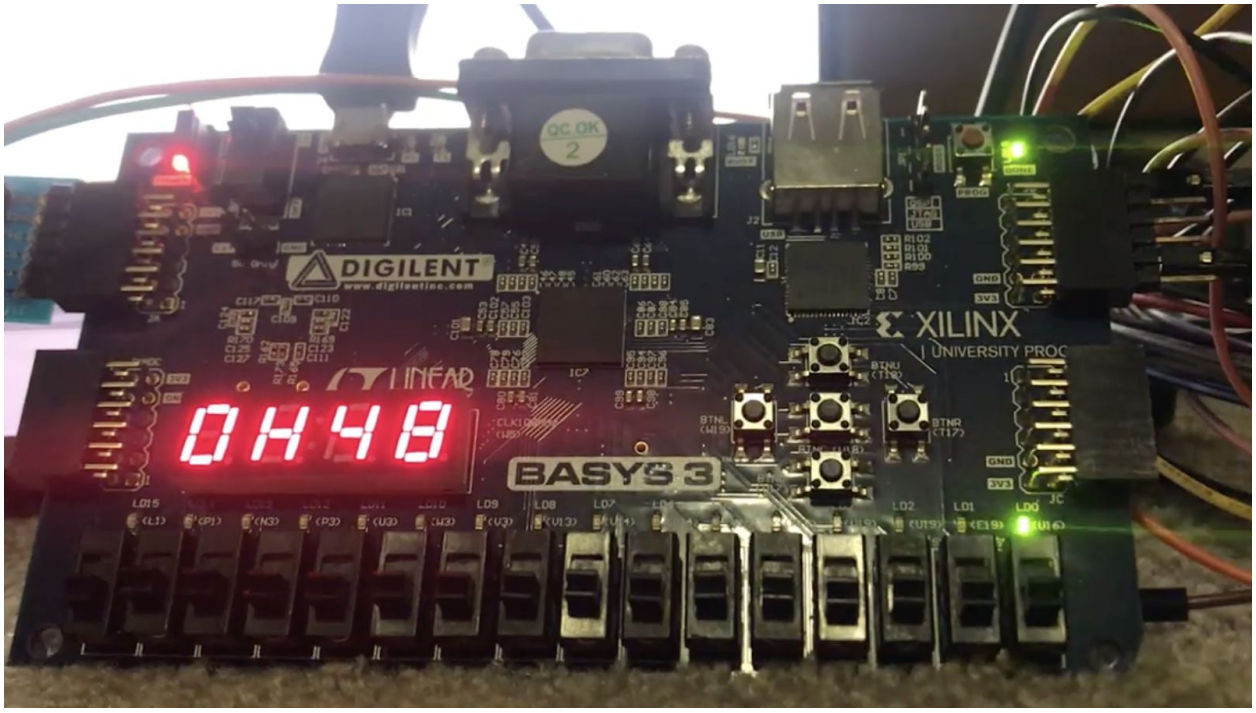
Audio Visual Morse Code with Basys 3 Board

ENGS 31 - Digital Electronics



27 May 2020

Authored and Created by:
Garret Andreine, Wesley Heim, Shane Hewitt

Abstract

The implementation and descriptions provided in this report are in reference to an audiovisual implementation of morse code through a Basys 3 board. Morse Code is a series of time sequences called ‘dits’ and ‘dahs.’ Our implementation is divided into four logic blocks. The input to the logic is a series of letters, making up any type of encoding or sentence. Each individual letter is received and transmitted in our UART block. Then, each letter gets put into a Queue which holds all letters in the same order as received. Once the sequence of letters is done transmitting to the logic, the Queue starts dequeuing letters to translate into morse code. Each letter is dequeued and the value of the letter’s ascii encoding is referenced to a look-up table which holds a 20 bit timing sequence that will be interpreted in the final output management block. The output management block interprets the 20 bit signal and determines the times in which the sound and light need to be ‘on’ in order to communicate each letter properly. Through this data flow, we implement a device which takes in an input of letters and translates that sequence of letters into audio and visual morse code. The following documentation describes the data flow as well as our experience as we implemented the design.

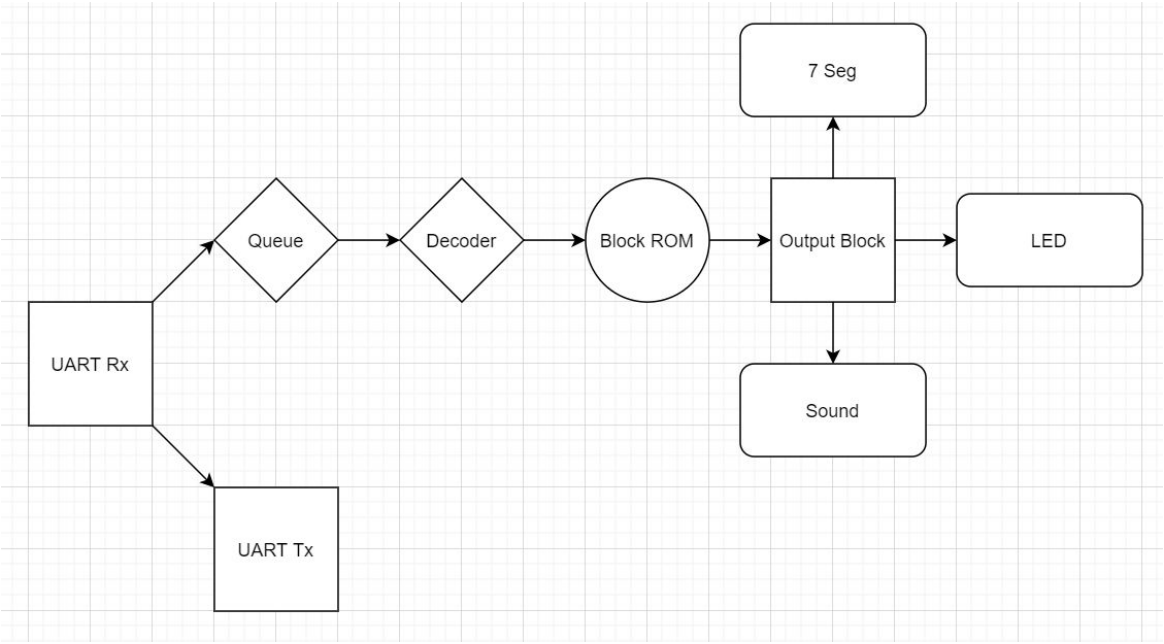


Table of Contents

Introduction

The project described in this report is to develop logic that is interpreted and output via a flashing LED light as Morse Code. The input can be any alphanumeric sequence, including spaces. The input terminates when the user hits ‘enter’ or ‘return,’ which indicate a new line or a ‘\n’ character. Our design meets all specification requirements and also includes an output onto the seven segment display on our Basys3 board. The output on the display is the hexadecimal encoding of the ASCII value currently being output.

The central data flow consists of five blocks that work together to implement our design and accomplish the task of translating letters into morse code. The blocks are: the UART Rx/Tx, Handle Input, Decoder, and Output Management. A couple additional blocks were added for the 7 segment display as well as the audio output. These included: the oscillator signal for audio output and the seven segment display for outputting ASCII values.

Approach

To start this project the first thing we did is schedule a time as a group where we could spend time together to work on the project. We selected a 9 pm to 11 pm EST as a two hour block that we could all meet and collaborate together and always met at this time from the beginning of our progress on the project (May 25th) to the end of this project (June 6th). We thought this would be a good way of holding each other accountable, communicating our questions and thoughts to each other, and as an overall good way to make sure we make progress on a timely schedule rather than procrastinating it till the last minute.

With this general collaborative environment we came up with a list of priorities and just all took one on the list and when we finished would work on the next item. We didn’t set specific roles as much as we just all worked a reasonable amount and took on components and tasks we

were each suited to. If someone didn't feel as strong on something, we were all working on the same Zoom call so it was very easy to ask questions or elaborate on the design as we went. Often we worked on things simultaneously,particularly with debugging.

Our priorities list is as follows with the big leftmost row of bullet points being top priorities, the middle row of bullets being secondary priorities, and the rightmost row of bullet points being tertiary priorities:

- UART Receiver/Transmitter
- Queue Logics
- Decoder/ROM Memory File
- Output Block logic
    - Testing each component and creating wavegens
    - Working on wiring all of them together in a top shell
    - Connecting the LED and Buzzer
        - Final testing of whole program
        - Video Recording/Documentation

System Design

Our first meeting consisted of us satisfying the requirements of “Checkpoint 1” and figuring out how we would go about attacking this project. Upon much deliberation , we came up with a Queue system that would take data from the UART receiver and hold it in a queue until an end character came through the UART. After this the queue would spit out the ASCII data it received from the UART into a decoder that would translate this into a string of 1s and 0s that represent our Morse code. This string of data would then be interpreted by an output logic block that would then create the signal for the buzzer and LED of the Basys 3 Board.

After having broken our project down into this higher level we then came up with our list of big components:

- UART

  → Gets the data the user enters and provides it to the system

- Input Logic/Queue

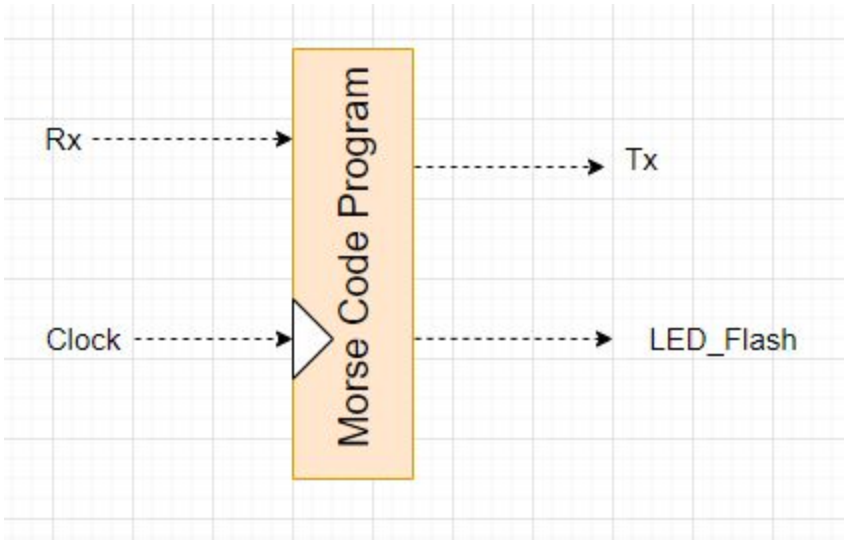
  → Takes in the user data until end character then spits it out again chunk by chunk to the Decoder

