain length based on the time period set in the shell. The FSM begins by moving to the Processor state if game_en is high, and will only remain there if game_en is high. From there, using datapath signals, it will progress through the sequence of combing morse code characters into one 10-bit register in the datapath shifted in by char_shift_en in the Char state. Once five or fewer characters have been entered, the FSM will then set compare_en high to parallel shift the memory block ASCII binary value into the letter_reg in the datapath. From there, the next state is Send1 with no restriction and new_data will go high, and the char_reg will be reset. Now depending on the morse code input, the FSM will either go back to Processor or continue down the sequence to either produce a space output or signal the phrase is done. Using another datapath signal, we check if word_done = '1', and then the FSM sets load_space_en high to shift a binary ASCII representation of a space into the letter_reg. From there, new data will only be set high if the Morse code button is pressed again, or else it will go to the Finish state if the phrase_done signal is high first. In the finished state, done_signal will go high, and the next state is Idle again.

Figure 6: Morse Code Decoder Finite State Machine



2.5.4) Description of Memory

A single-port ROM is utilized to map Morse code inputs (10-bit) to ASCII characters (8-bits). The ROM memory would store a look-up table that associates each Morse code sequence with its corresponding ASCII character. The ROM has a total of 1024 memory locations to accommodate all possible Morse code combinations. A dot is represented by 01, and a dash is represented by 10. For example, a specific memory location stores the Morse code sequence "-.-." or "0010011001". When the input sequence is presented to the ROM's address inputs, the ROM would output the corresponding ASCII character "C" at the data output.

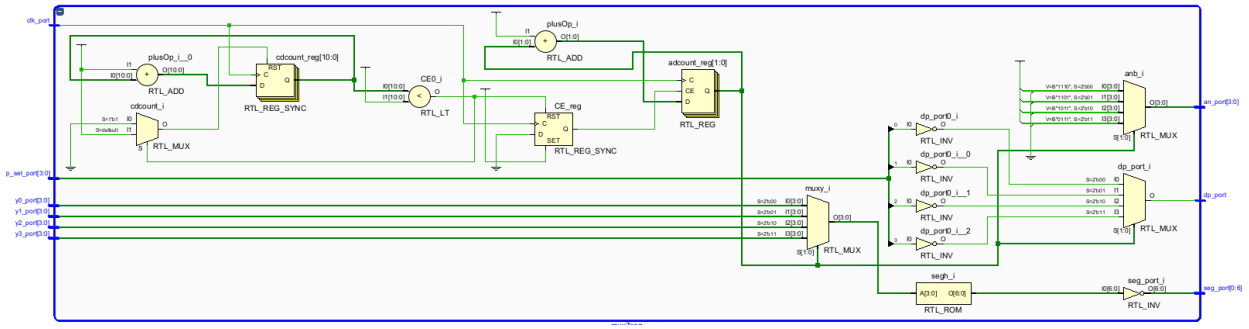
2.6) Mux Seven Seg

2.6.1) Description of Ports

- Inputs
  - clk (system clk of 25MHz)
  - y3_port (desired representation on left most display digit)
  - y2_port (desired representation on middle left display digit)
  - y1_port (desired representation on middle right display digit)
  - y0_port (desired representation on right most display digit)
  - dp_set_port (desired decimal representation on all digits, will be 0000)
  -
- Outputs
  - seg_port (segments (a...g) of each digit)
  - dp_port (decimal points, will always be a constant 1111 for us)
  - an_port (anodes to cycle digits)

2.6.2) Register Transfer Level (RTL) Diagram

Figure 7: Mux Seven Seg RTL Diagram



2.7) VGA Top Shell

The VGA top shell connects internal display components and updates status variables to determine the color of the letter on the screen with small glue logic components.

