TE3060 Embedded Systems Laboratory

Project Title:
Morse Code receiver and transmitter using FPGA/HPS

Final Project

Vanya Michelle Medina Garcia - A01228016
Luis Alfredo Aceves Astengo - A01229441
Eduardo García Olmos - A01351095

November 26th, 2020

Revision History

Rev. #	Date	Changes By	Description
0.1	17/10/20	Team	First general draft of proposal (whole document)
0.2	18/10/20	Team	Updated figure enumeration, added table of figures.
0.3	18/10/20	Team	Added reference documents.
0.4	19/10/20	Team	Last revision of document content before sending to revision.
0.5	20/11/20	Team	Review of proposal.
0.6	21/11/20	Team	Updated document structure for final report.
0.7	22/11/20	Team	Added relevant information for the understanding of our project.
0.8	23/11/20	Team	Updating block diagrams and images.
0.9	24/11/20	Team	Added relevant information regarding the implementation and references.
1.0	25/11/20	Team	Last revision of document content before submitting.

Table of Contents

Introduction **5**
- Scope 5
- Intended Audience 5
- Terminology 6
- Related Documents 6

Project Description **7**
- System Block diagram 10
- Project Goals 11

Theoretical Background **11**

System Requirements **13**

Design Description **13**
- Morse Code Decoder Architecture 13
- Morse Code Encoder Architecture 16

Test and Results **20**

Verification **22**

Conclusions **23**

References **23**

Table of Figures

Figure 2.0.0 Reception connection (proposal) 7
Figure 2.0.1 Reception connection (final) 8
Figure 2.0.2 Transmission connection (proposal) 8
Figure 2.0.3 Transmission connection (final) 9
Figure 2.1.0 System's high-level block diagram (proposal) 10
Figure 2.1.0 System's high-level block diagram (final) 10
Figure 3.0.0 International Morse Code 12
Figure 5.1.0 Logic gate of dot_in 13
Figure 5.1.1 Logic gate of dash_in 14
Figure 5.1.2 Logic gate of done_in 14
Figure 5.1.3 Decoder FSM 15
Figure 5.1.3 Encoder FSM 17
Figure 5.4.0 System Interconnection 20
Figure 6.1.0 Simulation of receiver module 21
Figure 6.1.1 Simulation of transmitter module 21

List of Tables

Table 1 Acronyms 6
Table 4.0.0 System Requirements 13
Table 5.1.0 Register use and description 14
Table 7.0.0 System requirement verification 22

1.0 Introduction

This document describes the final project that we propose to implement as part of the final evaluation of the course TE3060 Embedded Systems Laboratory. The goal of this document is to describe the main details of what we implemented, the objective of this project was to create an embedded system, using a development board with a Cyclone V SoC, we implemented a system whose function is to provide both input and output of a Morse Code communication, simulating the behavior of a transmitter and receiver. The Terasic DE10 Nano board was used. In order to develop our project, we must have a good understanding of how the FPGA and HPS in the SoC can be interconnected to achieve the desired functionality, as well as a theoretical background about the Morse Code. This report shows the path that we took along to complete our project, when implementing the project we differed in some areas, they will all be explained along the document. The fixed requirements achieved are described in the acceptance criteria and more advanced requirements in the project goals.

1.1 Scope

This document provides a high-level description of the system that was implemented, including its main features, as well as the functional blocks available in the SoC that we used, and how they are interconnected. A high-level block diagram of our system architecture is provided. Theoretical context regarding the Morse code communication system and how we implemented it. The results of our testing and functionality of our project. Finally, we will review the acceptance criteria that we previously set for our project to be considered successful.

1.2 Intended Audience

This document is intended for hardware engineers and embedded systems designers or developers who would like to learn about the implementation of Morse Code receiver and transmitter using a development board with Cyclone V SoC. A medium level understanding of microprocessors and FPGA topics is recommended.

1.3 Terminology

Acronym	Description
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HPS	Hard Processor System
LED	Light-emitting diode
SoC	System on Chip

Table 1 Acronyms

1.4 Related Documents

Morse Code Decoder (2013). *Computer Science University of Toronto.* Retrieved 18 October 2020, from http://www.cs.toronto.edu/~milic/csc258/report.pdf

This is a basic guide for building a program on the DE10 Nano SoC. This walks through using Quartus Prime, Qsys, and EDS from start to finish. The program counts up four LEDs on the HPS.

DE10 SoC Nano Basic Guide (2017). *Github.* Retrieved 18 October 2020, from https://gist.github.com/addisonElliott/8c229e47184a724d1a00328110dc094c

DE10 Nano User Manual (n.d.). *Intel.* Retrieved 18 October 2020, from https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-4302081511597-de10-nano-user-manual.pdf

2.0 Project Description

This project uses a Cyclone V SoC from INTEL, in particular, Terasic DE10 Nano development board to implement a system that can detect (receive) and generate (transmit) signals encoded in Morse Code. For this to be considered an embedded system, we used both FPGA and HPS available in the SoC. The software resources used in this project are Quartus Prime Lite and the Intel SoC FPGA Embedded Development Suite tool. For the verification part we used Modelsim. The functionality is achieved as follows:

Reception:

Figure 2.0.0 was the original proposal of how blocks inside the SoC would be connected to achieve the desired functionality. However, after re-evaluating the challenges of using a single button, it was decided to use two buttons and a slider switch. A third button would have been more efficient, but the DE-10 Nano board only has 2 buttons on the FPGA fabric, so a switch watch used to switch button functionality. The block diagram of the final implementation of the system is shown in Figure 2.0.1.

The input Morse Code signal is generated using two push buttons and a switch on the FPGA. The morse code decoder is in charge of detecting and decoding the signal (differentiate between a dot a dash and a done signal). More details of this are explained in section 5.0. Once the signal has been decoded, HPS reads the information (the received char) from the FPGA using the lightweight HPS-FPGA AXI Bridge. Once the HPS has received the data, it will be sent via UART interface to the terminal PUTTY in the computer, where the decoded Morse Code input is displayed as characters.