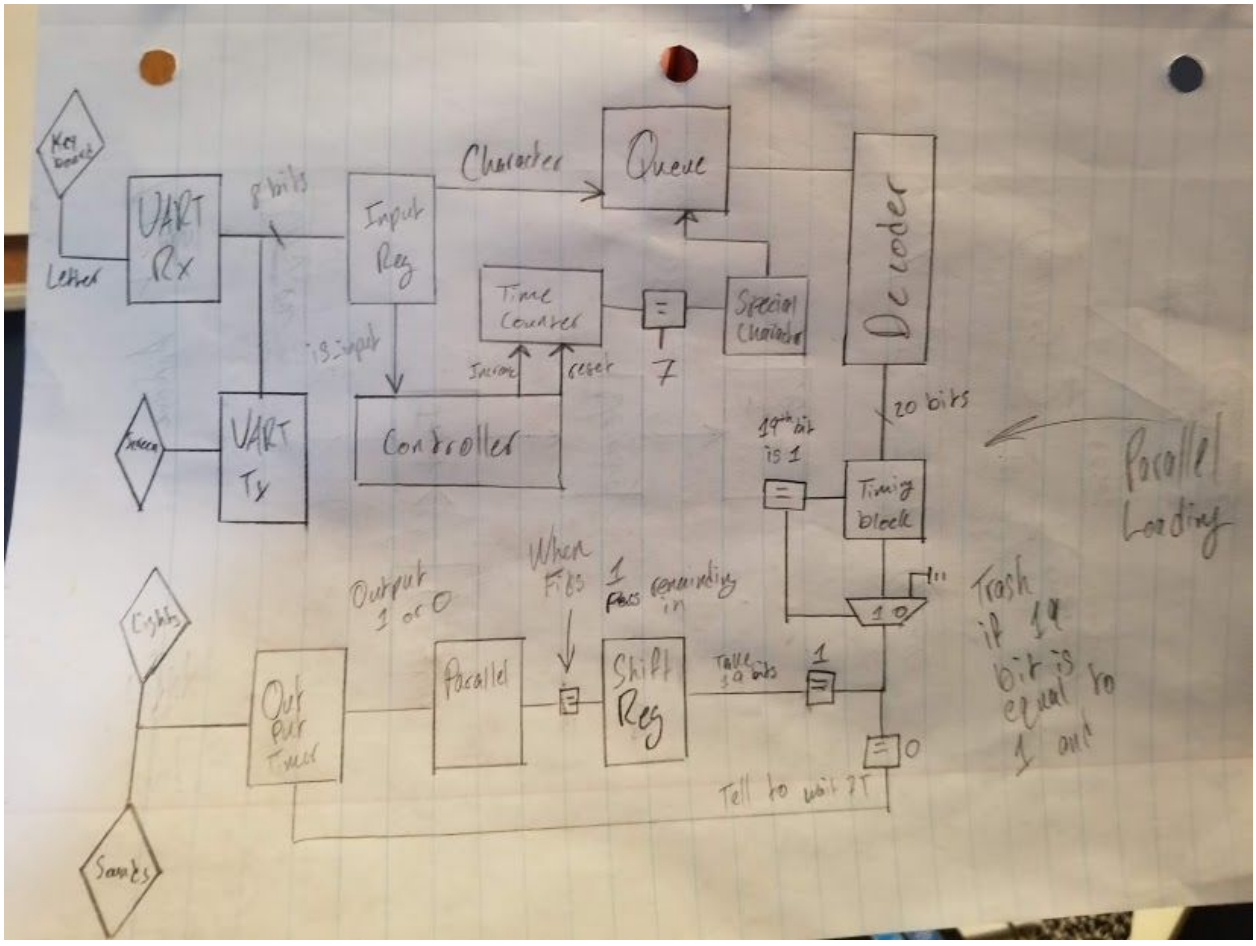
- Decoder/ROM Memory

  → Takes in an 8-bit ASCII code and translates it into Morse code where 1 represents high and a 0 represents a low, with 1 being a dit and 111 being a dah

- Output Logic

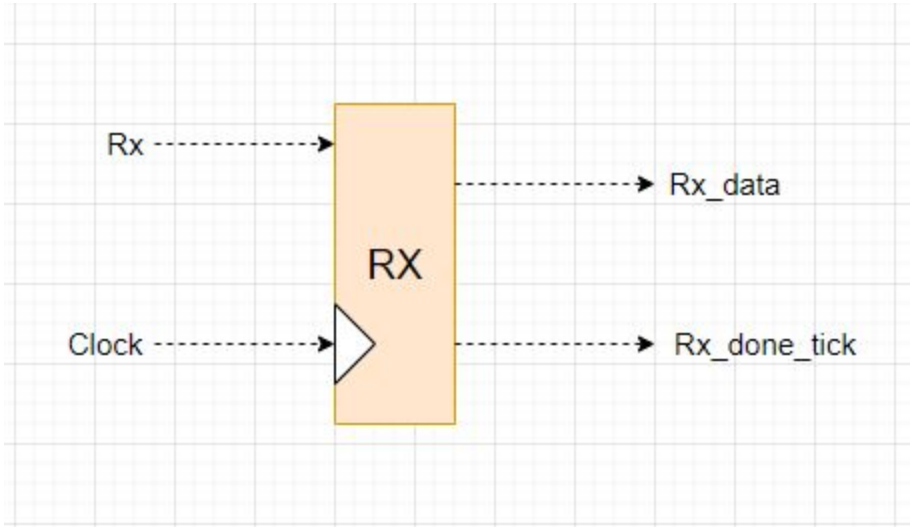
  → Take this string of 1s and 0s and output it appropriately with the correct time/delays.

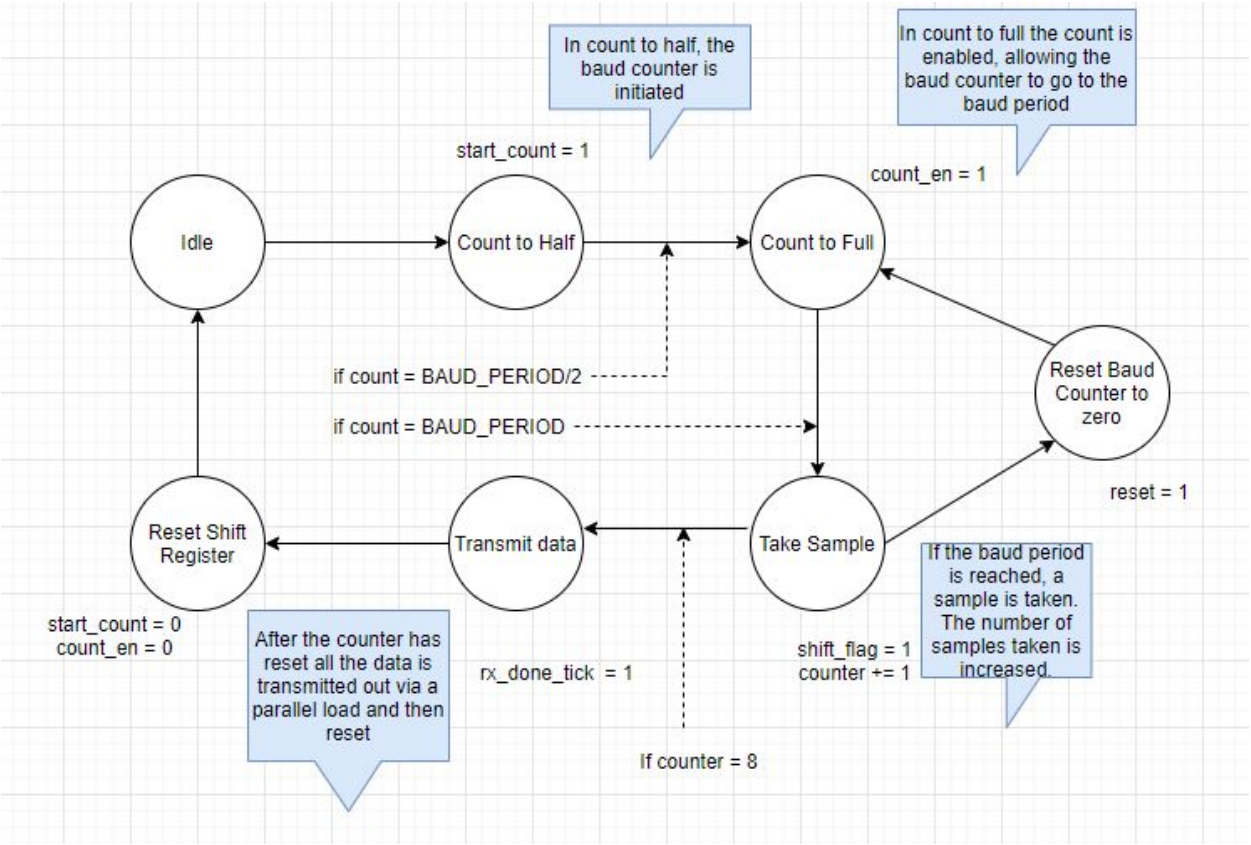

Rx
Clock
Morse Code Program
Tx
LED_Flash


Key Board
Letter
UART Rx
8 bits
Input Reg
Character
Queue
Decoder
Time Counter
=
Special Character
7
is_input
Increment
reset
Controller
Interrupt
UART Tx
19th bit is 1
20 bits
Timing block
Parallel Loading
Output 1 or 0
When Full
1 bits remaining in
1
0
Trash if 19 bit is equal to 1 out
Clocks
Output Timer
Parallel
Shift Reg
Take 19 bits
Tell to wait 2T
Samples

UART Receiver

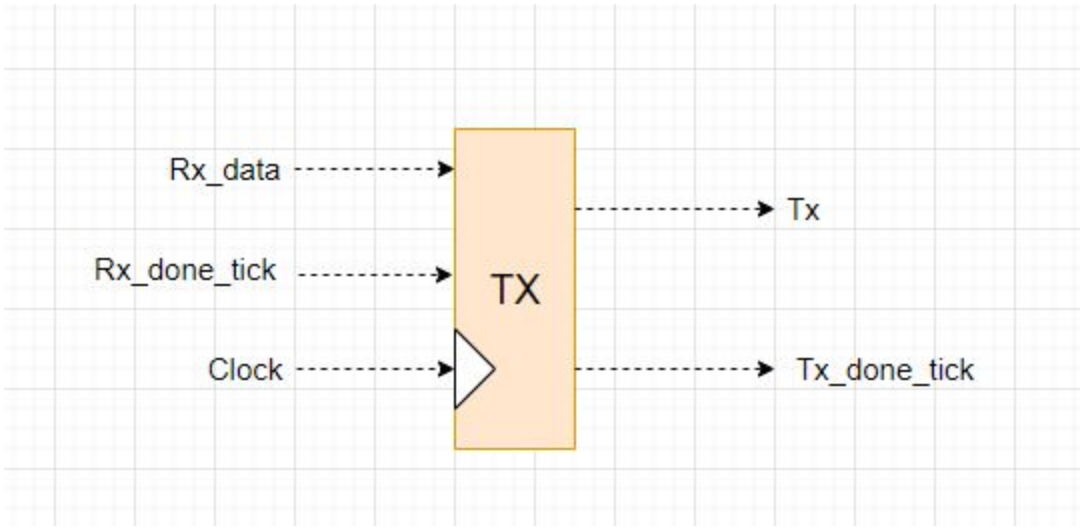
The UART Receiver follows the traditional role of a receiver in the Universal Asynchronous Receiver-Transmitter. The UART receives a 8 bit string at a baud rate of 9600. To do this it takes in a start bit, the 8 bit string, then the end bit. When the starting bit is received, signified by a falling edge, a counter is initiated that counts to half the baud period (determined by the clock divided by the baud rate). At this point halfway through a transmission, the baud counter is reset. From this midpoint of a signal the baud counter is initiated and then when it reaches the baud period value a sample is taken of the provided data, allowing it to be as accurate as possible when obtaining the data represented in that bit. After this initial sample is taken, 7 more samples are taken to obtain the full 8 bits of data sent by the UART protocol. This data is then transmitted out to the system through an out signal called rx data and a monopulse to signal that there is data is sent out, this is called rx done tick.

