Project 4: Extending a Simple Microprocessor – Building an Application

Rigoberto Orozco _____
Anthony Grigore _____

Contents

1 – ABSTRACT

This project required the construction of a functional Universal Asynchronous Receiver/Transmitter (UART) for a microprocessor-based application on the FPGA. The project needed to implement basic I/O processes for a complex system defined in C. Specific framing and timing techniques allowed for the asynchronous system to correctly implement I/O functionality. Furthermore, the system required network communication between gate arrays and between the console and the microprocessor. This system was initially tested by creating a testbench simulated using ModelSim and gtkwave. Synthesis into hardware was verified using SignalTap II. In hardware, the design allowed for both receiving and transmitting ten-bit words to memory. The second part of this project required the implementation of an application designed using the C language. This program allowed for console I/O interaction with the gate array.

2 – INTRODUCTION

The purpose of this assignment was to design, create, and synthesize a UART system to allow network communication between varying environments. This system included a microprocessor capable of receiving and transmitting PIO values through serial shift registers both to and from the system.

Received data initially came in serially as ten bits. This data word contained both start and stop bits as well as a parity bit, allowing for only even parity. Received data was then constructed to be loaded in parallel to memory in the microprocessor.

The microprocessor was also designed to transmit data through a ten-bit bus. This parallel load was then serially sent out within the constructs of the system, allowing for another gate array to receive the data with characteristics previously described.

The C-language application allowed the system to accept serial inputs from the console to be stored into the microprocessor. The functionality of this application also included the ability to read to console data that was serially transmitted from the UART to the CPU.

Many methods and tools were used in designing and implementing this project. Icarus Verilog, ModelSim, and SignalTap II offered a simulated version of the system. This verified the functionality of the design prior to synthesis.

3 - LAB DISCUSSION

3.1 - Design Specification

This project specification included various requirements both for the system operation and network interaction. Serial input assumed a specific format, including a *negative-edge start* and stop bit as well as a parity bit, defining the expected data to be stored. Beyond the scope of the defined serial input, the system maintained unrestricting design specifications within the details

of the system. Serial data in represented the LED bus output from the microprocessor. All data received by the processor needed to be represented by a set of 10-bit LED words. Additionally, the microprocessor needed to receive and transmit data through parallel busses. This defined the high-level system to and from the NIOS II component.

Data received by the system needed to be composed of a previously defined start bit and stop bit as well as a ninth bit *even parity*, which verified the eligibility of the received data. If data received by the microprocessor contained an odd number of true values, the system would send an error message to console representing an unexpected storage to memory. Furthermore, received data needed to be stored within a buffer prior to the parallel load to the NIOS II microprocessor. This buffer allowed the system to maintain a more relaxed timing script, allowing an uninterrupted flow of commands. This allowed the system to function without a strict requirement of timing between back-to-back serial inputs. Although the serial timing of the input required a designated clock, the final storage of the input data, through the buffer, allowed the system to function within a margin of error of ten clock cycles - the primary word needed to be received prior to the secondary word. In other words, the buffer allowed the system to properly pass back-to-back sets of data without required a strict timing script. All transmitted data from the microprocessor followed the previously described design specifications, including the start/stop bit implementation as well as an even parity definition within the parallel bus from the processor. All transmitted data was in *little endian* form.

The C program needed to give the microprocessor instructions on how to interface with the hardware. Specifically, the process of sending and receiving bytes. Another task the C program had to take care of was implementing a parity check by adding a parity bit to each byte sent, as well and checking a parity bit each time a message was received.

3.2 - Design Procedure

Initially, development of this project began with the creation of the system block diagram. The given definition of the serial input, including the specified start, stop, and parity bits, helped initiate construction of the overall design. The initial step in designing this project was creating a microprocessor with specified 10-bit GPIO busses. The primary goal was to construct a system capable of receiving this serial input and parallel-loading a 10-bit bus into the microprocessor.