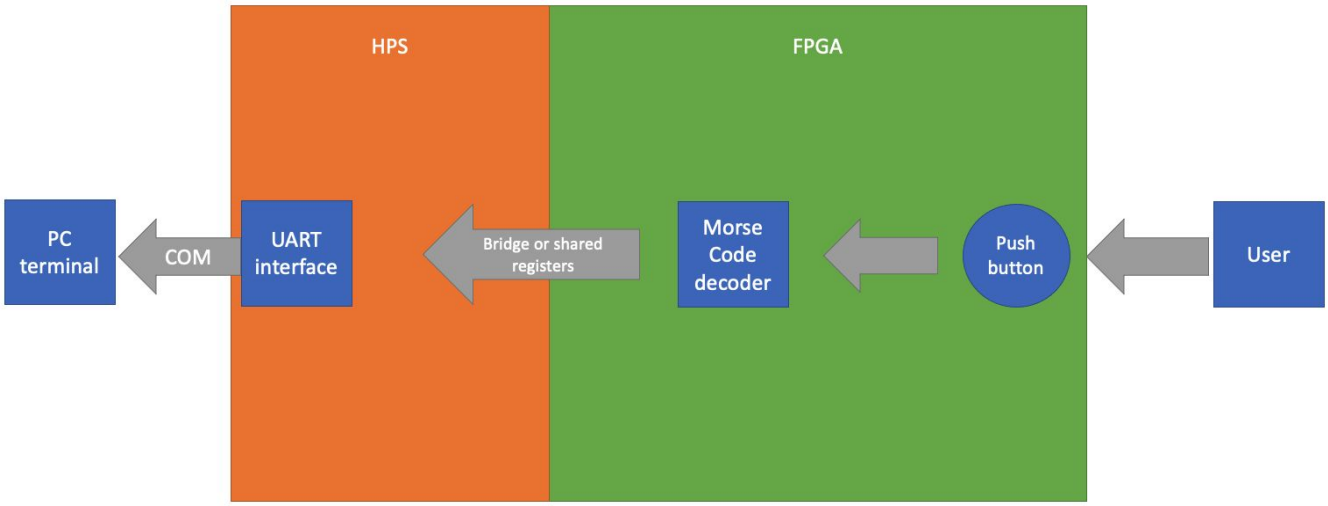

Figure 2.0.0 Original block diagram proposal of system's receptor

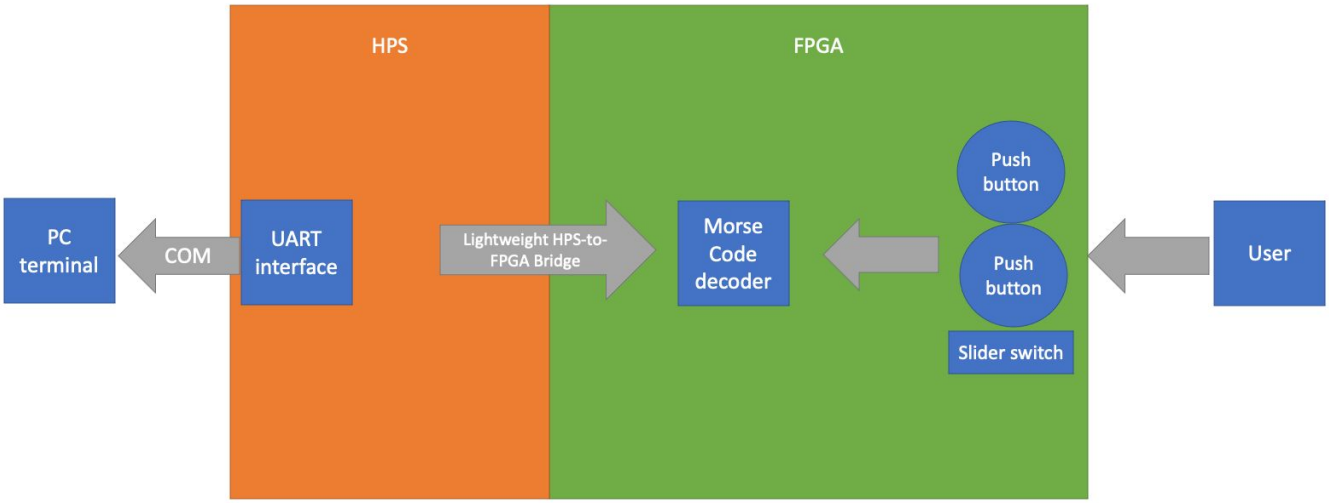

Figure 2.0.1 Block diagram of system's receptor final implementation

*Note: the middle arrow points from HPS to FPGA because this is the direction of this bridge. However, the bridge is used to read data from the decoder and bring it back to the HPS.

Transmission:

The transmission follows a similar process to the reception, but in inverse order. A message will be typed in the terminal and sent via UART to the HPS, which will store the received characters. The return (\r) char will be used as a "start" signal, to indicate the frame of characters is complete and ready to be sent. We decided to implement this behavior , so the characters in the message we want to transmit are sent consecutively, instead of just transmitting each key press as it is received . Once the start is detected, the whole frame will be sent from the HPS to the FPGA using the same lightweight HPS-FPGA AXI Bridge used for the reception. On the FPGA side, we have an encoder that encrypts the message into Morse Code. This same block is in charge of transmitting the Morse signal. The transmission is done by blinking the LEDs on the FPGA. Figure 2.0.2 was the original proposal on how the system would be connected. It only suffered one modification: the buffer was implemented inside the Morse Code encoder block on the FPGA side. This was done to avoid having to poll when the encoder has sent each character, instead we write all the characters inside a buffer on the encoder, and it takes care of sending them all. Figure 2.0.3 reflects this change.