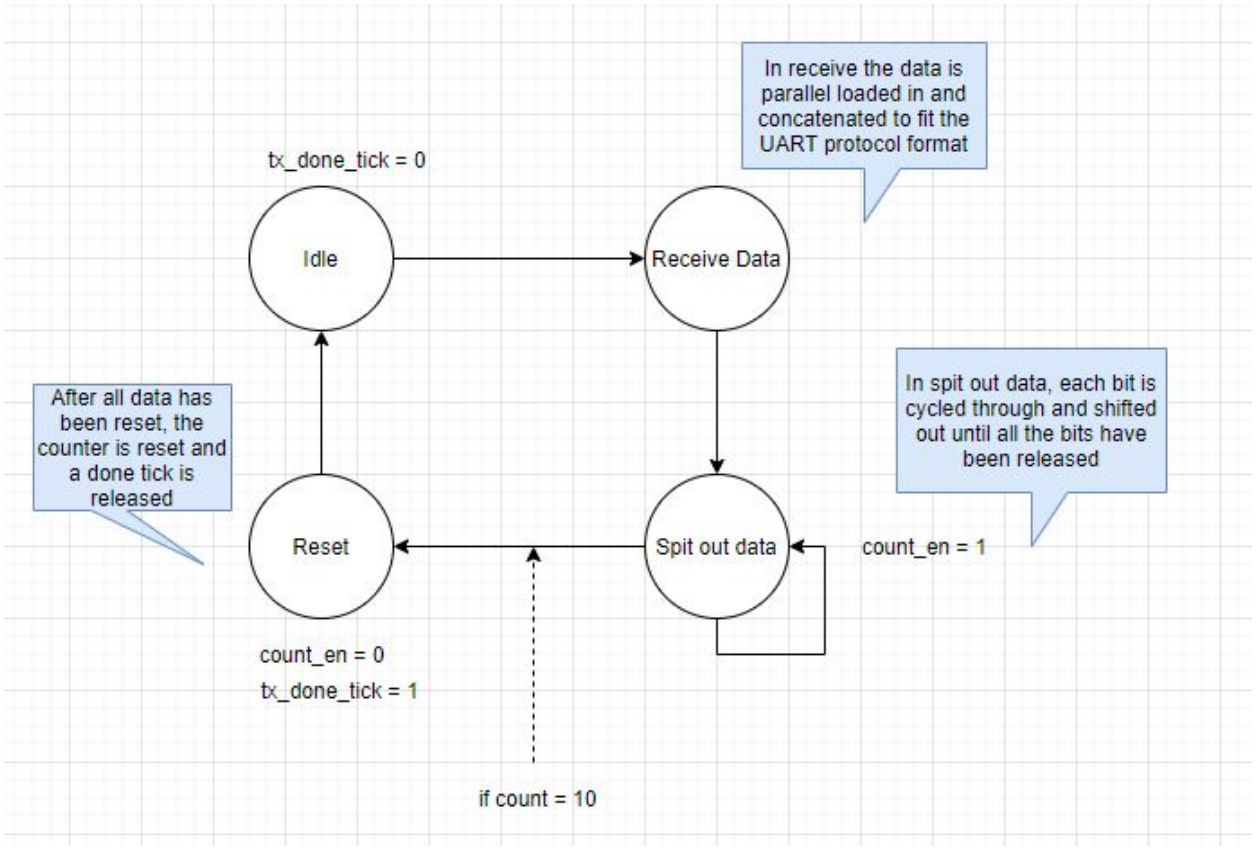




UART Transmitter

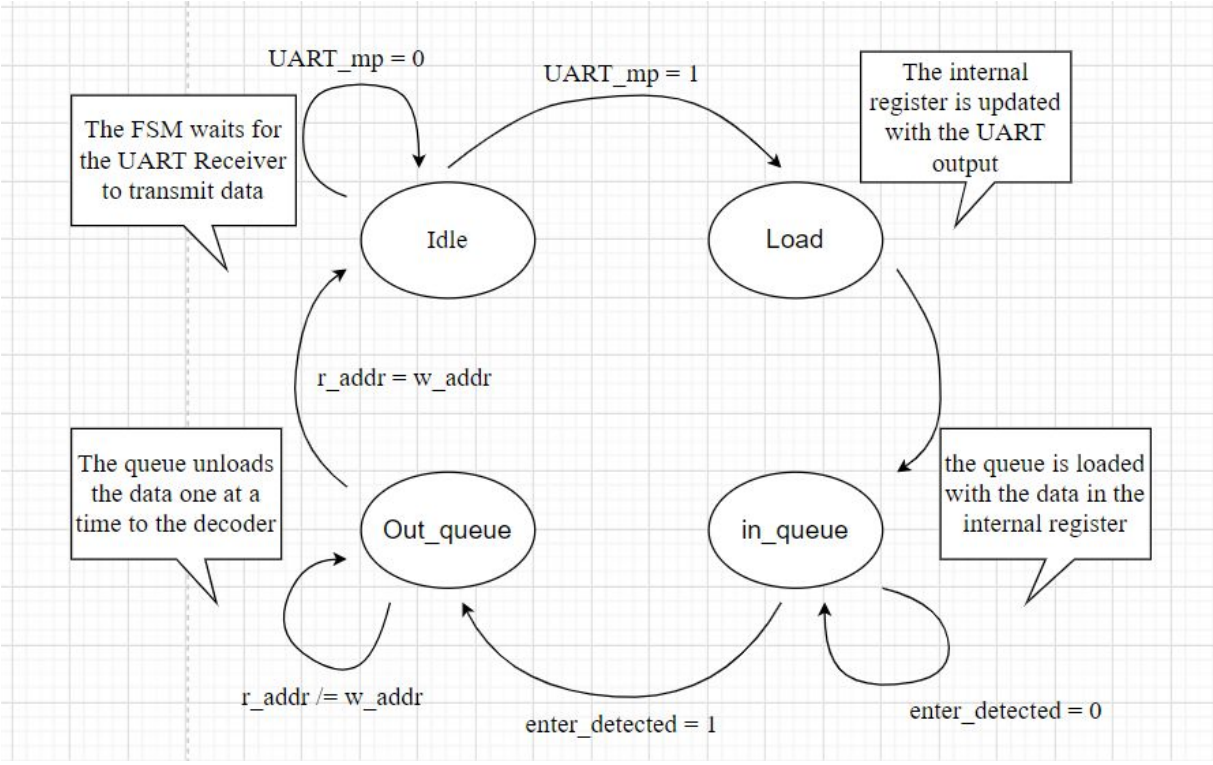
The UART Receiver follows the traditional role of a transmitter in the Universal Asynchronous Receiver-Transmitter. The transmitter receives a 8 bit string from the receiver via a parallel load method. After receiving this data it is enabled by the receiver done tick mentioned above. After being enabled it concatenates the provided data with a starting bit in the front, represented by a zero, and an end bit at the end of the string, represented by a one. After concatenating, this string is moved into a shift register. Every bit is then shifted out at the baud rate through the form of an output called Tx. This process continues until all ten bits have been released. Upon this release a signal showing that the process has completed is sent out, this is the tx done tick.


Rx_data
Rx_done_tick
Clock
TX
Tx
Tx_done_tick


In receive the data is parallel loaded in and concatenated to fit the UART protocol format
tx_done_tick = 0
Idle
Receive Data
In spit out data, each bit is cycled through and shifted out until all the bits have been released
After all data has been reset, the counter is reset and a done tick is released
Reset
Spit out data
count_en = 1
count_en = 0
tx_done_tick = 1
if count = 10

Input Logic

The input logic is a state machine with an associated datapath in one .vhd file. A state diagram detailing the FSM logic may be seen below. This block's purpose is to take 8-bit vectors representing ASCII characters and store them. After an 'enter' character is detected, the queue begins to unload, waiting for the signal decoder_in_en from the decoder before supplying another 8-bit output. At this time, the decoder simultaneously reads char_out, as the next value is already ready at this moment. The queue has a capacity of 150 8-bit vectors, and deletes all characters after they are outputted. The read and write addresses continuously increment, and if there is a very long session, they roll over back to 0 after 150 characters transmitted. The FSM stays in the out_queue state after an enter character until the queue is empty, while the datapath controls when and which character to output.



Memory Map

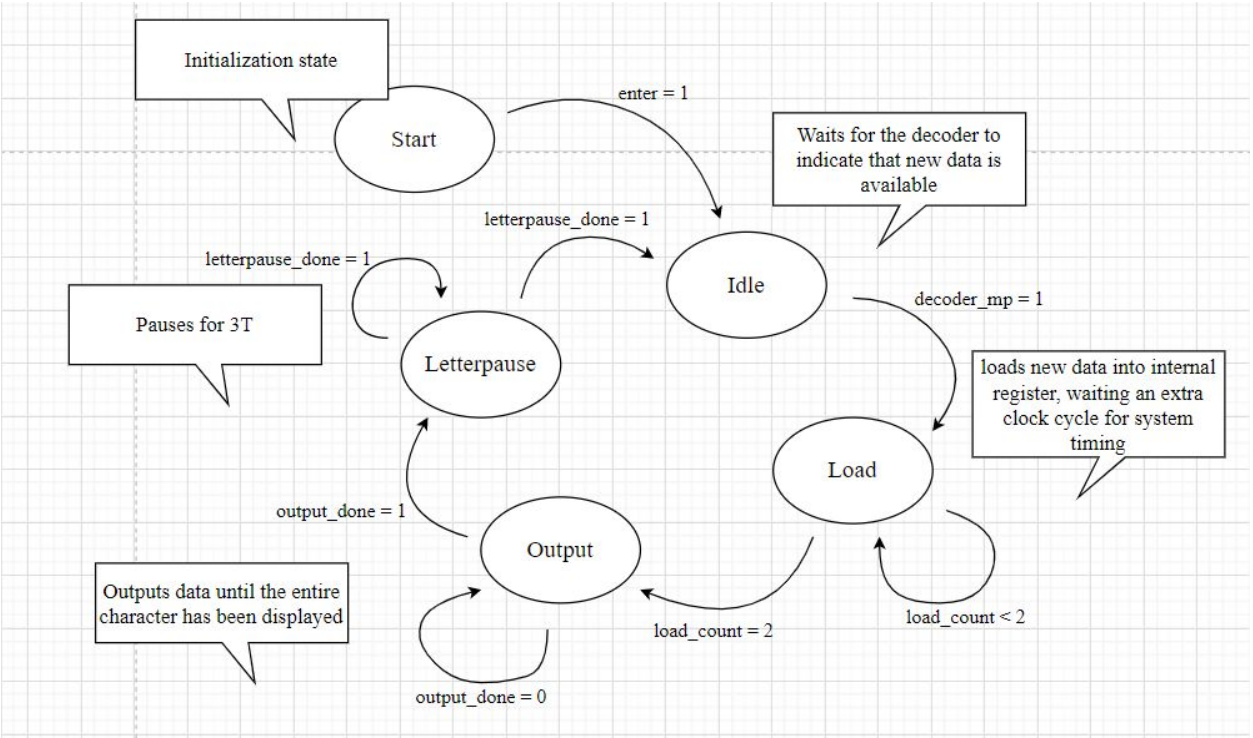
The memory map is generated as a Block ROM from a coe file. The coe file was generated manually as there is not an observable pattern in the output of morse code. Each address stores a 20 bit series of zeros and ones that indicate a pattern of dits and dahs. For example, the memory stored at address 0 was meant to represent the letter “A.” The bit sequence was “00000000000000010111.” This strategy was implemented for ease of output logic timing as we could iterate through the bits and wire this to one signal which is shown on the LEDs of the Basys 3 board.

This block provides functionality for both upper-case and lower-case letters. The decoder interprets their different ASCII values and references to the same spot for the look-up table to output.

This block was combined with a decoder that took in the ascii encoding of the letter we wanted to output and changed that encoding to an address in the Block ROM. As a result, the Decoder and the Block ROM were tested in tandem.

Output Timing Block

The output block receives a 20-bit vector intended to represent up to 19T of binary morse code. The first bit of the vector indicates if the character is a space or not, and the following 19 bits contain the character's timing, with zeroes padded on the left such that the final bit is always a 1 (except for the space character). The output logic operates with a finite state machine, shown below, that enables different datapath operations. The creation of the output led_buzz signal is done by using an integer to go through the vector bit-by-bit. All padding zeros are ignored, and the logic increments with the desired led_buzz output for a period of 1T for every non-padding bit, with 3T low after every character. In the case of a space, the logic simply waits for 4T, as it is after the 3T post-character delay, equaling a 7T delay.