The procedures for constructing the microprocessor were described in detail in tutorials provided on the class webpage. The first microprocessor required the construction of different components including a clock, a memory, a computer, and Parallel I/O. After specifying the target FPGA, the clock was constructed. The clock utilized the 50 MHz frequency provided by the gate array. This component was later connected to all other components within the microprocessor, effectively creating a synchronous environment. The memory stored in the microprocessor was specified to 20,480 bytes. This was mainly due to the large size of the NIOS platform. The Parallel I/O

components were constructed to define what the inputs and outputs to the processor would be and what size these variables were. In order for the system to effectively communicate with itself, the microprocessor needed to be comprised of a ten-bit parallel input bus as well as a ten-bit parallel output bus. Both of these GPIO variables are later described within the specifications of the hardware. The processor included a single bit reset represented by KEY [0] on the gate array.

Once the microprocessor was implemented in software using the Qsys tool within Quartus, the next step was to construct the hardware capable of receiving serial data from an arbitrary source. The serial data was expected to follow the formats described within Section 3.1: Design Specification. The serial data was defined as an input to the shift register. Values coming in to the system from an arbitrary source were stored in the shift register and parallel-loaded into a ten-bit buffer. This buffer then loaded the stored, ten-bit value through a parallel bus to the microprocessor.

The microprocessor was also required to transmit a ten-bit value through a parallel bus. This value needed to follow the specifications of the design, including an initial start bit, a stop bit, and a parity bit within the data. Once data was loaded into the parallel shift register, the data was sent out to an arbitrary source one bit at a time in little endian format. Serial counters allowed the system to run continuously while adhering to the expected length of all transmitted values.

The design of the hardware helped define the function of the C software. The C software commanded the hardware. This software sent values to the microprocessor on the gate array to be stored in memory. The console was also capable of receiving data sent from the microprocessor. The C program had to be designed to interface with the hardware. To start the program was given access to the inputs and outputs of the NIOS processor. With access to IO, the program was then configured to send and receive based on the inputs and user input at the console. When writing the program the thing to be addressed was the console input. After taking in the console inputs a method to add parity bits to each byte of the input was designed. After successfully sending bytes a method of receiving and checking parity was made. Finally a method of displaying the received message was made.

3.3 - System Description

The overall system was comprised of two different functions; the system needed to receive serial data and transmit serial data. The system in charge of receiving data allowed a serial input to the system and later stored the value of the transmitted word into memory within the microprocessor. The I/O to this half of the system included a serial input and a parallel output. The serial input needed to follow a strict format. Between transmitted words the serial input needed to hold a true value, only shifting in true bits to the shift register. The start of a transmitted word was represented by a low value. Once the serial input had a negative edge, the

system began storing values from the serial input. The transmitted word needed to be contained within ten bits. Nothing higher than ten bits was stored in the register and anything lower than ten bits contained false data. Once ten bits were stored in the register, a buffer parallel loaded the transmitted data into the microprocessor.

The system in charge of transmitting data parallel-loaded values from memory into a shift register and then serially transmitted the data through the shift register. Following the strict format of the serial output, this half of the system needed to properly encase the data within the required specifications. A start bit needed to be stored in the shift register. Also, a counter needed to ensure that the serial data being transmitted from the microprocessor only contained ten bits, with true values being transmitted prior to and after the word.

The gate array functioned at a higher clock frequency than was needed to store the received data. The gate array used the built-in 50MHz clock while the serial input was transmitted at a rate of roughly 9600 Hz. Therefore, the system receiving data needed to run at a clock cycle that was significantly slower than the built-in frequency. Also, the system transmitting data needed to send out bits at roughly 9600 Hz. This timing restraint required the construction of counters that allowed the shift register to only load data at a rate that adhered to the serial input. Serial data was recorded on the positive edge of this slower clock rate. This clock needed to record the data at the center of the transferred bit. That is, the clock cycle that represented the storage of the bits needed to fall at the center of the input signal. By constructing a timer that only recorded values within the center of the serial input bit, the system allowed for a larger margin for error. This strengthened the reliability of the system.