2.7.1) Description of Ports

- Inputs
  - Data_in (An input port that gets an 8-bit ASCII value of the displayed letter from the Morse code decoder. This value is used to compare whether the user entered the correct Morse code or not of the displayed letter.)
  - Fast_clock (100 MHz clock signal.)
  - Slow_clock (25 MHz clock signal)
  - New_data : (signal is high when data_in port is ready to be read)
  - Reset (input signal that changes the letter displayed.)
- Outputs
  - Hsync_sig_ext (carry the hsync signals to the VGA)
  - Vsync_sig_ext (carry the vsync signals to the VGA)
  - vgaBlue_ext (4-bit component output of blue color to VGA)
  - vgAGreen_ext (4-bit component output of green color to VGA)
  - vgaRed_ext (4 bit component output of red color to VGA)

2.7.2) Register Transfer Level (RTL) Diagram

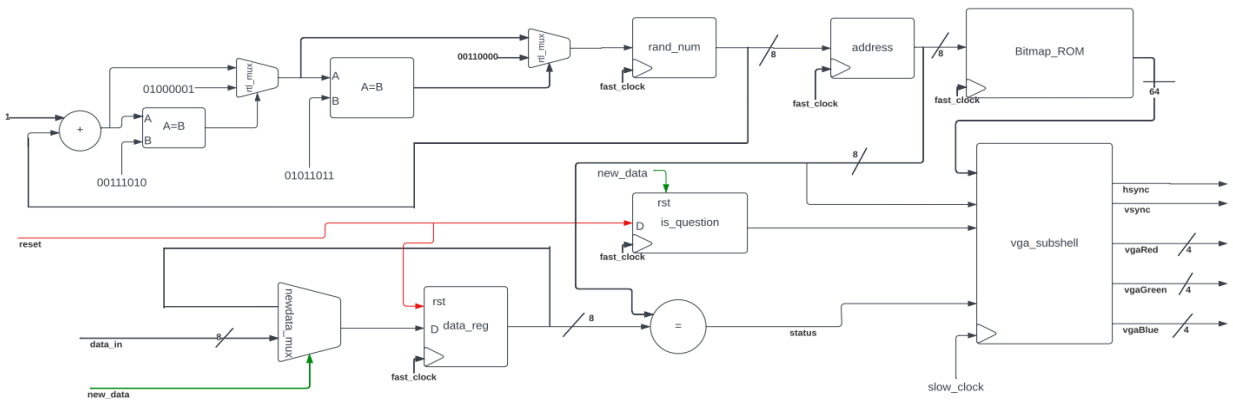
Address and data_reg are 8 bit registers containing the ASCII values of the question(letter being displayed) and the ASCII equivalent of the user's input respectively. Using these two registers, the color of the letter on the screen is updated to show whether it is a question, correct answer, or incorrect answer through variables is_question and status.

Random_num counter cycles through all possible ASCII values of letters and numbers. When the counter reaches the ASCII value of 58, the counter jumps to the ASCII value of 65. Likewise, when the counter reaches an ASCII value of 91, it jumps to an ASCII value of 48. This allows the counter to cycle continuously through the ASCII values of the letters and numbers without any gaps in between.

The value of random_num, when reset is high, is stored in the Address register. This ASCII address determines the character displayed on the screen.

Is_question flip flop determines whether the character being displayed on the screen is a question or not when its output is high and low respectively.

Figure 8: VGA Top Shell RTL Diagram



2.7.3) Description of Memory

The bitmap_ROM contains all representations of all the letters and numbers and outputs the single-character representation based on the input address. We got this file from Git Hub. The output of the ROM file is a 64-bit std_logic_vector that forms an 8x8 binary representation of all the characters. Bits 0 to 7 would form the bottommost row, 8 to 15 would form the row above, and so on. The rightmost column in this representation is only 0 to allow multiple characters to be displayed with appropriate spacing in between. This spacing between characters would have been useful for us when we display words or phrases on the screen(in the future ofc).