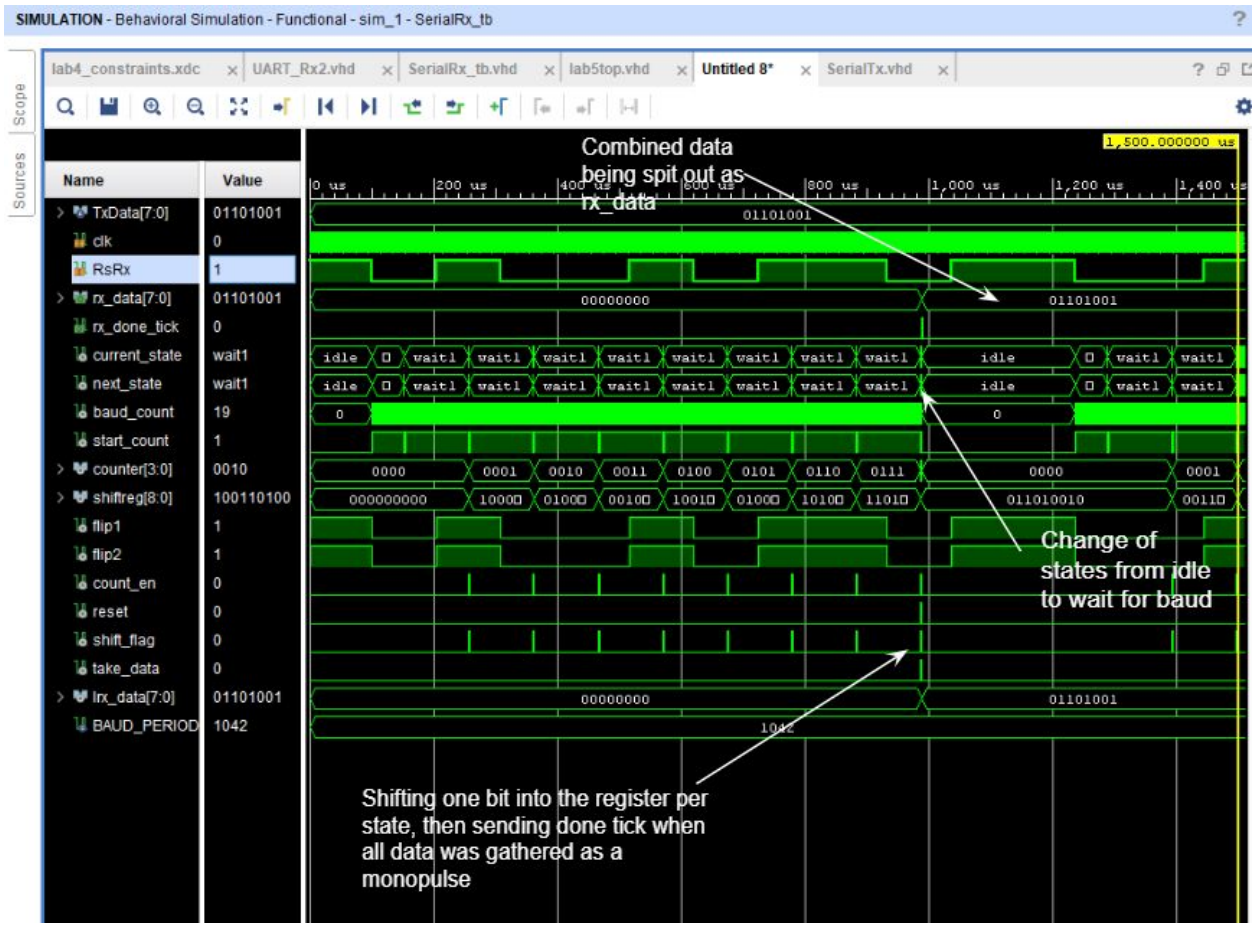
7 Segment Display

The 7 segment display component displays the hexadecimal value for the character that is currently being output by LED flashes. This component hinged off of the code for previous Labs in Engs 31 that we have completed. We had to make some noticeable changes. Each output is interpreted by the internal multiplexer as an 8 bit signal rather than 4 as seen previously. We needed to add the 4 bits to be able to output an ‘x’ in the 3rd slot (indicating the hexadecimal nature of the output) while preserving all potential hexadecimal values 0 - F. This component updates upon a new signal being transmitted. This signal comes from the Decoder, which indicates that the output logic is transmitting a new character.

Testing

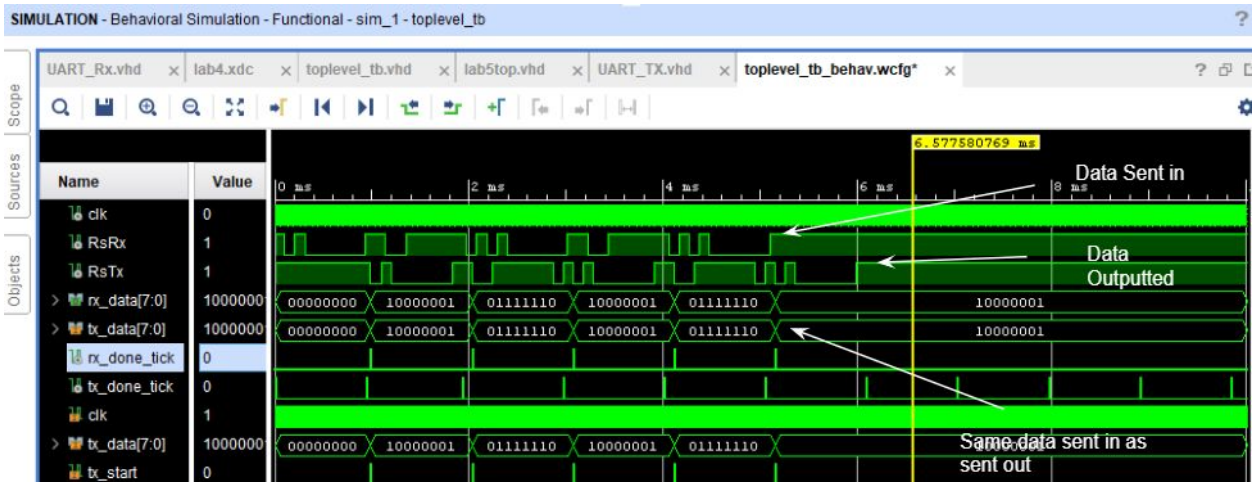
UART Receiver Testing

For testing the receiver a test bench was designed (see appendix) that would simulate the process of sending a string of bits at the specified baud rate. The resulting waveform was produced:



UART Transmitter Testing with Receiver

For testing the transmitter a testbench was designed (see appendix) that would simulate the process of sending a string of bits at the specified baud rate then checking to see if the transmitter was able to produce the same wave sent in.. The resulting waveform was produce:

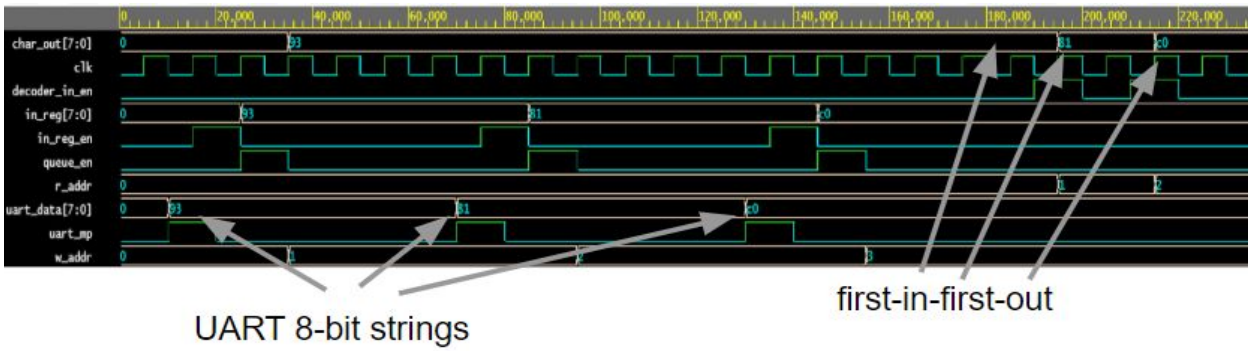


UART Protocol Testing with Oscilloscope

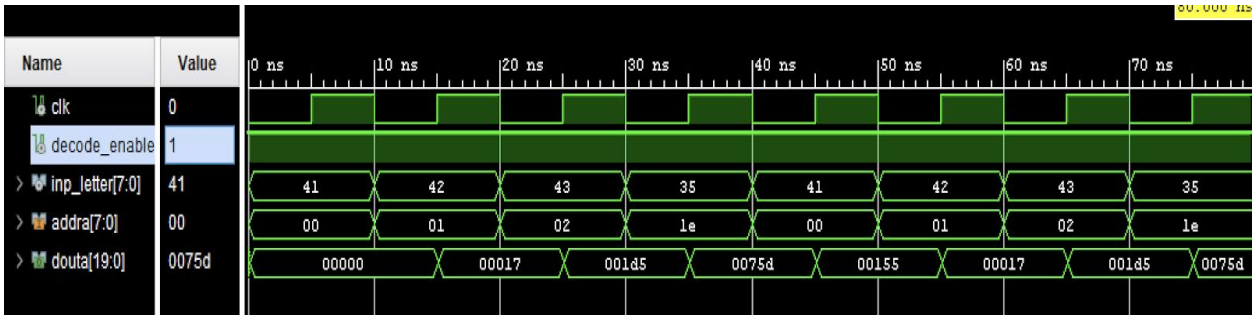
To confirm the UART was working properly, the Analog Digital Discovery 2 (AD2) Network function was used, which allows the user to turn the AD2 into a UART transmitter and receive the data back at a set baud rate (since the systems baud rate is 9600, this baud rate was used for the AD2). Below is the results of this testing.

Queue/Input Logic Testing

When the Decoder is ready for an 8-bit ascii string, it sets its output decoder_in_en high for a monopulse and reads the data on char_out, at which point the queue deletes that entry and moves to the next. The decoder must simultaneously set decoder_in_en high while it reads char_out, as seen in the waveform stimulus designed for this testing. One notable feature of the queue is that its next output is available just after a given input is read from the block, and is discarded just after it is read. The waveform below demonstrates sequential writes followed by sequential reads, which is how the Queue functions in practice, as the output state is entered upon entry of the enter character.



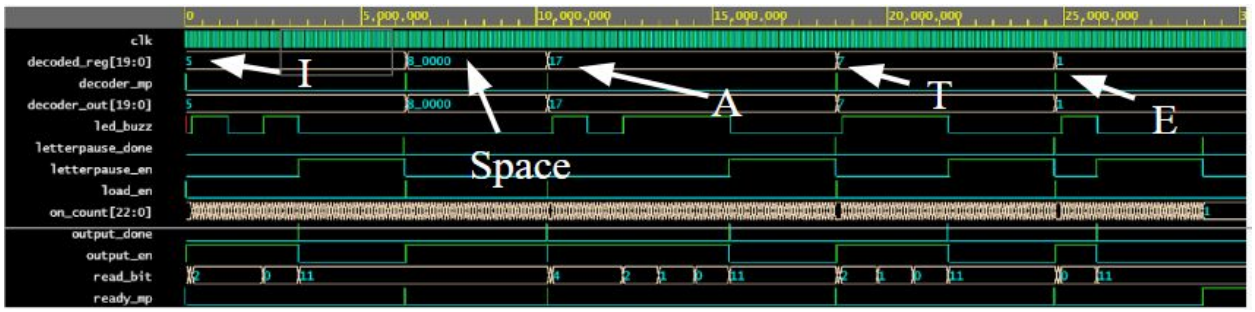
Decoder Testing



In the above waveform, we go through 4 different input characters, “A, B, C, 5” and then repeat the same characters. The output of the Block ROM is on a one clock cycle delay from the inputted address. In this test bench one clock period is 10 nanoseconds but in our real implementation the clock is much slower. The output seen in douta is a 20 bit sequence and is shown in binary coded decimal. The memory is generated from a coe file containing the actual 20 bit sequences.