The C program handles when the processor receives or sends data. The console input is taken in and sent out byte/character at a time. The program also implements an even parity check to verify if a character received is corrupted. If a character received is corrupt it will be shown as a "~" on the receiving end. The message sent is capped at a maximum of 160 characters.

3.4 - Software Implementation

The overall system required the transmission and reception of data. Below are the separate system functions described in detail. Overall system functionality is later analyzed.

Receiving Data:

Initially, data was received through a serial input to the shift register. This data held a clock cycle of roughly 9600 Hz. A new clock needed to be implemented within the system in order to efficiently store this data without missing a bit or storing the same bit more than once. In order to create a clock of about 9600 Hz, the built-in clock of 50 MHz needed to pass through a system of equations within the modules. Initially, the 50 MHz clock was inputted into a "divideClk"

module. This module inputted the 50MHz clock and outputted a clock that was roughly 153 kHz. The 50 MHz clock was divided by five to be a value of 10 MHz. From there, a clock divider was implemented to cut this value down to 153 MHz. The gate logic used for this module can be seen in Figure 3.4.1 below.

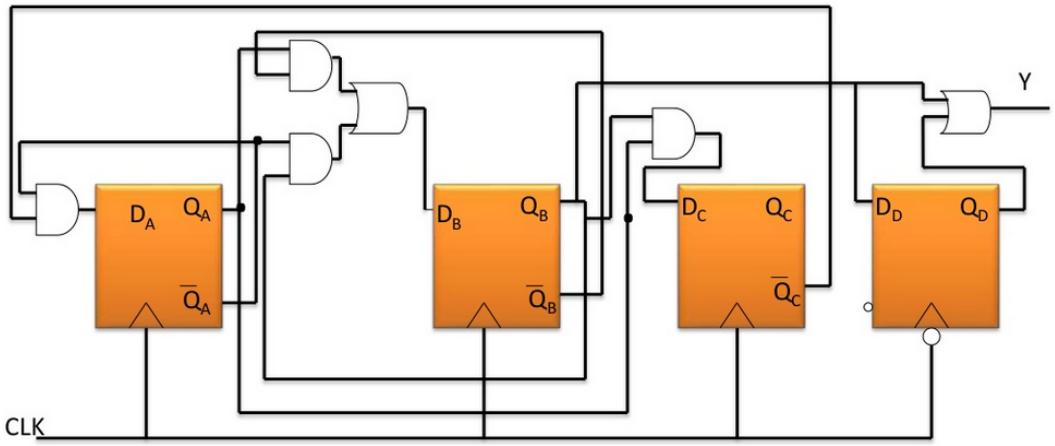

Figure 3.4.1 50MHz Clock Division by Five

The I/O definitions of the clock divider can be seen below in Figure 3.4.2.

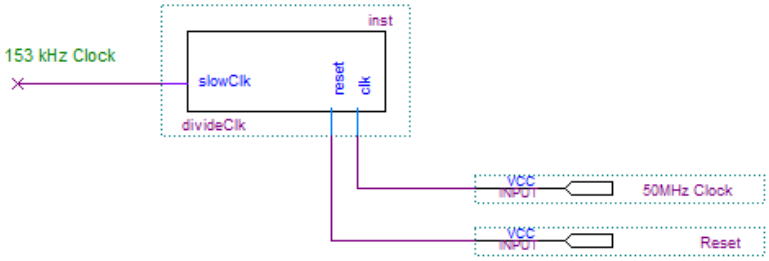

Figure 3.4.2 divideClk Module I/O

Once the clock was divided to be roughly 153 kHz, another module cut this value down to roughly 9600 Hz. This module took the 153 kHz clock input and divided it into 16 clock cycles at a time through a counter. From there, once the counter hit eight, the output clock inverted its value. This mechanism effectively cut the 153 kHz clock into a 9600 Hz frequency while also centering the clock to the serial input bits. This mechanism reduced the margin of error for the system. This is because clocking the system to read the center of an input bit results in a much higher rate of success rather than trying to record bit values at the beginning of the bit. The 9600

Hz output signal was used in the counter module described next. The “SixteenCounter” module is described in Figure 3.4.3 below.