2.8) Display Logic/VGA SubShell

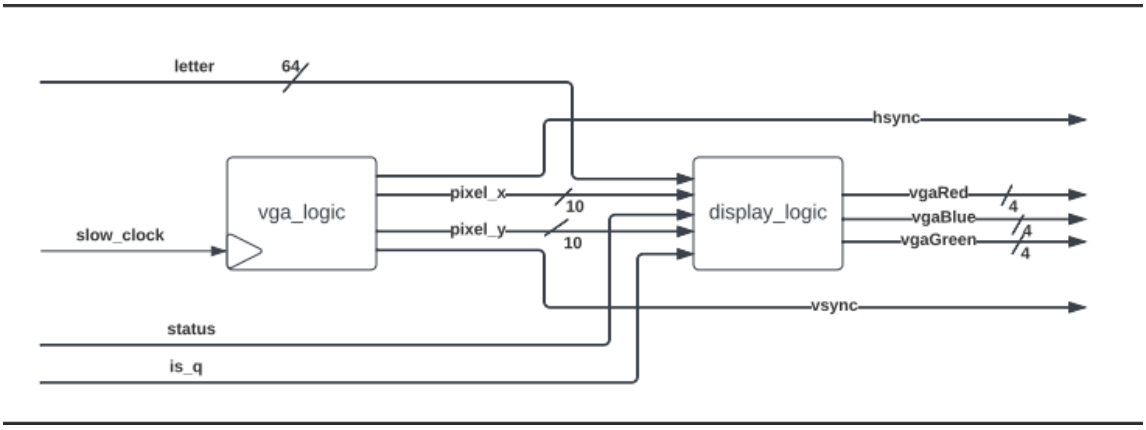
2.8.1) Description of Ports

- Inputs
  - Slow_clock (A single-bit input that is a 25 MHz clock signal for creating VGA signals)
  - Letter_sig (A 64-bit input std_logic_vector that represents each letter and number in an 8x8 binary grid)
  - Is_q (A single bit input that when high makes the letter being displayed on the screen white in color to show that the user has not yet entered the appropriate morse input)
  - Status (A single bit input that is when the user enters the correct morse code and low when the input is incorrect. The letter on the screen becomes green and red respectively)
- Outputs
  - Hsync_sig_ext (carry the hsync signals to the VGA)
  - Vsync_sig_ext (carry the vsync signals to the VGA)
  - vgaBlue_ext (4-bit component output of blue color to VGA)
  - vgAGreen_ext (4-bit component output of green color to VGA)
  - vgaRed_ext (4 bit component output of red color to VGA)

2.8.2) Register Transfer Level (RTL) Diagram

The VGA subshell connects the two components vga_logic and display_logic. Vga_logic uses two counters(x and y pixel locations) to generate hsync and vsync signals. Letter_sig tells display_logic whether or not the pixels in the center of the screen should be on or not, while status and is_q determine the color of the pixel when on. These two variables, along with the x and y location, can be used to make pixel (x,y) appear a particular color. As a result, a letter/number is seen on the screen in white/red/green. Note: Like the VGA test pattern generator, each bit in letter_sig is hard coded to a particular pixel coordinate to easily determine whether it outputs a color or not. This multilevel case scenario is all the logic inside display_logic.