Timing Block Testing

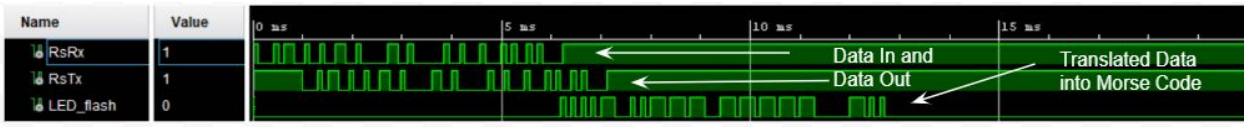
The output logic takes a 20-bit timing string, in this case corresponding to i_ate. When ready_mp goes high, the output logic outputs the morse code signal to the led and buzzer, and then sets ready_mp high, which prompts the decoder to then output the next string and set decoder_mp high. In the waveform below, the timing block is set to output 1000 times faster than normal operation, but demonstrates proper timing of each character’s output.



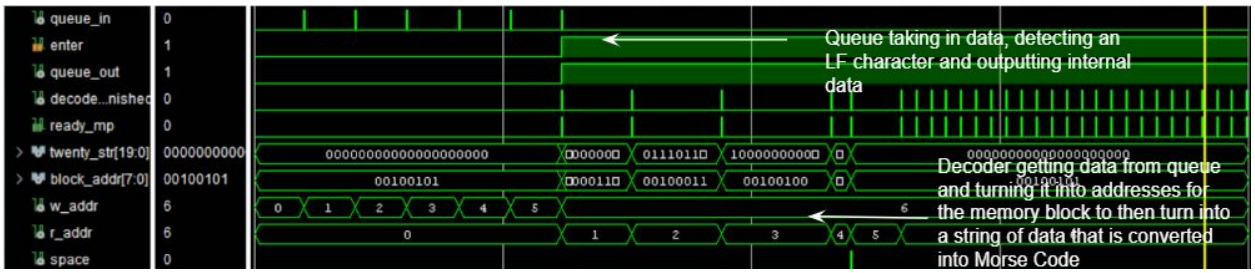
Whole System Testing

Once the whole system was combined together in a top level file, the whole system was tested with a testbench (see appendix) and all the signals were checked. Due to the large scale of this project, the wavegens are split up into the visible outputs and the internal logic.

System Outputs

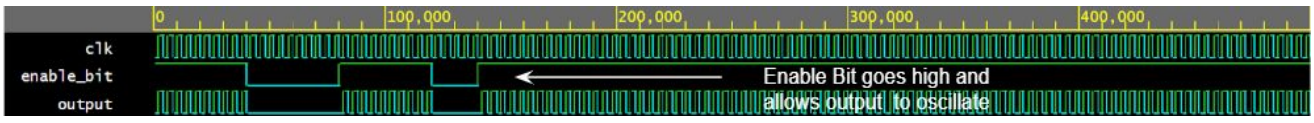


Internal Logic



Oscillator Testing

To clarify that the oscillator was working a testbench was made (see appendix) to simulate the enable bit for the oscillator being enabled and disabled. The below waveform was produced:



Conclusions

Link to Video of Working Design

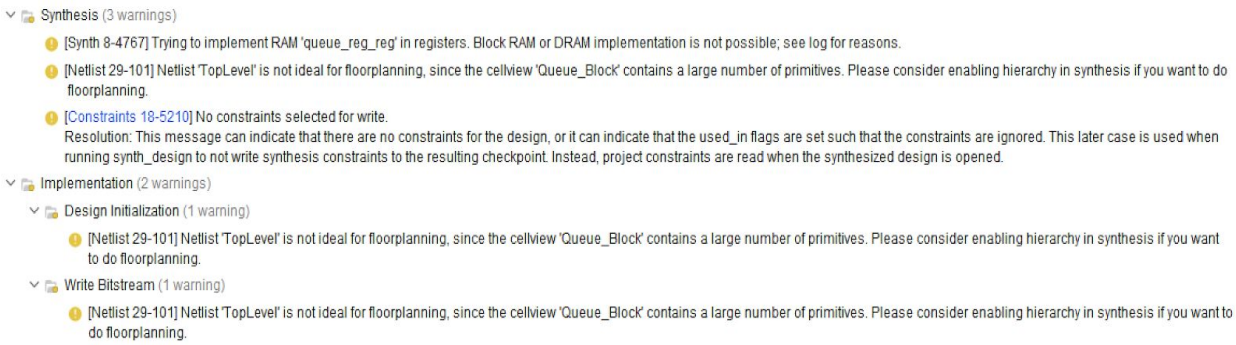
https://drive.google.com/drive/u/4/folders/1tW9BXfVq3xGpy5uyDnqivcXoU5h9u-WH

The above link contains a video of our Basys board outputting the correct output for the input shown in the picture, also available in google drive. The system functions fully with LED, audio, and hexadecimal 7-segment outputs corresponding to the UART input.

Residual Warnings

The functional hardware implementation returned 6 warnings during synthesis, implementation, and bitstream generation. There are three individual warnings, each of no concern:

- Synth 8-4767 - The synthesis step is detecting the queue ‘queue_reg’ and is unable to implement a RAM or DRAM block for this purpose. This is no matter, as it is implemented in registers
- Netlist 29-101 - This warning pertains to ‘floorplanning,’ an option in FPGA design that we are not taking advantage of, as the efficiency of hardware connectivity is not a priority in this project
- Constraints 18-5210 - This warning is of no concern, as no constraints are needed in this case, so the error may safely be ignored



Acknowledgements

We would like to use this space to express our gratitude towards Professor Luke and all those who have helped us in bringing this project to fruition. While continuing to provide guidance and insight throughout the design process, Professor Luke challenged us to tap into our creativity with this Morse Code Translator. We would also like to acknowledge our lab instructor, Ben Dobbins, who gracefully carried us through all of the labs and provided constant guidance for this project. Without him none of this would be possible.

This same sentiment goes for the lovely group of student course facilitators, our LFs and TAs. We would like to thank them for the large number of hours they put in working to ensure that our project was moving along smoothly. An enormous thank you to Hannah Gaven, Matt Gardner, Shailin Shah, Peyton Weber, and Yefri Figueroia.

Throughout the project, all members of our team were present whenever possible so that everyone understood each project decision. Top-level block diagrams, state machines, and component diagrams were designed with everyone's full understanding and awareness. This collaborative effort was instrumental in keeping our group on track to complete the project in a timely manner.

Appendix

Top Level Shell Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;  -- Include all needed Libraries
library UNISIM;     -- needed for the BUFG component
use UNISIM.Vcomponents.ALL;
entity TopLevel is
  Port( clk: in std_logic;
      RsRx: in std_logic;
      LED_flash: out std_logic;
      buzz_out: out std_logic;
      seg :   out std_logic_vector(0 to 6);
      an  :   out std_logic_vector(3 downto 0);
      RsTx: out std_logic );
end TopLevel;
architecture Behavioral of TopLevel is
-- Signals for the 100 MHz to 10 MHz clock divider
constant CLOCK_DIVIDER_VALUE: integer := 5;
signal clkdiv: integer := 0;   -- the clock divider counter
signal clk_en: std_logic := '0';  -- terminal count
signal clk10: std_logic;    -- 10 MHz clock signal
-- Signals for UART processes
signal rx_data : std_logic_vector(7 downto 0);
signal rx_done_tick : std_logic;
signal tx_done_tick: std_logic;
--Signals for Queue block
signal decoder_finished : std_logic  := '0';
signal decoder_in: std_logic_vector(7 downto 0) := (others => '0');
signal not_decoder_finished : std_logic := '0';
--Signals for Decode block
signal block_addr: std_logic_vector(7 downto 0) := (others => '0');
signal ready_mp: std_logic := '0';
--Signals for Memory map
signal twenty_str: std_logic_vector(19 downto 0) := (others => '0');
--Signals for output logic
signal enter_sig : std_logic := '0';
```

```
--Signal for LED
signal LED_flash_sig : std_logic := '0';
--Signal for oscillator
signal osc : std_logic := '0';
signal sound_clkdiv : integer := 0;
signal sound_clk_en :   std_logic;
-- Signals for 7 segment display
signal slot_one, slot_two :   std_logic_vector(7 downto 0) := (others
=> '0');
signal dp, updated_7seg   :   std_logic := '0';
-- Component Declarations
COMPONENT SerialRx  -- Serial Receiver
 PORT(
  Clk : IN std_logic;
  RsRx : IN std_logic;
  rx_data :  out std_logic_vector(7 downto 0);
  rx_done_tick : out std_logic  );
END COMPONENT;
COMPONENT SerialTx
     PORT(  -- Serial Transmitter
     clk : in  STD_LOGIC;  -- 10Mhz clock
     tx_data : in  STD_LOGIC_VECTOR (7 downto 0);  -- data to be
sent
     tx_start : in  STD_LOGIC;  -- start signal
     tx : out  STD_LOGIC; -- data out
     tx_done_tick : out  STD_LOGIC);  -- done signal
END COMPONENT;
Component Output_Logic
     Port (clk : in std_logic;
          decoder_out : in std_logic_vector(19 downto 0); --decoder
has updated decoder_out
          decoder_mp : in std_logic; --tells output logic that
          enter : in std_logic;
          ready_mp : out std_logic; --tells decoder that the output
          led_buzz : out std_logic); --logic is ready for another
signal
end Component;
Component decoder is
     port (clk : in std_logic;
     decode_en : in std_logic;
     letter : in std_logic_vector(7 downto 0);
          done_tick : out std_logic;
          address : out std_logic_vector(7 downto 0));
```

```
end Component;
component blk_mem_gen_0 is
     port(   clka    :   in  std_logic;
          ena :   in  std_logic;
          addra   :   in  std_logic_vector(7 downto 0);
          douta   :   out std_logic_vector(19 downto 0));
end component;
component Queue_Block is
     Port (clk : in std_logic;
          uart_mp : in std_logic;
          uart_data : in std_logic_vector(7 downto 0);
          decoder_in_en : in std_logic;
          enter : out std_logic;
          char_out : out std_logic_vector(7 downto 0));
end component;
component oscillator is
     Port (clk : in std_logic;
          enable_bit : in std_logic;
          output : out std_logic);
end component;
component mux7seg is
     Port ( clk : in  STD_LOGIC;         -- runs on a fast (1 MHz or
so) clock
          y0, y1, y2, y3 : in  STD_LOGIC_VECTOR (7 downto 0); --
digits
          dp_set : in std_logic_vector(3 downto 0);            --
decimal points
          seg : out  STD_LOGIC_VECTOR(0 to 6);       -- segments
(a...g)
          dp : out std_logic;
          an : out  STD_LOGIC_VECTOR (3 downto 0) );      -- anodes
end component;
begin
LED_flash <= LED_flash_sig;
buzz_out <= osc;
-- Clock buffer for 10 MHz clock
-- The BUFG component puts the slow clock onto the FPGA clocking
network
Slow_clock_buffer: BUFG
     port map (I => clk_en,
               O => clk10 );
-- Divide the 100 MHz clock down to 20 MHz, then toggling the
-- clk_en signal at 20 MHz gives a 10 MHz clock with 50% duty cycle
```

```
Clock_divider: process(clk)
begin
 if rising_edge(clk) then
       if clkdiv = CLOCK_DIVIDER_VALUE-1 then
        clk_en <= NOT(clk_en);
   clkdiv <= 0;
  else
   clkdiv <= clkdiv + 1;
  end if;
 end if;
end process Clock_divider;
SOUND_clk_divider:  process(clk10)
begin
     if rising_edge(clk10) then
     if sound_clkdiv = 10000 - 1 then
          sound_clk_en <= NOT(sound_clk_en);
          sound_clkdiv <= 0;
     else
          sound_clkdiv <= sound_clkdiv + 1;
     end if;
     end if;
 end process SOUND_clk_divider;
display_proc:   process(decoder_finished, clk10)
begin
     if rising_edge(clk10) then
     if decoder_finished = '1' then
          slot_one(3 downto 0) <= decoder_in(3 downto 0);
          slot_two(3 downto 0) <= decoder_in(7 downto 4);
     else

     end if;
     end if;

end process display_proc;
--slot_one(3 downto 0) <= decoder_in(3 downto 0) when
decoder_finished = '1';
--slot_two(3 downto 0) <= decoder_in(7 downto 4) when
decoder_finished = '1';
Receiver: SerialRx PORT MAP(
     clk => clk10,  -- receiver is clocked with 10 MHz clock
     RsRx => RsRx,  --
     rx_data => rx_data,  -- Data for Queue
     rx_done_tick => rx_done_tick);
```