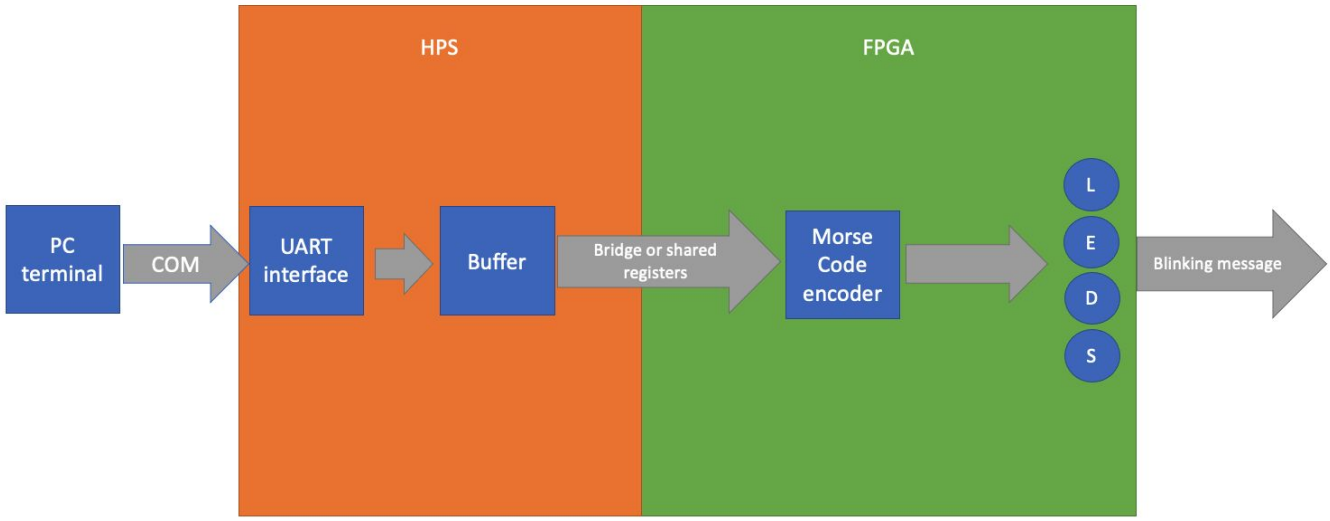

Figure 2.0.2 Original block diagram of system's transmitter

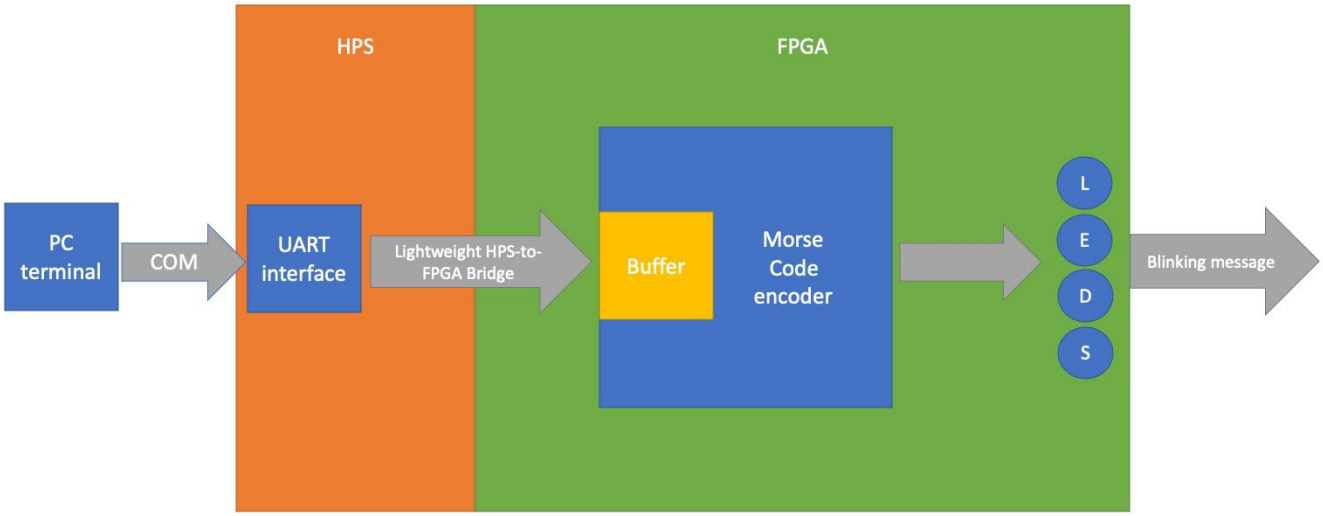

Figure 2.0.3. Block diagram of system's transmitter final implementation

*Note: all the LEDs will blink simultaneously to give a stronger visual representation of the message being sent (think of it as a ship sending an SOS signal in the middle of the ocean).

2.1 System Block diagram

Summing up the partial functionalities described in Section 2.0, Figure 2.1.0 shows the system's original proposal high-level block diagram I, and Figure 2.1.1 shows the system's final implementation block diagram.

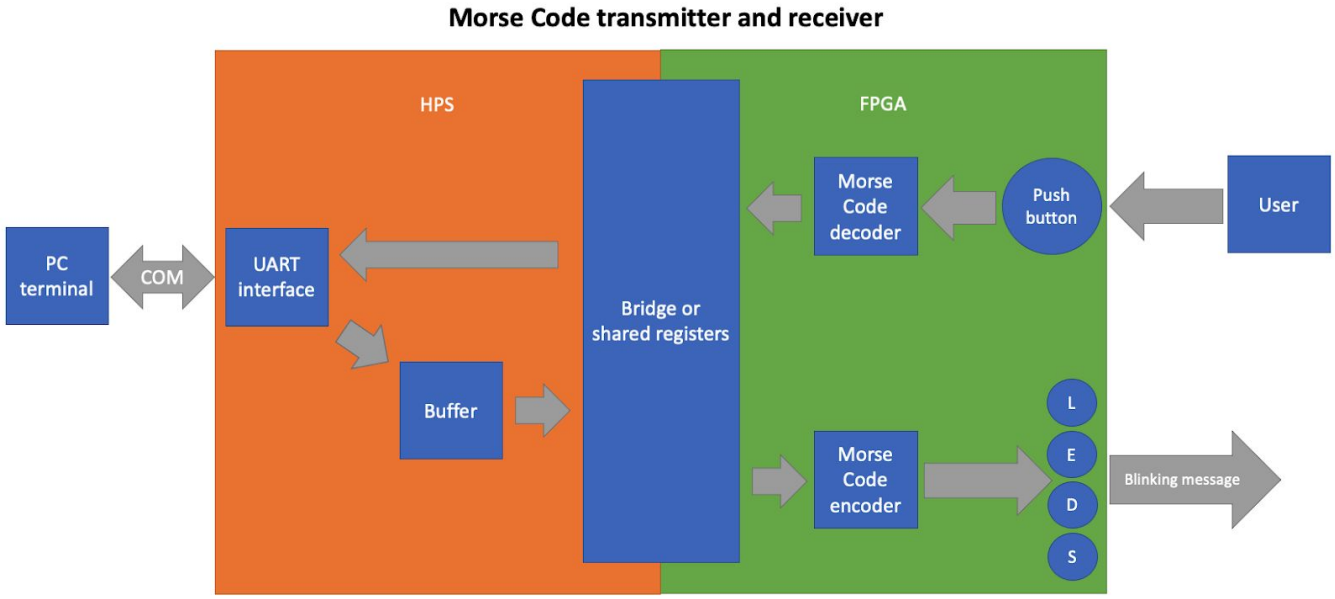

Figure 2.1.0 System's original proposal high-level block diagram

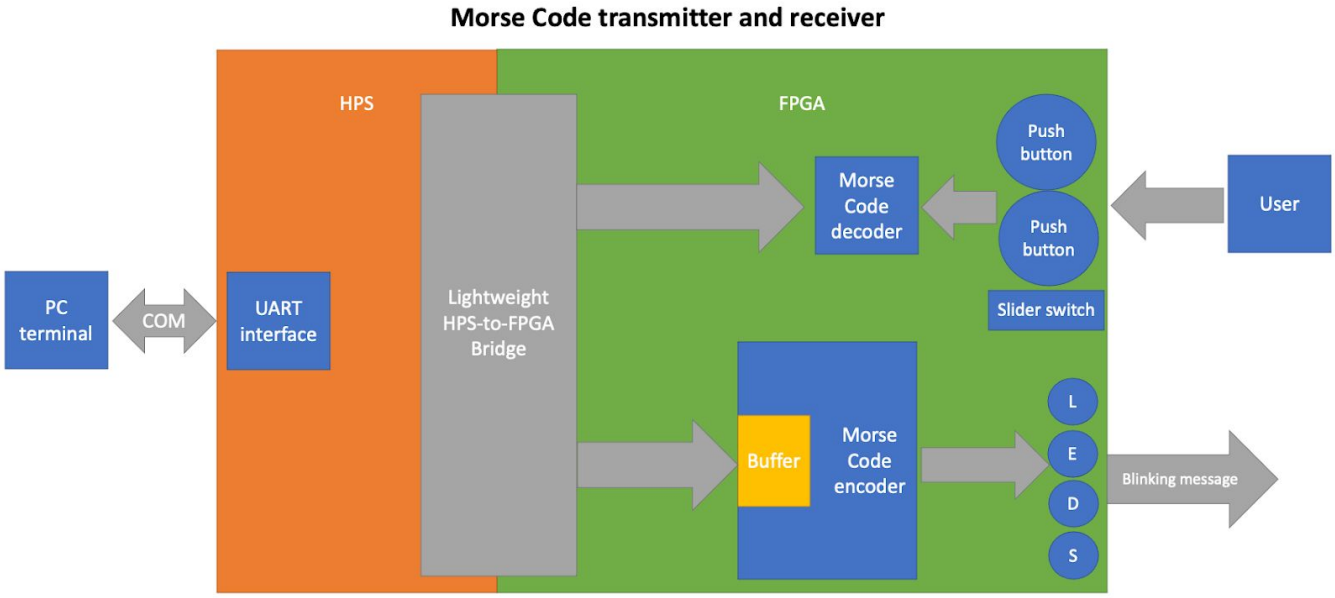

Figure 2.1.1 System's final implementation high-level block diagram

2.2 Project Goals

The goals of the project are:

1) Understand how the Morse Code is used to encode messages.
2) Establish bi-directional communication between the HPS and the FPGA on the SoC.
3) Implement a hardware block that decodes pulses in Morse Code into characters.
4) Implement a hardware block that encodes characters into dots and dashes.
5) Correctly interface the inputs (push-button & switch) and output (LEDs) blocks to the transmitter and receiver blocks.
6) Learn how to properly generate an input stimulus in Morse Code. As we will manually input the message in Morse Code to the system using a push button, we might need some practice on how to accurately do this (think of a telegram operator), in order to verify that we are attempting to send is what is displayed in our terminal as the detected message.
7) Develop a way to control the system's behavior (switch between send and receive mode).