Figure 9: Display Logic RTL Diagram



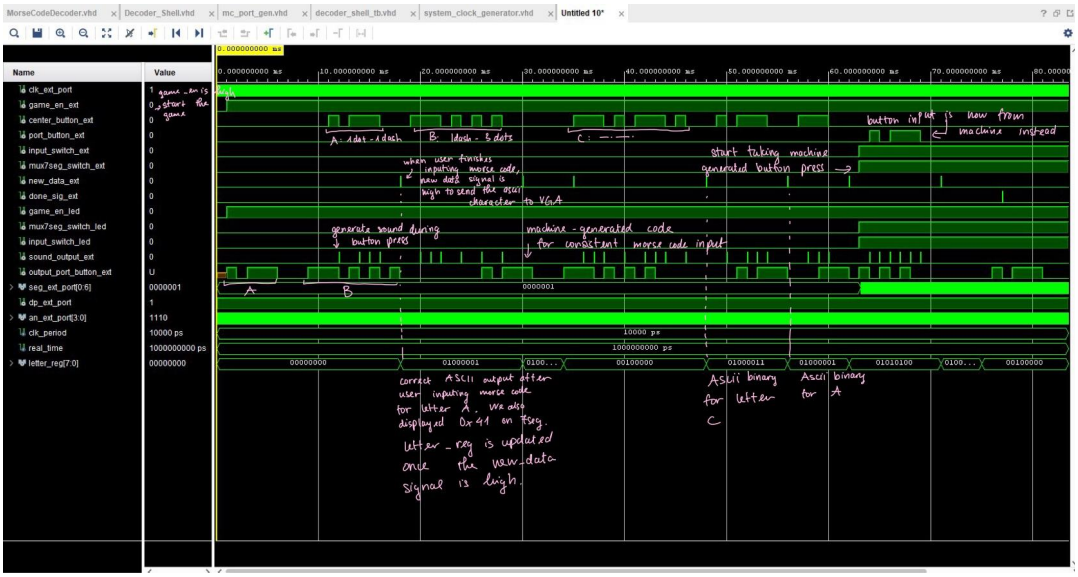
3. Design Validation

We felt it was not necessary to test the clock divider, button interface, and mux seven seg because it was previously analyzed and tested during labs and proven to work. Therefore, if this component is incorporated correctly, there should be no issue in the top shell test bench. We organized the software and hardware validation into three separate initiatives: Morse Code, VGA, and Combined.

3.1) Morse Code

Testbench

Figure 10: Morse Code Decoder Top Shell Testbench



Hardware

- SevenSeg/Translator:
    - Involves verifying the correct operation and functionality of the translation of Morse code to ASCII characters. This validation ensures that the displayed ASCII letters on the seven-segment display accurately correspond to the Morse code input. The 2nd digits of SevenSeg display the first 4 bits of the ASCII binary and the 3rd digits of SevenSeg display the last 4 bits of the ASCII binary.
    - For simplicity of the program, the seven-seg displays two separate four bits of an ASCII binary representation. Therefore we need our own translation to understand it well. Please see Appendix C for the translation.
- Machine Code Gen
    - Involves verifying decoder functionality by simulating user input buttons with machine-generated input and observing the decoder's actual decoding of the correct Morse code.

3.2) VGA

3.2.1) Display Logic/VGA sub-shell

For testing the VGA component, we began by testing the test pattern provided on Canvas. This did not require any specific stimulus from the user so it was straightforward. We used the slides presented in class to get an understanding of how to build the display logic. Using the test pattern file provided, here is the test pattern we saw on our VGA.