```
Trasmitter: SerialTx PORT MAP(  --changed name from UART_Tx
      clk => clk10, -- made Clk upper case c
      tx_data  => rx_data,  -- data to be sent
      tx_start => rx_done_tick,  -- start signal
      tx => RsTx, -- data out
      tx_done_tick => tx_done_tick);  -- done signal

Decoder_map: decoder PORT MAP(
      clk => clk10,
      decode_en => ready_mp,
      letter => decoder_in,
      done_tick => decoder_finished,
      address => block_addr);
Queue_input: Queue_Block PORT MAP(
      clk => clk10,
      uart_mp => rx_done_tick,
      uart_data => rx_data,
      decoder_in_en => decoder_finished,
      enter => enter_sig,
      char_out => decoder_in);
Memory_map: blk_mem_gen_0 PORT MAP (
      clka => clk10, --changed from clk10
      ena => '1', -- comming from wes output block
      addra => block_addr,
      douta => twenty_str);

Output_timer: Output_Logic PORT MAP (
      clk => clk10,
      decoder_out => twenty_str, --decoder has updated decoder_out
      decoder_mp => '1',--decoder_finished,
      enter => enter_sig,
      ready_mp =>  ready_mp, --tells decoder that the output

      led_buzz => LED_flash_sig);--logic is ready for another signal
oscillator_block: oscillator PORT MAP (
      clk => sound_clk_en,
      enable_bit => LED_flash_sig,
      output => osc);

display: mux7seg port map(
           clk => clk10,    -- runs on the 1 MHz clock
            y3 => x"00",
```

```
            y2 => x"10", -- A/D converter output
            y1 => slot_two,
            y0 => slot_one,
            dp_set => "0000",         -- decimal points off
            seg => seg,
            dp => dp,
            an => an );
end Behavioral;
```

Top Level Testbench Code

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY toplevel_tb IS
END toplevel_tb;

ARCHITECTURE behavior OF toplevel_tb IS

component TopLevel is
     Port ( Clk : in  STD_LOGIC;
          RsRx  : in  STD_LOGIC;
     RsTx  : out  STD_LOGIC );
end component;
   --Inputs
   signal clk : std_logic := '0';
   signal RsRx : std_logic := '1';
  --Outputs
   signal RsTx : std_logic := '0';
   -- Clock period definitions
   constant clk_period : time := 10ns;  -- 10 MHz clock

 -- Data definitions
 constant bit_time : time := 104us;  -- 9600 baud

 -- Characters for Transmission
 constant TxDataA : std_logic_vector(7 downto 0) := "01000001"; --
letter A
 constant TxDataB : std_logic_vector(7 downto 0) := "01000010"; --
letter B
 constant TxDataC : std_logic_vector(7 downto 0) := "01000011"; --
letter C
 constant TxDataD : std_logic_vector(7 downto 0) := "01000100"; --
letter D
 constant TxDataE : std_logic_vector(7 downto 0) := "01000101"; --
letter E
 constant TxData2 : std_logic_vector(7 downto 0) := "00110010"; --
Number 2
     constant TxData4 : std_logic_vector(7 downto 0) := "00110100";
-- Number 4
```

```
     constant TxData7 : std_logic_vector(7 downto 0) := "00110111";
-- Number 7
     constant TxData0 : std_logic_vector(7 downto 0) := "00110000";
-- Number 0
 constant TxDataS : std_logic_vector(7 downto 0) := "00100000"; --
Space Char
 constant TxDataL : std_logic_vector(7 downto 0) := "00001010"; --
Cariage Return Char


BEGIN
 -- Instantiate the Unit Under Test (UUT)
   uut: TopLevel PORT MAP (
          clk => clk,
          RsRx => RsRx,
          RsTx => RsTx
     );
   -- 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
  wait for 100 us;
  wait for 10.25*clk_period;

  RsRx <= '0';  -- Start bit
  wait for bit_time;

  for bitcount in 0 to 7 loop
   RsRx <= TxDataA(bitcount);
   wait for bit_time;
  end loop;

  RsRx <= '1';  -- Stop bit
  wait for bit_time; --200 us;
```

```
RsRx <= '0';  -- Start bit
wait for bit_time;

for bitcount in 0 to 7 loop
 RsRx <= TxDataD(bitcount);
 wait for bit_time;
end loop;

RsRx <= '1';  -- Stop bit
wait for bit_time; --200 us;

RsRx <= '0';  -- Start bit
              wait for bit_time;

   for bitcount in 0 to 7 loop
        RsRx <= TxDataS(bitcount);
        wait for bit_time;
   end loop;

        RsRx <= '1';          -- Stop bit
        wait for bit_time;

        RsRx <= '0';  -- Start bit
        wait for bit_time;

        for bitcount in 0 to 7 loop
             RsRx <= TxData2(bitcount);
             wait for bit_time;
        end loop;

        RsRx <= '1';          -- Stop bit
        wait for bit_time; --200 us;

        RsRx <= '0';  -- Start bit
        wait for bit_time;

        for bitcount in 0 to 7 loop
             RsRx <= TxDataD(bitcount);
             wait for bit_time;
        end loop;

        RsRx <= '1';          -- Stop bit
        wait for bit_time;
```

```
                RsRx <= '0';  -- Start bit
                wait for bit_time;

                for bitcount in 0 to 7 loop
                     RsRx <= TxDataL(bitcount);
                     wait for bit_time;
                end loop;

                     RsRx <= '1';          -- Stop bit
                     wait for bit_time;

  wait;
   end process;
END;
```

UART Rx/Tx

All of the UART components were made following the instructions of the provided SPI Lab, written by Eric Hansen. The test bench for the receiver/transmitter were provided and the transmitter was made in collaboration with Matt Gardner. All code for the UART components can be found below.

UART Rx Code

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

entity SerialRx is
Port (      clk : in STD_LOGIC; -- 10MHz master clock
            RsRx : in STD_LOGIC; -- received bit stream
            rx_data : out STD_LOGIC_VECTOR (7 downto 0); -- data byte
            rx_done_tick : out STD_LOGIC ); -- data ready tick
end SerialRx;

ARCHITECTURE behavior of SerialRx is

type state_type is (idle, wait2, wait1, done, reset_b, count_r, baud_eq);
signal current_state, next_state : state_type;
-- Declares states

--Datapath elements
constant BAUD_PERIOD : integer := 1042; --(10 MHz / 9600 = 1042) Round Up
signal baud_count : integer := 0;
signal start_count: std_logic := '0';

--create your other datapath elements here
signal counter: unsigned(3 downto 0) := "0000";
signal shiftreg: std_logic_vector(8 downto 0) := (others => '0');

signal flip1: std_logic := '1';  -- flip flop synchronizer
signal flip2: std_logic := '1';
signal edge_detect : std_logic := '0';
signal count_en: std_logic := '0';
signal reset: std_logic := '0';

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

```
signal take_data: std_logic := '0';
signal Irx_data : STD_LOGIC_VECTOR (7 downto 0) := "00000000";

BEGIN

stateUpdate: process(clk) --Makes state change each clk rise
begin
      if rising_edge(clk) then
            current_state <= next_state;
        end if;
end process stateUpdate;

nextStateLogic: process(current_state, flip2, counter, baud_count) --switch
for states
begin

      case (current_state) is
            when idle =>

             rx_done_tick <= '0';
             shift_flag <= '0';
             take_data <= '0';
             start_count <= '0';
             reset <= '0';
             count_en <= '0';

            if flip2 = '0' and edge_detect = '1' then  -- when we get a
leading bit signal go to next wait state
                  next_state <= wait2;
            else
                next_state <= current_state;
            end if;

            when wait2 =>

            rx_done_tick <= '0';
            shift_flag <= '0';
            take_data <= '0';
            start_count <= '1';  -- Enables baud counter
            reset <= '0';
            count_en <= '0';
```

```
            if baud_count = BAUD_PERIOD/2 - 1 then  -- Don't need to shift
in this bit
                 next_state <= reset_b;              -- Because it's just
our starting zero
            else
                 next_state <= current_state;
            end if;

        when reset_b =>

            rx_done_tick <= '0';
            shift_flag <= '0';
            take_data <= '0';
            start_count <= '0';  -- Resets baud counter
            reset <= '0';
            count_en <= '0';

            next_state <= wait1;

        when wait1 =>
            rx_done_tick <= '0';
            shift_flag <= '0';
            take_data <= '0';
            start_count <= '1';  -- to start baud counter
            reset <= '0';
            count_en <= '0';

            if counter = 8 then -- when you get last bit
                next_state <= count_r;
            elsif baud_count = BAUD_PERIOD - 1 then  -- Shift in data to
register
                next_state <= baud_eq;
            else
                next_state <= wait1;
            end if;

        when baud_eq =>
            rx_done_tick <= '0';
            shift_flag <= '1';
            take_data <= '0';
            start_count <= '0';  -- to reset baud counter
            reset <= '0';
```

```
            count_en <= '1';  -- to enable bit counter

            next_state <= wait1;

        when count_r =>
            rx_done_tick <= '0';
            shift_flag <= '0';
            take_data <= '1';
            start_count <= '0';  -- to reset baud counter
            reset <= '1';
            count_en <= '0';

            next_state <= done;

        when done =>

            rx_done_tick <= '1';
            shift_flag <= '0';
            take_data <= '0';
            start_count <= '0';  -- to reset baud counter
            reset <= '0';
            count_en <= '0';  -- to enable bit counter

            next_state <= idle;

            when others =>
            rx_done_tick <= '0';
            shift_flag <= '0';
            take_data <= '0';
            start_count <= '0';
            reset <= '0';
            count_en <= '0';

                next_state <= idle;  -- Just in case

  end case;

end process nextStateLogic;

end_data: process(clk)
begin
    if rising_edge(clk) then
```

```
        if take_data = '1' then
            Irx_data <= shiftreg(8 downto 1); -- maybe issue
        else
            Irx_data <= Irx_data;
        end if;
    end if;
end process end_data;

rx_data <= Irx_data; -- set the internal data out to output asynchronously

baud_tick: process(clk, start_count)
begin

    if rising_edge(clk) then
        if start_count = '1' then  -- if enabled to baud count
            baud_count <= baud_count + 1;
        else
            baud_count <= 0;
        end if;
    end if;

end process baud_tick;

flip_flop: process(clk)
begin

    if rising_edge(clk) then
      flip1 <= RsRx;  -- flip flop synchronizer to just make sure the data
is accurate
      flip2 <= flip1;
      edge_detect <= flip2;
    end if;

end process flip_flop;

count_proc: process(clk)
begin

    if rising_edge(clk) then
        if count_en = '1' then
            counter <= counter + 1; -- keep track of how many bits you
recieve and shift them in
```

```
        elsif reset = '1' then
            counter <= "0000";
        end if;
    end if;


end process count_proc;

register_proc: process(clk)
begin

    if rising_edge(clk) then  -- To output data
        if shift_flag = '1' then
            shiftreg <= flip2 & shiftreg(8 downto 1);
        end if;
    end if;

end process register_proc;

end behavior;
```

UART Rx Testbench Code

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY SerialRx_tb IS
END SerialRx_tb;