3.0 Theoretical Background

The telegraph and Morse code had a profound effect on the development of the American West. Railroad companies used it to communicate between their stations and telegraph companies started to pop up everywhere, shortening the amount of time needed to communicate across the country. Later on, in the 1890's radio communication was invented and Morse code was used to transmit messages at sea. It became extremely important in maritime shipping and aviation. Pilots were required to know how to communicate using Morse Code until the 1990s. Now Morse code is primarily used among amateur radio users.

Along with the telegraph, the Morse Code was one of the very first methods for encrypting, transmitting and receiving a message using electrical signals. As future electronic engineers, we would like to pay a small tribute to the humble beginnings of telecommunications, and learn a bit more about this code that is almost forgotten and not used anymore. while learning about the capabilities of our SoC. This project will also give us some insight on how the more complex transmission and encoding protocols that are used nowadays are implemented using embedded systems.

Before entering fully to design our system we had to first understand the way Morse code works, from the valid symbols that could be encoded, and the timing parameters for each pulse duration.

At first we got familiar with how the alphabet looked like in Morse code, shown in Figure 2.1.0.

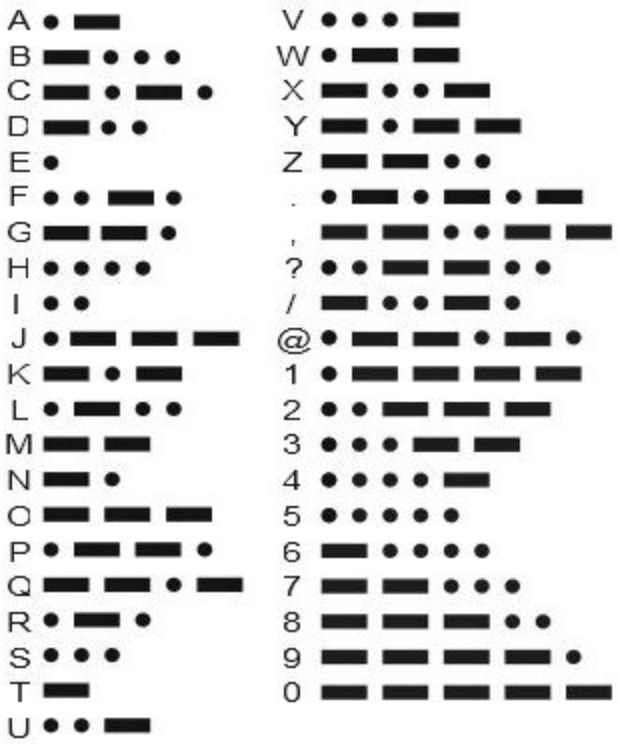

Figure 3.0.0 International Morse Code

We have the following rules:

- 1 dash = 3 dots
- the space between parts of the same letter =1 dot
- the space between letters = 3 dots
- the space between words =7 dots

These rules are important in order to define our timing parameters for the blinking LEDs.

4.0 System Requirements

Table 4.0.0 shows the minimal requirements for our project to be considered successful.

State	ID	Name	Description	Importance level	priority	comments
Active	SR-01	RX-detection	System correctly detects any Morse Code messages that it receives.	required	9	
Active	SR-02	TX-detection	System correctly encodes and sends any desired message in Morse Code.	required	9	
Active	SR-03	RX/TX-toggle	System can correctly toggle between transmit and receive mode.	required	9	
Active	SR-04	RX/TX-parallel	System can receive and transmit at the same time.	nice to have	1	UART is needed for both purposes(bonus feature)

5.0 Design Description

5.1 Morse Code Decoder Architecture

As mentioned in Section 2, two buttons and a switch are used to define a dot signal, a dash signal or a done signal. One button is used as the done signal. The second button is used to send a dot or a dash, depending on the state of the switch (switch=0 used for dot, and switch=1 used for dash). The following figures show the logic gates that represent each signal. Figure 5.1.0 represents a dot, Figure 5.1.1 represents a dash and Figure 5.1.2 represents the done signal. The buttons in the FPGA are active low, so we inverted this input for the signals to be 1 when pressed.

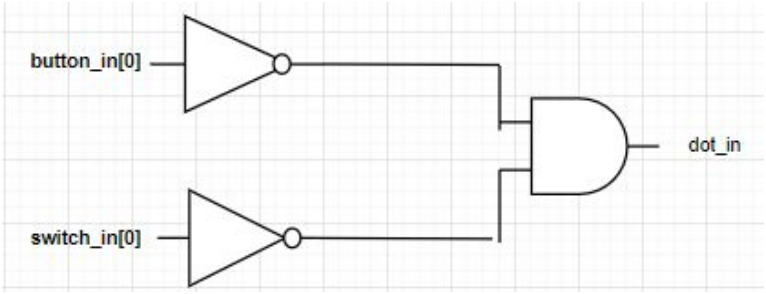

Figure 5.1.0 Logic gate of dot_in

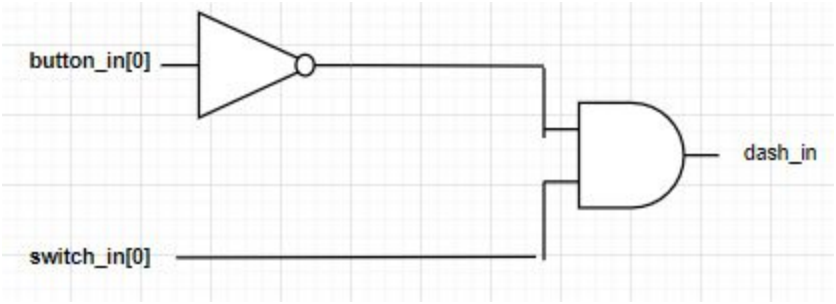

Figure 5.1.1 Logic gate of dash_in

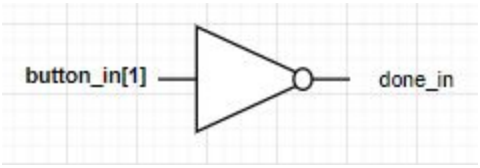

Figure 5.1.2 Logic gate of done_in

Besides the buttons and switch, the decoder module has the following input and outputs:

```
module MorseDecoder2(

    // Avalon interface
    input logic clk,
    input logic rst_n,
    input logic [1:0] address,
    output logic [7:0] read_data,
    input logic write_enable,
    input logic [7:0] write_data,

    // FPGA input
    input logic [1:0] button_in,
    input logic [3:0] switch_in   // connect whole dip switch interface, although we use only one

    );
```

These are used to create an avalon interface, so that the HPS can read data from our module via de Lightweight bridge. Data can be read and written from a 4 byte register inside the module. Each byte has a different function, as seen in Table 5.1.0 :

Register	Use	Description
register[0]	ready flag	The decoder sets this flag to 1 when a char was received. HPS must poll this flag to know when a char is ready to read, and clear it once it has done so, so it can detect the next ready for a future received char.
register[1]	received char	Stores the received and decoded char. Char “#” is stored when a non-valid Morse combination was received. There is no need for more registers to store received chars, because the HPS can poll this received char, faster than we can input a new one to the decoder.
register[2]	buffer in	The module has an internal buffer to store the inputs of dashes and dots. We mapped the contents of this buffer to this register for debug purposes.

register[3]	counter	The module has an internal counter to keep track of the inputs. We also mapped this counter to this register for debug purposes.