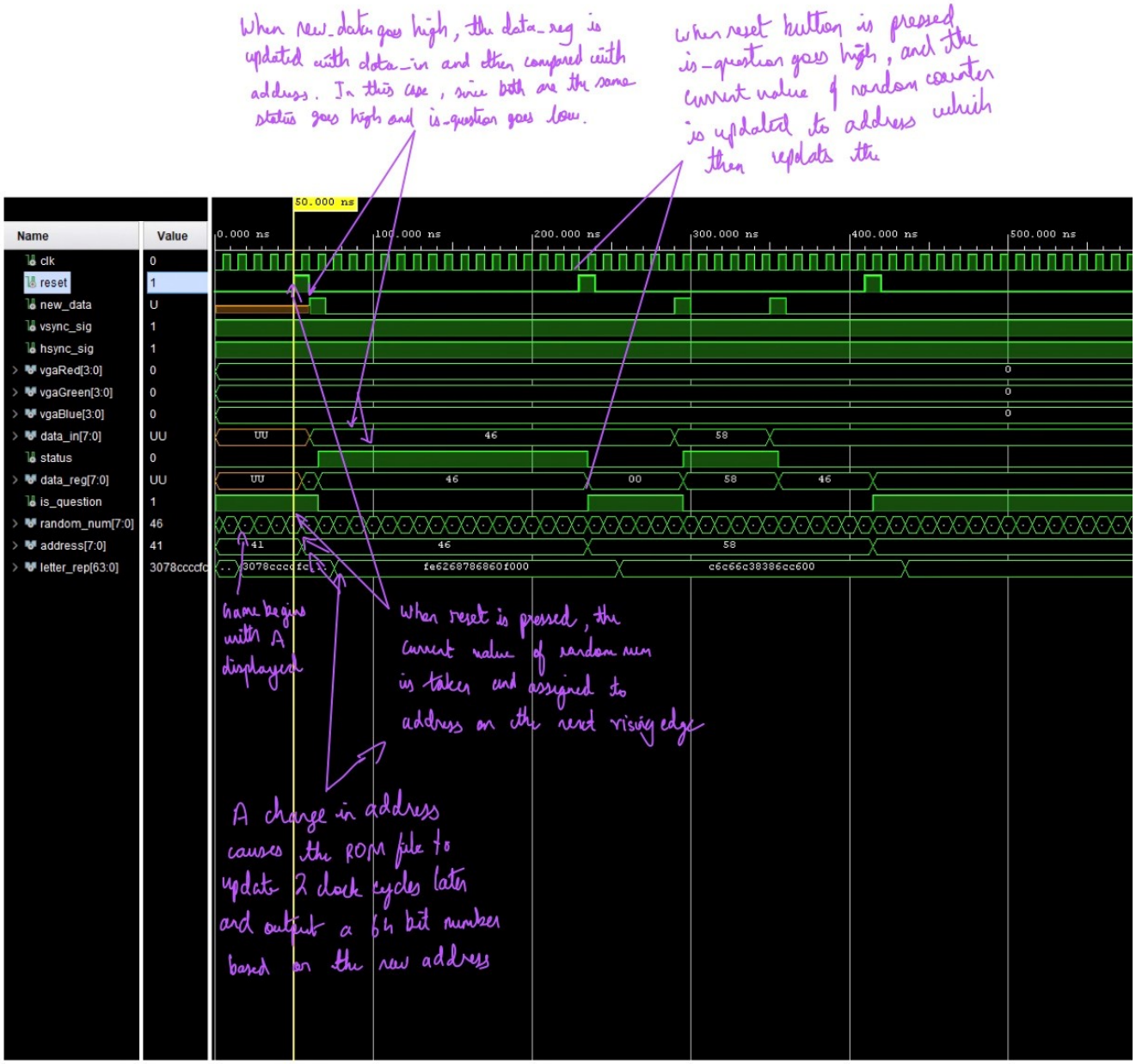
Figure 11: Testing Pattern

To display letters on the screen, we got a letter bitmap from the bitmap ROM, which hardcoded the center pixels of the screen to a particular bit of the letter bitmap. Status and is_q signals determined the colors while the letter bitmap determined which questions were on to display the character in an 8x8 grid.

3.2.2) VGA Top Shell

For our next stage of testing, we directly jumped to testing the top-level shell of our program because the components of our program were minimal and strongly tied to each other. The tests were simple, we sent data into the FPGA three times and observed how the VGA responded when the data sent matched what was displayed on the screen and how it did when they did not match. We also tested the reset button to see if a random letter was picked to be displayed when it was pressed. Below is the testbench waveform simulation for the VGA top level:

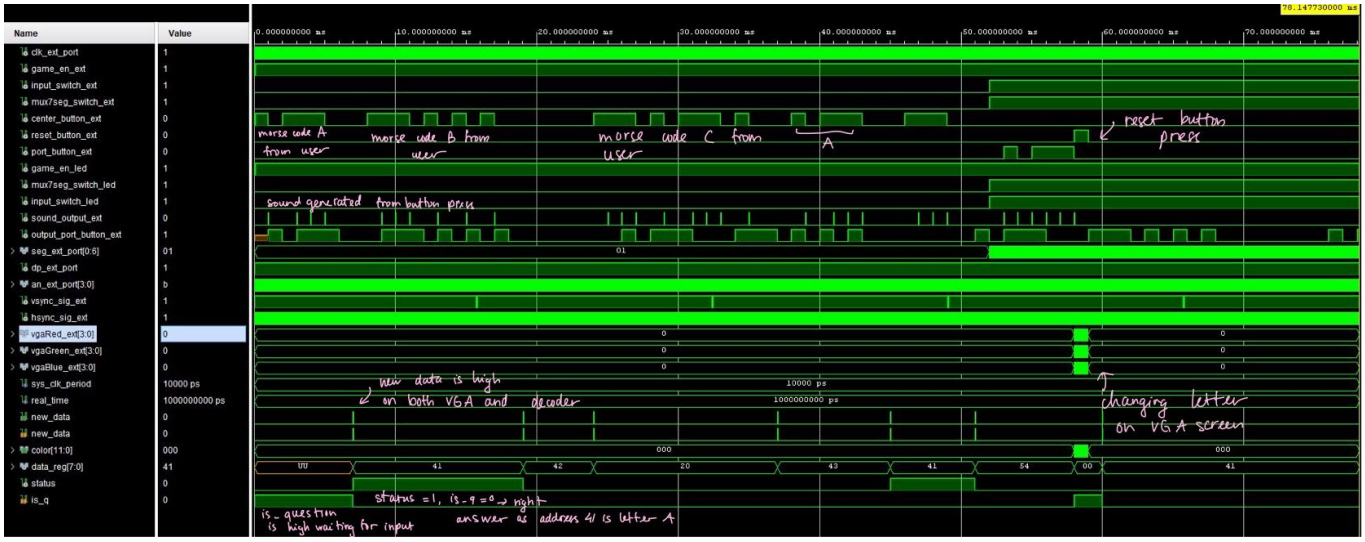
Figure 12: VGA Top Shell Testbench



3.3) Overall Design

3.3.1) Behavioral Simulations

Figure 13: Overall Testbench Simulation



3.3.2) Hardware Validation

Morse Code:

To validate the Morse code functionality, we used a combination of the 7-segment display and machine-generated code.

7-Segment Display Validation:

Generate Machine-Generated Morse Code: Use a signal generator to generate machine-generated Morse code signals representing different characters.

Connect the Machine-Generated Morse Code to the Morse Code Decoder: Connect the output of the signal generator to the input of the Morse code decoder module. Ensure that the decoder accurately interprets the machine-generated Morse code and produces the corresponding ASCII characters.

Display Morse Code on the 7-Segment Display: Connect the output of the Morse code decoder to the input of the 7-segment display module. Confirm that the module correctly displays the decoded Morse code on the 7-segment display, allowing visual verification of the Morse code translation.

Validate Accuracy of the Displayed Morse Code: Cross-reference the displayed Morse code on the 7-segment display with the expected Morse code patterns. Verify that the displayed Morse code matches the machine-generated Morse code, ensuring accurate translation.

Link to the video of Morse Code Game on VGA:

4. Analysis of the Design

4.1) Resource Utilization

What resources on the FPGA did your design utilize? How does this relate to the total capacity of the FPGA.

Ports:

- Two buttons were used on the FPGA to
  - Input the Morse code
  - reset the game.
- Three switches were used on the FPGA to
  - Enable the game
  - Enable the Seven Seg
  - Switch to machine code

Memory:

- For the VGA component, 2 ROM files were stored in memory. One ROM was 16384 bits large and the other ROM file was 2048 bits large.
- For the Morse Code component, 1 ROM file of size 8192 bits was stored in memory.

MUX Seven Seg:

- The FPGA's MUX Seven Seg was used to display Morse Code input from the user.

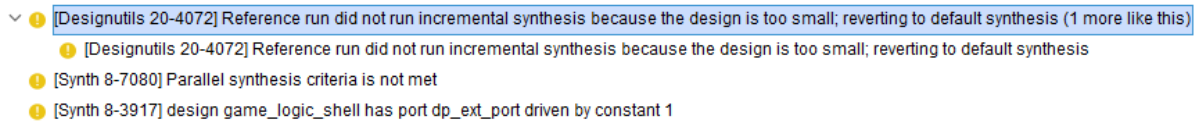
Audio Port → JA7

- JX1 & JX2 can be linked together for the Machine Morse Code Signal Generator.

4.2) Residual Warnings

Include a list of all residual warnings and a short description of why you can ignore it. Note: you should resolve all warnings about unintended latches!

Figure 14: Error 1



Above are the residual warnings we had when we ran our code:

Warning 1: Reference run did not run incremental synthesis because the design is too small:

In Vivado, incremental synthesis is a feature that allows for faster synthesis by reusing previously generated results from a reference run. However, if the design is determined to be small enough, the incremental synthesis may not provide a significant improvement in

performance compared to the default synthesis process. Therefore, Vivado reverts to using the default synthesis instead.
Essentially, this error message is informative and does not necessarily indicate a problem or an issue with your design. It's simply stating that the design is small enough that the incremental synthesis feature is not necessary.