ARCHITECTURE behavior OF SerialRx_tb IS

COMPONENT SerialRx
      PORT(
            Clk : IN std_logic;
            RsRx : IN std_logic;
            rx_data :  out std_logic_vector(7 downto 0);
            rx_done_tick : out std_logic  );
      END COMPONENT;
```

```
   --Inputs
   signal clk : std_logic := '0';
   signal RsRx : std_logic := '1';

      --Outputs
   signal rx_data : std_logic_vector(7 downto 0);
   signal rx_done_tick : std_logic;

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

      -- Data definitions
      constant bit_time : time := 104us;        -- 9600 baud
      --constant bit_time : time := 8.68us;           -- 115,200 baud
      constant TxData : std_logic_vector(7 downto 0) := "01101001";

BEGIN
      -- Instantiate the Unit Under Test (UUT)
   uut: SerialRx PORT MAP (
          clk => clk,
          RsRx => RsRx,
          rx_data => rx_data,
          rx_done_tick => rx_done_tick
        );

   -- 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
            wait for 100 us;
            wait for 10.25*clk_period;

            RsRx <= '0';            -- Start bit
```

```
            wait for bit_time;

            for bitcount in 0 to 7 loop
                  RsRx <= TxData(bitcount);
                  wait for bit_time;
            end loop;

            RsRx <= '1';            -- Stop bit
            wait for 200 us;

            RsRx <= '0';            -- Start bit
            wait for bit_time;

            for bitcount in 0 to 7 loop
                  RsRx <= not( TxData(bitcount) );
                  wait for bit_time;
            end loop;

            RsRx <= '1';            -- Stop bit

            wait;
   end process;
END;
```

UART Tx Code

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

entity SerialTx is
    Port ( clk : in  STD_LOGIC;
           tx_data : in  STD_LOGIC_VECTOR (7 downto 0);
           tx_start : in  STD_LOGIC;
           tx : out  STD_LOGIC;                                    -- to RS-232
interface
           tx_done_tick : out  STD_LOGIC);
end SerialTx;
```

```
ARCHITECTURE behavior of SerialTx is
--Datapath elements

constant BAUD_PERIOD : integer := 1042; --Number of clock cycles needed to
achieve a baud rate of 256,000 given a 100 MHz clock (100 MHz / 256000 =
391)
--adjusted to 9600 baud/10MHz clock -> 1042

--create your other datapath elements here
signal shift_reg : std_logic_vector(9 downto 0) := (others => '1'); --all
zeros w/ active low bus
signal baud_count : unsigned(10 downto 0) := (others => '0'); -- count up
to 1042
signal count : unsigned(3 downto 0) := "0000";
signal baud_tc : std_logic := '0';

BEGIN

--Datapath
datapath : process(clk) --everything should be synchronous in this design.
begin
      if rising_edge(clk) then

        -- create baud counter
        baud_tc <= '0';
        baud_count <= baud_count + 1;
        if baud_count = BAUD_PERIOD then
            baud_tc <= '1';
            baud_count <= (others => '0');
        end if;

        if tx_start = '1' then
            baud_count <= (others => '0');
        end if;

        -- create shift register
        if tx_start = '1' then
            shift_reg <= '1' & tx_data & '0';
        elsif baud_tc = '1' then
            shift_reg <= '1' & shift_reg(9 downto 1);
        end if;
        --right-shift and append a '1' in the MSB
```

```
    end if;
end process datapath;

counter: process (clk)
begin

      if rising_edge(clk) then  -- emission process for tx_done tick
      if tx_start = '1' then
            count <= (others => '0');
        else
            if baud_tc = '1' then
                  count <= count + 1;
            end if;
        end if;

        if count = 9 then
            count <= (others => '0');
            tx_done_tick <= '1';
        else
            tx_done_tick <= '0';
        end if;
    end if;
end process counter;

--hardwire the LSB of the Shift Register to the output Tx.
tx <= shift_reg(0);

end behavior;
```

UART Tx Testbench Code/Whole UART Component Testbench Code

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY toplevel_tb IS
END toplevel_tb;
```

```
ARCHITECTURE behavior OF toplevel_tb IS

component TopLevel is
    Port ( Clk : in  STD_LOGIC;
           RsRx  : in  STD_LOGIC;
                  RsTx  : out  STD_LOGIC );
end component;

   --Inputs
   signal clk : std_logic := '0';
   signal RsRx : std_logic := '1';

      --Outputs
   signal RsTx : std_logic := '0';

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

      -- Data definitions
      constant bit_time : time := 104us;        -- 9600 baud

      -- Characters for Transmission
      constant TxDataA : std_logic_vector(7 downto 0) := "01000001"; --
letter A
      constant TxDataB : std_logic_vector(7 downto 0) := "01000010"; --
letter B
      constant TxDataC : std_logic_vector(7 downto 0) := "01000011"; --
letter C
      constant TxDataD : std_logic_vector(7 downto 0) := "01000100"; --
letter D
      constant TxDataE : std_logic_vector(7 downto 0) := "01000101"; --
letter E
      constant TxData2 : std_logic_vector(7 downto 0) := "00110010"; --
Number 2
    constant TxData4 : std_logic_vector(7 downto 0) := "00110100"; --
Number 4
    constant TxData7 : std_logic_vector(7 downto 0) := "00110111"; --
Number 7
    constant TxData0 : std_logic_vector(7 downto 0) := "00110000"; --
Number 0
      constant TxDataS : std_logic_vector(7 downto 0) := "00100000"; --
Space Char
```

```
      constant TxDataL : std_logic_vector(7 downto 0) := "00001010"; -- LF
Char


BEGIN
      -- Instantiate the Unit Under Test (UUT)
   uut: TopLevel PORT MAP (
          clk => clk,
          RsRx => RsRx,
          RsTx => RsTx
        );

   -- 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
            wait for 100 us;
            wait for 10.25*clk_period;

            RsRx <= '0';            -- Start bit
            wait for bit_time;

            for bitcount in 0 to 7 loop
                  RsRx <= TxData4(bitcount);
                  wait for bit_time;
            end loop;

            RsRx <= '1';            -- Stop bit
            wait for bit_time; --200 us;

            RsRx <= '0';            -- Start bit
            wait for bit_time;

            for bitcount in 0 to 7 loop
```

```
            RsRx <= TxData2(bitcount);
            wait for bit_time;
        end loop;

        RsRx <= '1';            -- Stop bit
        wait for bit_time; --200 us;

        RsRx <= '0';            -- Start bit
            wait for bit_time;

    for bitcount in 0 to 7 loop
        RsRx <= TxData0(bitcount);
        wait for bit_time;
    end loop;

       RsRx <= '1';         -- Stop bit
       wait for bit_time;

       RsRx <= '0';        -- Start bit
       wait for bit_time;

       for bitcount in 0 to 7 loop
           RsRx <= TxDataS(bitcount);
           wait for bit_time;
       end loop;

       RsRx <= '1';         -- Stop bit
       wait for bit_time; --200 us;

       RsRx <= '0';        -- Start bit
       wait for bit_time;

       for bitcount in 0 to 7 loop
           RsRx <= TxDataD(bitcount);
           wait for bit_time;
       end loop;

          RsRx <= '1';         -- Stop bit
          wait for bit_time;

        RsRx <= '0';            -- Start bit
        wait for bit_time;
```

```
        for bitcount in 0 to 7 loop
            RsRx <= TxData2(bitcount);
            wait for bit_time;
        end loop;

           RsRx <= '1';         -- Stop bit
           wait for bit_time;

            RsRx <= '0';         -- Start bit
           wait for bit_time;

           for bitcount in 0 to 7 loop
               RsRx <= TxData7(bitcount);
               wait for bit_time;
           end loop;

              RsRx <= '1';         -- Stop bit
              wait for bit_time;

             RsRx <= '0';        -- Start bit
            wait for bit_time;

            for bitcount in 0 to 7 loop
                RsRx <= TxDataS(bitcount);
                wait for bit_time;
            end loop;

               RsRx <= '1';         -- Stop bit
               wait for bit_time;

              RsRx <= '0';             -- Start bit
              wait for bit_time;

              for bitcount in 0 to 7 loop
                  RsRx <= TxDataL(bitcount);
                  wait for bit_time;
              end loop;

                 RsRx <= '1';         -- Stop bit
                 wait for bit_time;
        wait;
```

```
   end process;
END;
```

Oscillator Component Code

```
-- Code your design here
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

ENTITY oscillator IS
PORT (      clk                     :     in    STD_LOGIC;
            enable_bit : in std_logic;
        output: out std_logic);
end oscillator;


ARCHITECTURE behavioral of oscillator is
begin

  output_logic: process(enable_bit, clk)
  begin
    if enable_bit = '1' then
        output <= clk;
    else
        output <= '0';
    end if;

  end process output_logic;

end behavioral;
```

Oscillator Testbench Code

```
-- Code your testbench here
library IEEE;
use IEEE.std_logic_1164.all;

entity osc_tb is
end osc_tb;

architecture testbench of osc_tb is

component oscillator IS
PORT (      clk                     :     in    STD_LOGIC;
            enable_bit  :     in    STD_LOGIC; --user inputs
        output    :     out   STD_LOGIC --outputs to LEDs after game is
over
        );
end component;



signal clk : std_logic := '0';
signal enable_bit : std_logic := '0';
signal output     : std_logic := '0';


begin

uut : oscillator PORT MAP(clk => clk,
                          enable_bit => enable_bit,
                    output => output);


clk_proc : process
BEGIN

  CLK <= '0';
  wait for 2ns;

  CLK <= '1';
  wait for 2ns;
```

```
END PROCESS clk_proc;

stim_proc : process
begin
      enable_bit <= '1';
    wait for 40ns;

    enable_bit <= '0';
    wait for 40ns;

    enable_bit <= '1';
    wait for 40ns;

    enable_bit <= '0';
    wait for 20ns;

    enable_bit <= '1';
    wait for 20ns;

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

Storing Input (Queue) Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Queue_Block is
      Port (clk : in std_logic;
            uart_mp : in std_logic;
            uart_data : in std_logic_vector(7 downto 0);
            decoder_in_en : in std_logic;
            enter : out std_logic;
            char_out : out std_logic_vector(7 downto 0));
end Queue_block;
architecture Behavioral of Queue_Block is
type state_type is (idle,in_queue,out_queue,load);
signal current_state, next_state : state_type;
signal in_reg_en,queue_in, queue_out,enter_detected : std_logic :=
'0';
signal in_reg : std_logic_vector(7 downto 0) := (others => '0');
signal first_char : unsigned(2 downto 0) := (others => '0');
type regfile is array(0 to 149) of STD_LOGIC_VECTOR(7 downto 0);
signal queue_reg : regfile;
signal w_addr : integer := 0;
signal r_addr : integer := 0;

begin
datapath : process(clk)
begin
      if rising_edge(clk) then
      current_state <= next_state;

      if (in_reg_en = '1') then
       enter_detected <= '0';
       if uart_data = "00001010" then
            enter_detected <= '1';
            in_reg <= uart_data;
           else
        in_reg <= uart_data;
           end if;
      end if;

      if (queue_in = '1') then
```

```
       Queue_reg(w_addr) <= in_reg;
            if w_addr = 149 then
             w_addr <= 0;
        else
             w_addr <= w_addr + 1;
            end if;
      end if;

      if (decoder_in_en = '1') and (r_addr /= w_addr) and (queue_out
= '1') then
           Queue_reg(r_addr) <= (others => '0');
       if r_addr = 149 then
            r_addr <= 0;
        else
                 r_addr <= r_addr + 1; -- r_addr when in this state is
not incrementing.
                end if;
      end if;
      end if;
end process datapath;
char_out <= queue_reg(r_addr) when (decoder_in_en = '1') and (r_addr
/= w_addr) and (queue_out = '1') else
           "00000000";
enter <= enter_detected;
nextStateLogic: process(uart_mp,enter_detected,current_state, r_addr,
w_addr)
begin
next_state <= current_state;

case (current_state) is
      when idle =>    in_reg_en <= '0';
                queue_in <= '0';
                           queue_out <= '0';
                if uart_mp = '1' then
                 next_state <= load;
                           else
                      next_state <= current_state;
                           end if;

      when load =>  next_state <= in_queue;
           in_reg_en <= '1';
               queue_in <= '0';
                           queue_out <= '0';
```