Table 5.1.0 Register use and description

Main internal signals of the module:

```
logic [4:0] buffer_in;          // input buffer to store dash and dots
logic [2:0] counter;            // count how many dots and dashes were input
logic [7:0] register [3:0];     // a register to be accessed by HPS
                                // register[0] will act as ready flag
                                // register[1] will store detected char
```

A FSM describes the behaviour of the decoder in order to translate an input from Morse to a char.

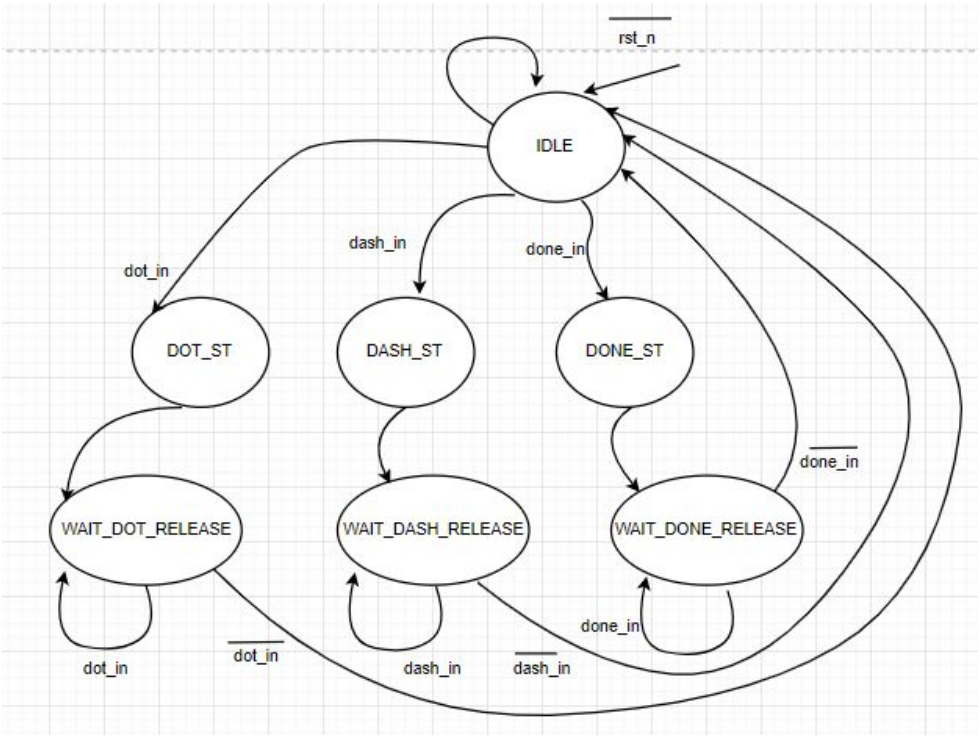

Figure 5.1.3 Decoder FSM

The state machine is idle until an input is detected. In the DOT_ST and DASH_ST, a counter is increased to keep track of how many dots or dashes the decoder has received. Also, in these states, a 0 (representing a dot) or a 1 (representing a dash) is pushed inside a buffer to keep track of them as well. The DONE_ST indicates that the letter on the number in Morse being received is complete, so it proceeds to store the detected char in register[1] as described above, set the ready flag and clear buffer and counter for the next reception. Both the buffer and the counter are important to correctly detect the input, as letters in Morse can use between 1 or 5 dots or dashes. All these states also have a WAIT_RELEASE state after them. This is to debounce the pressing of buttons. As the clock runs at 50MHz, it is almost certain that the buttons are going to be pressed for more than 1 clock cycle, so we need to handle this for the FSM not to move between states before it is needed. The machine stays in these states until the button is released and then it goes back to IDLE.

5.2 Morse Code Encoder Architecture

These are the inputs and outputs to the system:

```
module MorseEncoder(

    // Avalon interface
    input logic clk,
    input logic rst_n,
    input logic [4:0] address,
    output logic [7:0] read_data,
    input logic write_enable,
    input logic [7:0] write_data,

    // 7 LEDs in FPGA
    output logic [6:0] leds

    );
```

Similar to the decoder, we have the necessary ports so that the HPS can access an internal register. We have an output to the FPGA LEDs to blink them in the required fashion.

The main internal signals of the module are:

```
logic [7:0] register [31:0];     // 32-bit register accessible from HPS

logic [4:0] index;               // keep track of characters send

logic [2:0] queue [0:5];         // 6 position queue

// to generate delays (periods with leds off or on)
logic [31:0] counter1;   // dot
logic [31:0] counter2;   // dash
logic [31:0] counter3;   // short
logic [31:0] counter4;   // long
```

The functionality of each is as follows:

- Register: 32 bytes. register[0] acts a start and busy flag. HPS sets it to 1, and the transmission begins. It remains in 1 until the transmission is complete (serving as a busy flag if the HPS reads it). The remaining 31 bytes are used as a buffer, to store the chars that will be translated and transmitted to Morse.
- Index: used to keep track of how many characters have been set. It also helps to detect when the maximum amount of chars have been sent.
- Queue: it is a FIFO of states that the FSM (described below) will go through, depending on the char that will be transmitted.
- Counters: use to keep track of different delays, so that the LEDs blink with timing according to the Morse Code rules.

Keeping in mind what this signals represent, the FSM that describes the behavior of the encoder is the following:

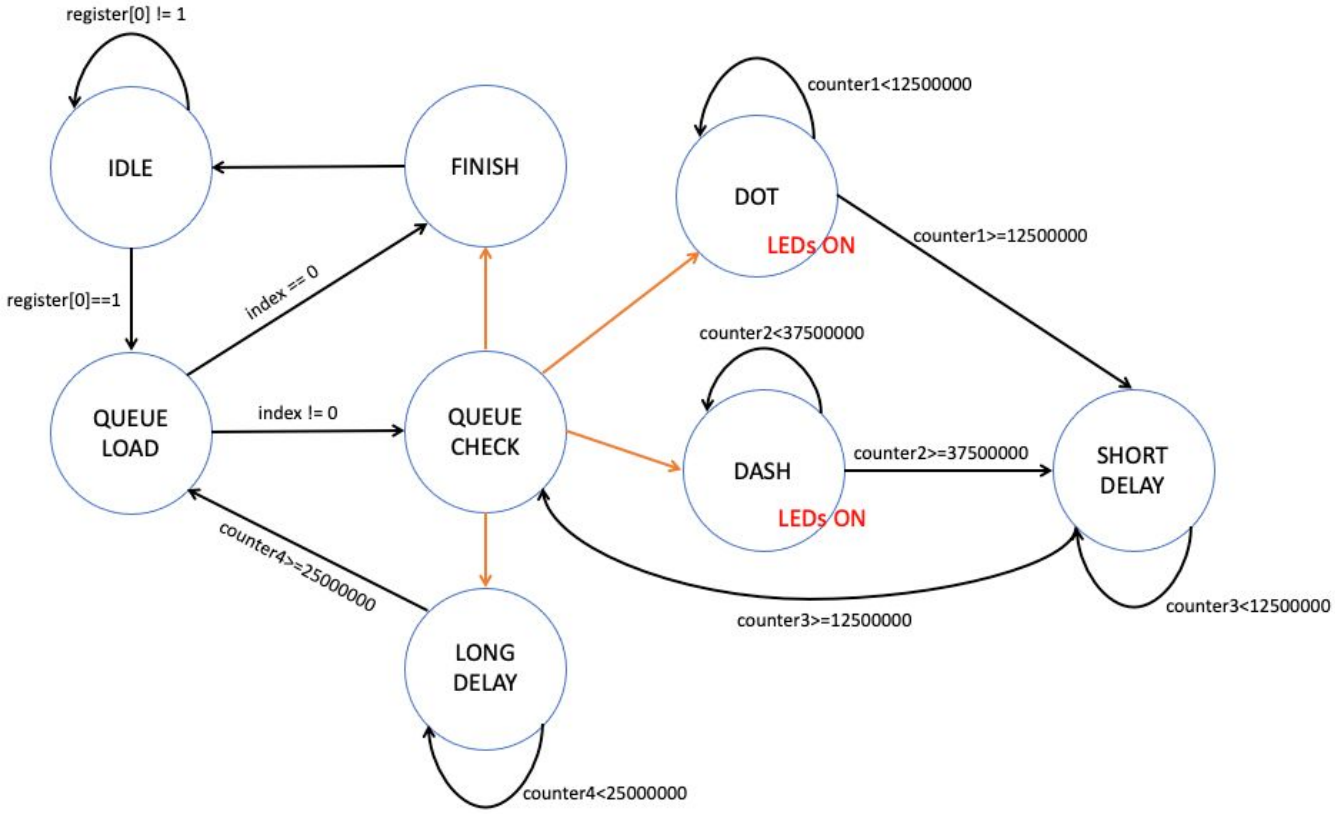

Figure 5.1.4 Encoder FSM

A description of what each state does is given below:

- IDLE: idle until start (register[0]) is set to 1. Counters and registers are reset here. Also this is the only state where a write to the register is possible, to avoid any corruption when transmitting.
- QUEUE_LOAD: this state checks to which element of the buffer index is pointing. Depending on the char stored in that position, it loads the queue with the needed states. For example:

```
case (register[index])
   "0": {queue[0],queue[1],queue[2],queue[3],queue[4],queue[5]} = {DASH,DASH,DASH,DASH,DASH,LONG_DELAY};
   "1": {queue[0],queue[1],queue[2],queue[3],queue[4],queue[5]} = {DOT,DASH,DASH,DASH,DASH,LONG_DELAY};
```

  If the char is “0”, then we need to go through the DASH state 5 times, as a “0” in Morse is -----. This translation is done for any char (letter or number, which are the basics in Morse code). The use of LONG_DELAY state is explained below.

- QUEUE_CHECK: This state checks queue[0], to set the next state accordingly. That is why in the diagram the arrows that leave this state are of a different color, indicating that it may go to any of those states. Sets counters to 0 in case they weren’t before, as they will be used next.
- DOT: This state increases counter1 by 1 on each clock cycle, until it reaches 12500000. This is equivalent to staying in this state for 250 ms (given that the clock runs at 50 MHz). LEDs are turned on during this state, indicating that we are transmitting a dot.
- DASH: Same as DOT state, but we use counter2 as reference to stay here 750 ms, complying with Morse code rule that indicates a dash must last 3x a dot. LEDs are on in this state as well.
- SHORT_DELAY: Morse code rules indicate that the time space between a dash or dot from the same letter or number must be the same length of a dot, so this state lasts 250ms with the LEDs off. In this delay, the queue push is done, so once we return to QUEUE_CHECK, it sends us to the next corresponding state.

- LONG_DELAY: Morse codes rules indicate that the space between different letters or numbers is the same of a dash. So, once we have finished sending the dashes or dots of a char, we go to this state to give this delay or separation. This is why we load this state at the end of the queue for each char. This lasts only 500 ms, because before arriving to this state we already have passed a short_delay before QUEUE_CHECK. So 250 ms from short delat plus 500 ms gives the required 750 ms delay, equivalent to a dash. LEDs are off as well here to give this visual spacing effect. This state increases index, so when it goes to QUEUE_LOAD after this state, the next char in buffer is read and queue load happens accordingly.
- FINISH: we get here when the transmission is over, or an invalid scenario is detected. For example, if an invalid char is loaded to the buffer, the FINISH state is assigned in queue[0] so the system does nothing for that case. This state resets register (including register[0], so now HPS doesn't see the encoder as busy), buffers and counters. We couldn't do this reset in IDLE state, to avoid corruption of the writes by HPS, which only happen in IDLE state.

5.3 Software

Our software gives the user the choice to use the decoder or the encoder, one at a time. Depending on what he choses, the HPS controls the corresponding module in the proper manner.

To control the decoder:

We initialize a pointer to the base address of the decoder module (assigned by Qsys). With respect to this, we can create a pointer to register[0], named ready_flag, and a pointer to register[1], named received, as show in the code below:

```
// decoder pointers
void *decoder_ptr;      // base
char *ready_flag;       // ready flag (register[0])
char *received;         // received char (register[1])
decoder_ptr = virtual_base + ( ( unsigned long  )( ALT_LWFPGASLVS_OFST + MORSEDECODER_0_BASE ) & ( unsigned long)( HW_REGS_MASK ) );
ready_flag = (char *)decoder_ptr;
received = (char *)decoder_ptr;
received++;
```

Now, as described in the functionality of the decoder (Section 5.1), all that is left is for the HPS to poll the ready_flag. Once it is set, it means a char has been received, and HPS can read it using. Once this is done, we display the char in the terminal and clear the ready_flag to read the future chars that may arrive. This is achieved with the following fragment of code:

```
while (1){
    if ((*ready_flag) == 1){
        *ready_flag = 0;
        printf("Received char: %c\n\r", (*received));
    }
}
```

To control the encoder:

We also set some pointers, according to the base address that Qsys assigned to our module:

```
// encoder pointers
void * encoder_ptr;     // base
char * busy_flag;       // busy or start flag (register[0])
char * encoder_idx;     // index (to fill buffer)
encoder_ptr = virtual_base + ( ( unsigned long  )( ALT_LWFPGASLVS_OFST + MORSEENCODER_0_BASE ) & ( unsigned long)( HW_REGS_MASK ) );
busy_flag = (char *) encoder_ptr;
encoder_idx = (char *) encoder_ptr;
encoder_idx++;
```

busy_flag points to register[0], which is used as start and busy signal, as described in Section 5.2. Pointer encoder_idx points to register[1] to fill the buffer starting in this position, and not corrupt register[0].

Now, in this mode the user can input the message to be sent. This message is then written char by char in the encoder buffer (register[1] to register[31]). We added some logic to stop writing if we exceed the 31

available spaces, to avoid corrupting the start or busy register. Once the message has been copied to the buffer, we set register[0] to start the transmission. Remember the state machine clears this register when done, so we poll it until it is 0 again to detect when the transmission is completed. The code that achieves this is the following:

```
while (1){
    printf("Input a message to transmit (use caps, no spaces):\n\r");
    scanf("%s",message);
    i = 0;
    while (message[i] != '\0'){
        *encoder_idx = message[i];
        i++;
        encoder_idx++;
        if (i == 31){           // avoid overwriting internal encoder buffer
            break;
        }
    }
    printf("Starting Morse transmission (LEDs should be blinking)\n\r");
    *busy_flag = (char) 1;      // send start
    while(1){                   // poll done
        if ((*busy_flag) == 0){
            printf("Done\n\r");
            break;
        }
    }
    encoder_idx = (char *) encoder_ptr;     // reset this pointer to initial state
    encoder_idx++;
}
```

5.4 System Interconnection

Qsys provides a high level block diagram on how the system is interconnected. It can be seen in Figure 5.4.0 below. It contains all the basic blocks from the GHRD project, but we can see our modules at the top in the pink square. Notice how the dashed blue line connects our avalon_slave_0 interfaces with the mm_bridge, and this bridge is connected to the h2f_lw_axi_master (red line), which is the Lightweight HPS-to-FPGA Bridge, as we defined in our specification. Our modules also have interfaces for the LEDs, the buttons and the switches (green and yellow lines). The GHRD project connects the actual FPGA pins to the led, button and dipsw_pio blocks, so we had to switch these pin connections to our interfaces in the soc_system.v file.