Warning 2: Parallel synthesis criteria is not met

The specific criteria for parallel synthesis may vary depending on the EDA tool or environment being used. It could involve factors such as the size of the design, available system resources (such as CPU cores), or specific settings and configurations.
When the parallel synthesis criteria are not met, it means that the conditions necessary to enable parallel synthesis are not satisfied. As a result, the synthesis tool falls back to using a default or non-parallel synthesis mode, which might be slower.

Warning 3: design game_logic_shell has port dp_ext_port driven by constant 1

This warning suggests that the decimal point signal for the seven seg display is constantly on. Given that the Seven Seg display does not affect the functionality of the game and also considering that the decimal point itself does not affect the number on the display, this warning is ignorable.

4.3) Division of Labor

Use this space to describe the division of labor between group members. How did your group divide up the project to make it more manageable? Was the strategy effective?

The primary division of labor within our group was to split into pairs so that two of us could work on the VGA component while the other pair could work on the Morse Code component. This was an obvious choice of division but regardless it proved effective as both components were demanding their own rights and therefore required an equal amount of effort. Within the two components themselves, there was no specific division of labor as we always tried to work on wherever our teammates left off rather than work on separate components. This worked well for this project because the compactness and smaller scale of the project meant that changes in one part of the project could have a significant impact on the other easily leading to issues if there is a lack of communication.

4.4) Future Work

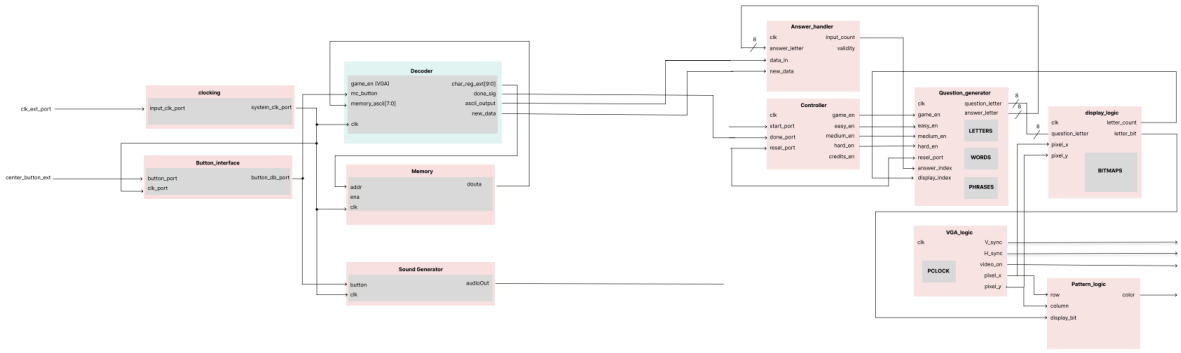
What would be the next steps if you wanted to expand or improve the design?

A few improvements that we were working on and hoped to incorporate into the project include:

- Redesign the display logic so that we could dynamically display any letter on the screen rather than hard code them giving us the ability to display words and phrases as well.
- Design a menu screen from which the difficulty can be chosen, this decides if there are letters, words, or phrases on the screen.

- Add 64-bit number representation to the coefficient file to give the option to display numbers as well. This would be especially useful to us to display the score of the person playing the game.

Figure 15: Future Work Diagram



Above is a block diagram of our future plan for the project if we had been given more time. The game logic would be much more fleshed out.

We also had a VGA design for the menu of the game:

Figure 16: Demorse Proposition

5. Acknowledgments

We would like to acknowledge Professor Luke, Tad, Kendall, and Katherine for being incredibly helpful in the design process and helping point us in the right direction while we worked on scoping our project down.

Sources used in this project:

- ★ Bitmap ROM: https://gist.github.com/rothwerx/700f275d078b3483509f

6. Conclusions

The above project walks one through on how to build a game that integrates Morse code inputs with a VGA output. The workload for this project is definitely intense, so we would recommend two two-person teams to work on this. We had one team working on the VGA side and one working on the Morse code side.

Given the choice to do this project again, we would try completing our initial goal of being able to play the game with words and phrases instead of just letters and numbers. Although this project was intellectually challenging and time-consuming, it was very rewarding.