```
      when in_queue => in_reg_en <= '0';
                           queue_out <= '0';
                queue_in <= '1';
        if enter_detected = '1' then
                            next_state <= out_queue;
                           else
                            next_state <= idle;
                           end if;

      when out_queue =>   in_reg_en <= '0';
           queue_in <= '0';
           queue_out <= '1';
                           if (r_addr = w_addr) then
                            next_state <= idle;
                           else
                            next_state <= current_state;
                           end if;

      when others =>       next_state <= idle;
                           in_reg_en <= '1';
                queue_in <= '0';
                           queue_out <= '0';

end case;

end process nextStateLogic;
end Behavioral;
```

Decoder Code

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

entity decoder is
port ( clk : in std_logic;
  decode_en : in std_logic;
  letter : in std_logic_vector(7 downto 0);
     done_tick : out std_logic;
     address : out std_logic_vector(7 downto 0));
end decoder;

architecture behavior of decoder is
-- signals
signal letter_encoding : unsigned(7 downto 0) := (others => '0');
signal decode_en_signal_1 : std_logic := '0';
signal decode_en_signal : std_logic := '0';
signal done_count : std_logic := '0';
signal temp_count, count : std_logic := '0';
begin
letter_encoding <= unsigned(letter(7 downto 0));
path : process( clk)
begin
 --done_tick <= '1';
     --address <= (others => '0');
     --letter_encoding <= unsigned(letter(7 downto 0));
   if rising_edge(clk) then
     --done_tick <= '0';
     --letter_encoding <= unsigned(letter(7 downto 0));
     done_count <= '0';
     temp_count <= '0';
     if decode_en = '1' then --if output logic asking
          done_count <= '1';
          temp_count <= done_count;
     end if;
     end if;

end process path;
address <= std_logic_vector(letter_encoding - 65) when
letter_encoding <= 90 and letter_encoding >= 65 else -- letters
```

```
          std_logic_vector(letter_encoding - 23) when
letter_encoding <= 57 and letter_encoding >= 49 else -- 1 through 9
          std_logic_vector(letter_encoding - 13) when
letter_encoding = 48 else --0
          std_logic_vector(letter_encoding + 4)  when
letter_encoding = 32 else -- space
          std_logic_vector(letter_encoding - 91) when
letter_encoding >= 91 and letter_encoding <= 116 else -- lowercase
          "00100101"; -- Enter character
done_tick <= temp_count xor done_count;
end architecture;
```

Morse Code ROM Coe File

```
; Block ROM, created 27-May-2020 22:29:00
MEMORY_INITIALIZATION_RADIX=2;
MEMORY_INITIALIZATION_VECTOR=
00000000000000010111,
00000000000111010101,
00000000011101011101,
00000000000001110101,
00000000000000000001,
00000000000101011101,
00000000000111011101,
00000000000001010101,
00000000000000000101,
00000001011101110111,
00000000000111010111,
00000000000101110101,
00000000000001110111,
00000000000000011101,
00000000011101110111,
00000000010111011101,
00000001110111010111,
00000000000001011101,
00000000000000010101,
00000000000000000111,
00000000000001010111,
00000000000101010111,
00000000000101110111,
00000000011101010111,
00000001110101110111,
00000000011101110101,
00010111011101110111,
00000101011101110111,
00010101011101110111,
00000000010101010111,
00000000000101010101,
00000000011101010101,
00000001110111010101,
00000111011101110101,
00011101110111011101,
01110111011101110111,
```

```
1000000000000000000,
0000000000000000000;
```

Handle Output (Output Logic) Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Output_Logic is
     Port (clk : in std_logic;
          decoder_out : in std_logic_vector(19 downto 0);
          decoder_mp : in std_logic; --tells output logic that
               --decoder has updated decoder_out
          enter : in std_logic;
          ready_mp : out std_logic; --tells decoder that the output
                 --logic is ready for another signal
          led_buzz : out std_logic);
end Output_Logic;
architecture Behavioral of Output_Logic is
type state_type is (start,idle,load,output,letterpause);
signal current_state, next_state : state_type;
signal read_bit : integer := 18; --initial 18
signal decoded_reg : std_logic_vector(19 downto 0) := (others =>
'0');
signal on_count : unsigned(22 downto 0) := (others => '0');
signal load_en, idle_en, output_en, letterpause_en, start_en,
output_done, letterpause_done, read_ahead, space : std_logic := '0';
signal load_count : unsigned(1 downto 0) := "00";
constant T : integer := 1000000;
begin
datapath : process(clk)
begin
if rising_edge(clk) then
     current_state <= next_state;

     if start_en = '1' then
     led_buzz <= '0';
     else

     end if;

     if load_en = '1' then
      decoded_reg <= decoder_out;
     --led_buzz <= '0';
     load_count <= load_count + 1;
```

```
end if;


if output_en = '1' and output_done = '0' then
load_count <= "00";
if decoded_reg(19) = '1' then
     if (on_count < 4*T) then
          on_count <= on_count + 1;
          led_buzz <= '0';
     else
          on_count <= (others => '0');
          output_done <= '1';
          space <= '1';
     end if;
else
 if decoded_reg(read_bit) = '1' then
      if (on_count < T) and read_ahead = '0' then
           on_count <= on_count + 1;
           led_buzz <= '1';

          elsif read_ahead = '1' then
           if (on_count < T) then
                led_buzz <= '0';
                on_count <= on_count + 1;
           else
                on_count <= (others => '0');
                if read_bit > 1 then
                     read_bit <= read_bit - 2;
               else
                     output_done <= '1';
                     read_bit <= 18;
               end if;
               read_ahead <= '0';
               end if;

      else
           on_count <= (others => '0');
           if read_bit > 0 then
                if decoded_reg(read_bit - 1) = '0' then
                read_ahead <= '1';
               else
                read_bit <= read_bit - 1;
```

```
                    end if;
                    else
                     output_done <= '1';
                    read_bit <= 18;
                    end if;
               end if;
          else
               if read_bit > 0 then
               read_bit <= read_bit - 1;
            else
               output_done <= '1';
               read_bit <= 18;
            end if;
          end if;
     end if;
     else
      output_done <= '0';
     space <= '0';
     end if;


     if letterpause_en = '1' then
      if on_count < 2*T then --wait for 0.3s
           led_buzz <= '0';
           on_count <= on_count + 1;
     else
      on_count <= (others => '0');
   letterpause_done <= '1';
     end if;
     else
      letterpause_done <= '0';
     end if;


end if;
end process datapath;

nextStateLogic: process(enter,current_state, decoder_mp, output_done,
letterpause_done, decoder_out, space, load_count)
begin
next_state <= current_state;

case (current_state) is
```

```
when start =>   load_en <= '0';
     output_en <= '0';
                      letterpause_en <= '0';
                      ready_mp <= '0';
                      start_en <= '1';
                      if enter = '1' then
                           next_state <= idle;
                      end if;

when idle =>    load_en <= '0';
                      idle_en <= '1';
     output_en <= '0';
                      letterpause_en <= '0';
                      start_en <= '0';
                      ready_mp <= '1';
          if decoder_mp = '1' then
           next_state <= load;
           else
                next_state <= current_state;
                      end if;

when load =>  --next_state <= output;
     load_en <= '1';
     idle_en <= '0';
     output_en <= '0';
     start_en <= '0';
                letterpause_en <= '0';
                      ready_mp <= '0';
                      if load_count = 2 then
                           next_state <= output;
                      else
                           next_state <= load;
                      end if;

when output =>  load_en <= '0';
     output_en <= '1';
                      letterpause_en <= '0';
                      ready_mp <= '0';
                      start_en <= '0';

                      if output_done = '1' and space = '0' then
                       next_state <= letterpause;
```

```
                              elsif output_done = '1' and space = '1'
then
                               next_state <= idle;
                              end if;

      when letterpause => load_en <= '0';
           output_en <= '0';
                              letterpause_en <= '1';
                              ready_mp <= '0';
                              start_en <= '0';

                              if letterpause_done = '1' then
                               next_state <= idle;
                              end if;

      when others =>          next_state <= idle;
                              load_en <= '0';
           output_en <= '0';
                              ready_mp <= '0';
                              letterpause_en <= '0';
                              start_en <= '0';

end case;
end process nextStateLogic;
end Behavioral;
```

7 Segment Display Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
entity mux7seg is
     Port ( clk : in  STD_LOGIC;         -- runs on a fast (1 MHz or
so) clock
           y0, y1, y2, y3 : in  STD_LOGIC_VECTOR (7 downto 0); --
digits
           dp_set : in std_logic_vector(3 downto 0);            --
decimal points
           seg : out  STD_LOGIC_VECTOR(0 to 6);       -- segments
(a...g)
           dp : out std_logic;
           an : out  STD_LOGIC_VECTOR (3 downto 0) );      -- anodes
end mux7seg;
architecture Behavioral of mux7seg is
 constant NCLKDIV:   integer := 11;                   -- 1 MHz / 2^18
= 381 Hz
     constant MAXCLKDIV:   integer := 2**NCLKDIV-1;        -- max
count of clock divider
     signal cdcount:      unsigned(NCLKDIV-1 downto 0);   -- clock
divider counter register
     signal CE :          std_logic;                      -- clock
enable
 signal adcount : unsigned(1 downto 0) := "00";       -- anode / mux
selector count
 signal anb: std_logic_vector(3 downto 0);
 signal muxy : std_logic_vector(7 downto 0);          -- mux output
 signal segh : std_logic_vector(0 to 6);        -- segments (high
true)
begin
-- Clock divider sets the rate at which the display hops from one
digit to the next.  A larger value of
-- MAXCLKDIV results in a slower clock-enable (CE)
ClockDivider:
process(clk)
begin
 if rising_edge(clk) then
     if cdcount < MAXCLKDIV then
     CE <= '0';
```

```
   cdcount <= cdcount+1;
     else CE <= '1';
     cdcount <= (others => '0');
     end if;
 end if;
end process ClockDivider;
AnodeDriver:
process(clk, adcount)
begin
 if rising_edge(clk) then
     if CE='1' then
     adcount <= adcount + 1;
     end if;
 end if;

 case adcount is
  when "00" => anb <= "1110";
  when "01" => anb <= "1101";
  when "10" => anb <= "1011";
  when "11" => anb <= "0111";
  when others => anb <= "1111";
 end case;
end process AnodeDriver;
an <= anb;
Multiplexer:
process(adcount, y0, y1, y2, y3, dp_set)
begin
 case adcount is
  when "00" => muxy <= y0; dp <= not(dp_set(0));
  when "01" => muxy <= y1; dp <= not(dp_set(1));
  when "10" => muxy <= y2; dp <= not(dp_set(2));
  when "11" => muxy <= y3; dp <= not(dp_set(3));
  when others => muxy <= x"00"; dp <= '1';
 end case;
end process Multiplexer;
-- Seven segment decoder
with muxy select segh <=
 "1111110" when x"00",  -- active-high definitions
 "0110000" when x"01",
 "1101101" when x"02",
 "1111001" when x"03",
 "0110011" when x"04",
 "1011011" when x"05",
```

```
 "1011111" when x"06",
 "1110000" when x"07",
 "1111111" when x"08",
 "1111011" when x"09",
 "1110111" when x"0a",
 "0011111" when x"0b",
 "1001110" when x"0c",
 "0111101" when x"0d",
 "1001111" when x"0e",
 "1000111" when x"0f",
 "0110111" when x"10",
 "0000000" when others;
seg <= not(segh);    -- Convert to active-low
end Behavioral;
```