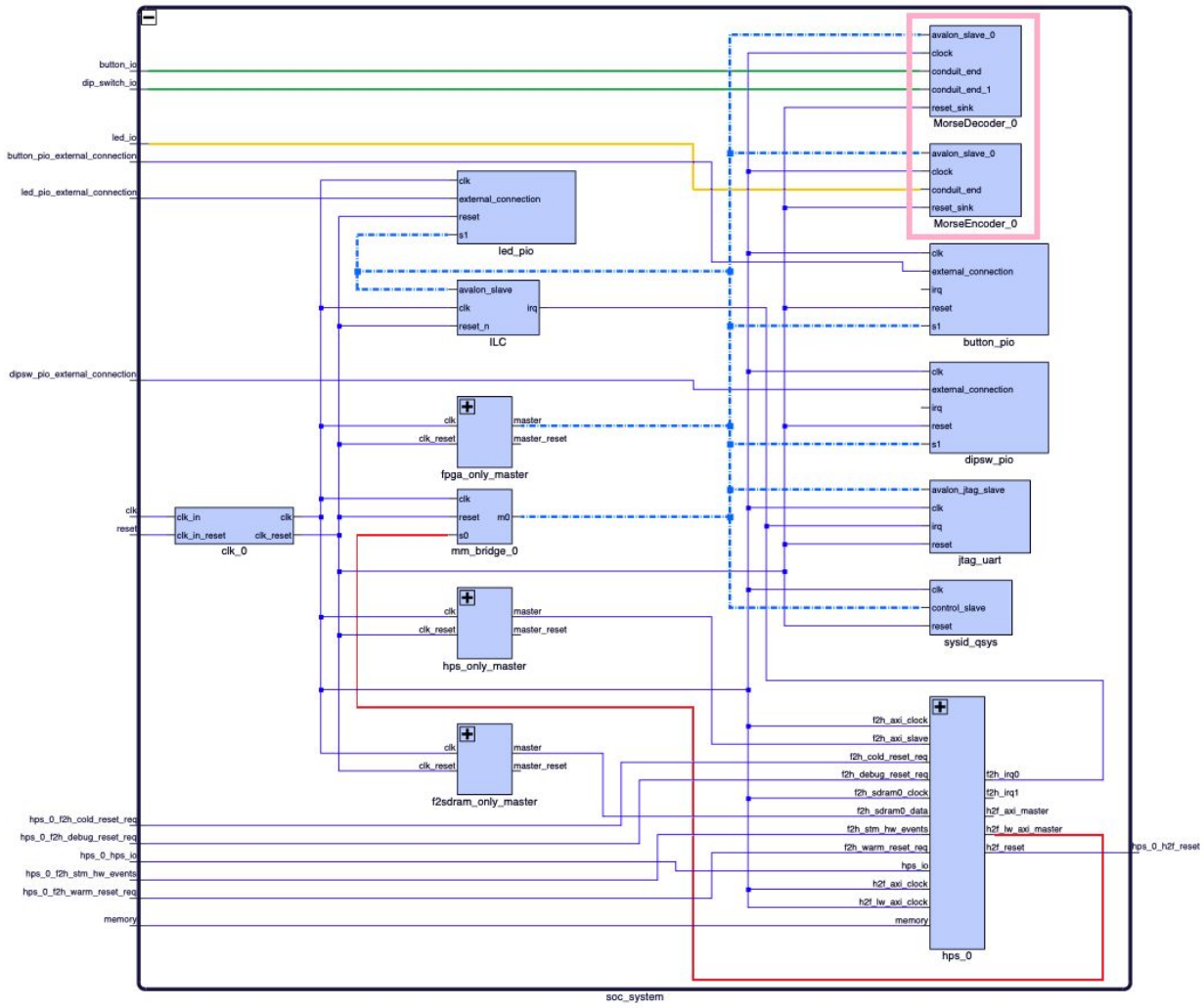

Figure 5.4.0 System Interconnection

6.0 Test and Results

6.1 Test Strategy

Before trying to make a custom IP and connect it to the HPS, we began by taking small steps. First one was to create a simple register in the FPGA, connect it as an Avalon slave, and then read with a simple C code using pointers. We verified that this worked by hardwiring values to the registers, so we could see them when reading, instead of default 0. Once this worked, we modified the register to change internal values once we pressed a button on the FPGA. With this working, we were ready to create our custom IPs. For every IP we made we needed to generate a testbench or simulation in order to debug failures and fix them before moving to the next step. The same applied when we interconnected the whole system; a profound test of the system's functionality had to be performed.

Our test strategy was first focused on the functionality of the individual blocks/modules of the receiver and transmitter. To achieve this, a testbench was made in order to simulate each one of the modules working.

Receiver

The simulation of module **Morse decoder** is shown in Figure 6.1.0.

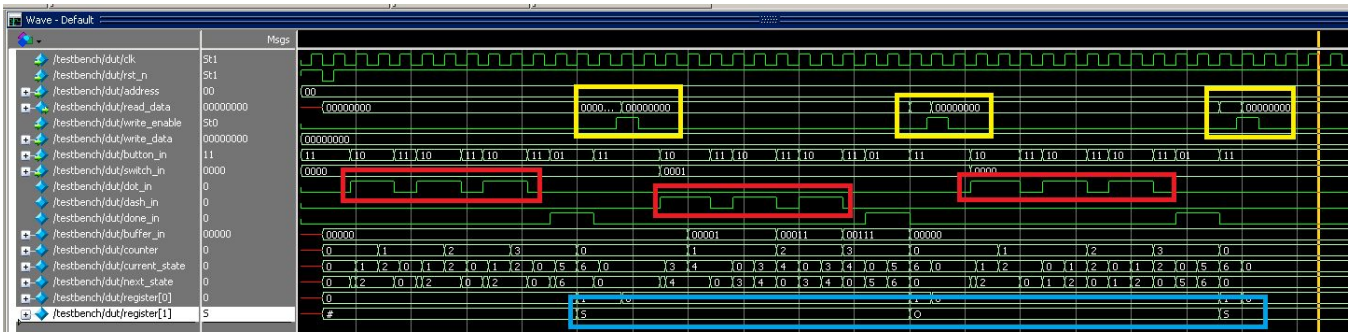

Figure 6.1.0 Simulation of receiver module

For this test, we send the classic Morse S-O-S signal to our module. Red squares show the DOT-DOT-DOT, DASH-DASH-DASH and DOT-DOT-DOT, and the correspondent letters are correctly detected and stored in register[1] (blue square). Yellow squares show the functionality of clearing the done bit as if the HPS was doing so.

Transmitter

The simulation of module **Morse encoder** is shown in Figure 6.1.1.

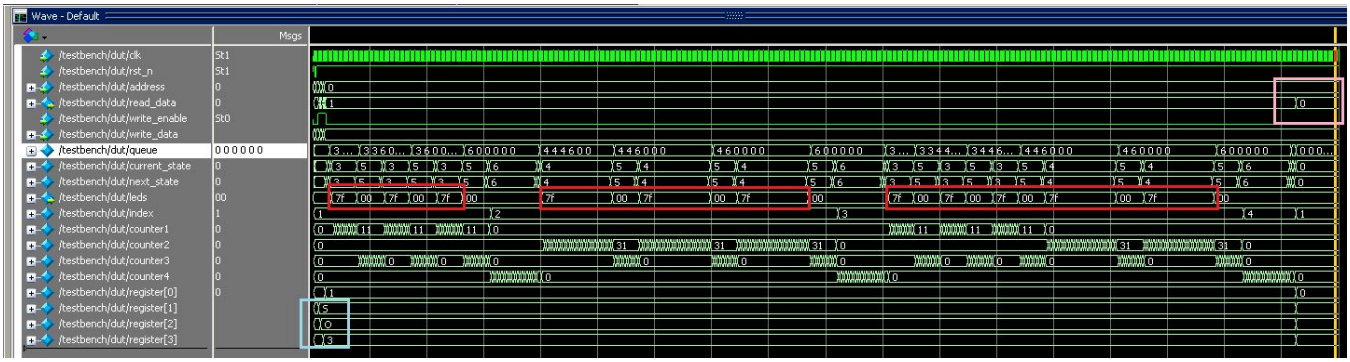

Figure 6.1.1 Simulation of transmitter module

For this test, we loaded the chars “S”, “O” and “3” in the encoder's buffer, and set the start bit (blue square). We can see the LEDs blinking as they should (red squares). A 0x7F is all LEDs on, we can see on the first block 3 sets of 0x7F of short duration; this is an “S”. For the “O” we see the same, but with longer periods on, representing 3 dashes that form an “O” and for the number “3” it is DOT-DOT-DOT-DASH-DASH, which can be clearly identified. Our queue works as required, and we can notice the busy bit being clear at the end (yellow square). Also, for this test we reduced the counters that generate the delay, to avoid long simulation times. However, we kept the ratio between counters so that the Morse code relative timings were met.

6.2 Test Results

Simulating our encoder and decoder was really useful, and a key step to detect failures in our system. In the case of the decoder, when we tried to load it in the board and test it, it didn’t. However, the simulation showed that the design should work as expected, so this led us to think what else could be wrong.

Reviewing the User Manual, we realized that the buttons of the FPGA are active low, meaning we had to invert that input to our module for it to behave as expected (our FSM considered detecting buttons pressed as a logic 1). Fixing this, the module worked correctly: we were able to display in the terminal the chars that our decoder detected as we pressed the buttons according to Morse code encoding.

Simulating the encoder was even more useful as we found some important bugs in the design, which made our tests fail when trying to use the FPGA. The queue is pushed in SHORT_DELAY state. However, as we stayed in this cycle multiple cycles, the queue was getting updated in every cycle; breaking the desired order that the FSM should follow. The same happened in LONG_DELAY state, where we updated the index variable, so it increased multiple times in this state. We noticed this on the simulation, and fixed it by adding a condition to only make these updates when the counter for that state is 0 (which just happens once in that state; at the beginning).

With the custom IPs fixed and working properly, we were able to test connectivity with the HPS by using simple codes C similar to the ones in Section 5.3. And it worked properly now.

Finally we modified the C code a little more, in order to have some sort of menu for the used to choose the operating mode, and once again we verified that we could use both functions.

7.0 Verification

In order to verify our system we first defined our objective evidence which is the system requirement in section 4.0 that had to be fulfilled. For this to be done a comparison had to be made between the expected result and the obtained result in order to define the correctness of our project. The technique used to verify our project was simulation using the ModelSim tool and testing our project in the DE10 nano board. On table 7.0.0 we can see the system requirements with their verification action and a column that indicates if the requirement was met or achieved.

	System Requirement	**Verification**	**Achieved**
1	System correctly detects any Morse Code messages that it receives.	This was verified in the terminal's output	☑
2	System correctly encodes and sends any desired message in Morse Code.	This is verified by observing that the LEDs blink in a proper fashion.	☑
3	System can correctly toggle between transmit and receive mode.	Using a menu in the terminal	☑
4	System can receive and transmit at the same time.	*bonus feature	☐

Table 7.0.0 System requirement verification

8.0 Conclusions

8.1 Vanya Michelle Medina Garcia

Overall we consider our project to be successful since it achieves the acceptance criteria mentioned in section 6.0, we have left the bonus features as a future challenge in our semester. From an overall perspective I consider the project to be challenging in every aspect since it was our first time working with qsys and interconnecting HPS with FPGA. Along the project some first approaches had to be changed in order to optimize and correctly implement our design. It was very helpful for this project to first compile and have running the projects suggested in the manual, as in MY_FIRST_FPGA, MY_FIRST_HPS, and the GHRD project. This gave me a reference on how the qsys works and how the pins are connected .I also made a review of how the morse code works and how it was useful some years ago. The most challenging part was the encoder and decoder which at the end were easy to follow ones having the FSM corresponding to their behaviour. This project was a review of the programming languages seen in past courses as in System Verilog and also of the interconnection of HPS and FPGA. We left some bandwidth in order to optimize and include all the rules from the morse code in the future.

8.2 Luis Alfredo Aceves Astengo

For me, the main challenge for this project was to interconnect the HPS with the FPGA. I had used DE10 boards in the past, but only for the FPGA fabric, which has an easier procedure to load a design into it. However after a few attempts we learned and were able to interconnect custom IPs with the HPS. As described in Section 6.2, all the system failures were identified and solved. One of the main struggles for me was to find where the pins to the FPGA fabric were, in order to connect them to our modules, as we couldn't do that in Qsys (or at least we didn't know how). However I found them in the soc_system.v file, and "stole" them from the pio blocks generated by the GHRD. I don't know if this is the proper approach, but fortunately it worked. Future improvements for the project that I would have liked to have but didn't have enough time are: add a reset button to restart a module (we just have the general reset of the SoC to our modules), allows reception and transmission at the same time by using interrupts in the HPS, and perhaps improve the code to detect user mistakes (like writing something invalid to encoder buffer). Overall, the project was quite a challenge, but I enjoyed making it function, and I found it interesting to combine concepts from past courses (like previous Verilog knowledge, or C programming for microcontrollers) with what we learned in this course.

8.3 Eduardo Garcia Olmos

The project was implemented successfully, we were able to implement a Morse code receiver and transmitter using both the HPS and FPGA. As we already had experience with the FPGA DE10 boards the challenge of the project was to interconnect the HPS and FPGA through the AXI Bridge, we did this by customizing our IPs. We had different challenges and failed attempts to overcome during the making of our project, but we successfully found the issues and worked through them. We had troubles working with the Qsys platform designer as we didn't have any more experience than the User Guide tutorial example. I can't say I'm already comfortable working with Qsys, there are things that I haven't completely understood how to configure properly or the effect it has physically in the connection to the board. Future improvements I think the project can have is to make it more user friendly, maybe adding a display or make it fool proof in the means of the input characters it receives and the procedures to be redundant. The project put to the test our knowledge in topics related to the FPGA and made me see there are many areas I can improve my knowledge but had a great time troubleshooting and combining C programming and Verilog.

9.0 References

[1] Become a Morse Code Expert (2019). *Art Of Manliness.* Retrieved from https://www.artofmanliness.com/articles/morse-code/

[2] Making Qsys Components (n.d.). *Intel.* Retrieved from ftp://ftp.intel.com/pub/fpgaup/pub/Intel_Material/16.1/Tutorials/Making_Qsys_Components.pdf

[3] Morse Code Decoder (2013). *Computer Science University of Toronto.* Retrieved 18 October 2020, from http://www.cs.toronto.edu/~milic/csc258/report.pdf

[4] DE10 SoC Nano Basic Guide (2017). *Github.* Retrieved 18 October 2020, from https://gist.github.com/addisonElliott/8c229e47184a724d1a00328110dc094c

[5] DE10 Nano User Manual (n.d.). *Intel.* Retrieved 18 October 2020, from https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-4302081511597-de10-nano-user-manual.pdf