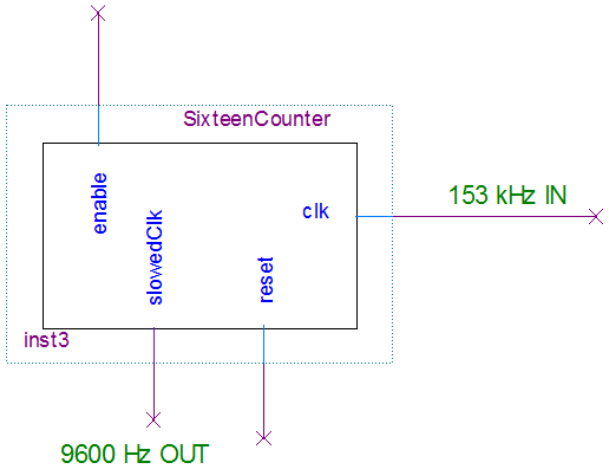

Figure 3.4.3 SixteenCounter Module I/O

Now that the system maintained a clock frequency that matched that of the input, serial data could be successfully recorded without glitches, recurrences, or absences within the shift register. Therefore, the next step was to efficiently store serial data received by the system.

The serial data was an input to the start-bit detector as well as the parallel shift register. Once the system recognized a low value for the start bit from the “StartBitDetector” module, a counter was initialized to keep track of the data bits being stored to the register. The initialization of the counter needed to be reduced to the 9600 Hz clock rate because the counter module ran at the slower frequency. This ensured that the beginning of a transmitted word would not be missed, since the longer clock period would be more efficient at catching the initialization of the counter. In order to maintain a systematic line of communication, a “doneDetector” module reduced the frequency of the previously defined “done” signal coming from the StartBitDetector. From there the “SampleCounter” module allowed the system to follow how many clock cycles were necessary to fully read and store the input word. The end of the word was represented by the end of the counter, where a “done” signal was sent from the counter. The I/O interaction between the “StartBitDetector”, the “doneDetector”, and the “SampleCounter” can be seen below in Figure 3.4.4.

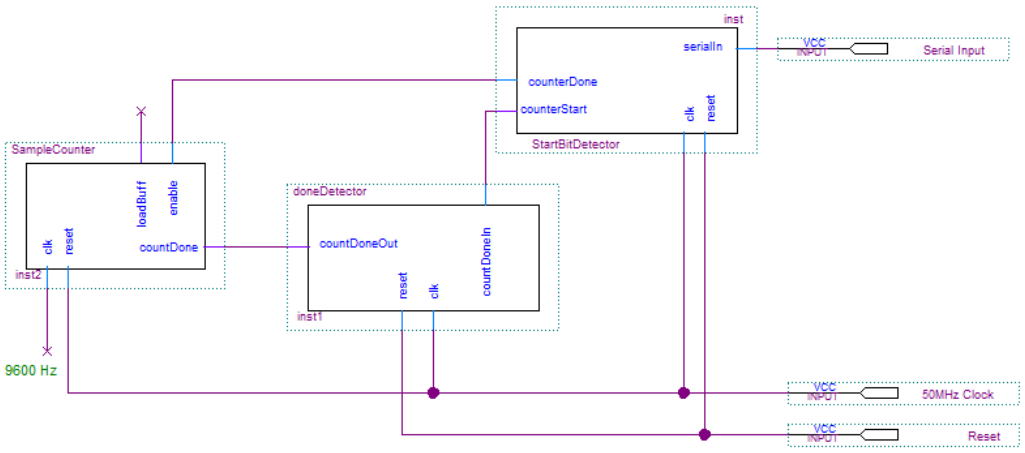

Figure 3.4.4 Tracking the Serial Input Received

While the SampleCounter incremented from zero to nine, the shift register shifted in values from the serial input one clock cycle at a time. Again, the serial input into the shift register was recorded using the 9600 Hz clock frequency that was timed to be centered at each serial bit. These bits were stored into the lowest bit, effectively shifting out the highest bit. The counter allowed the shift register to continuously shift, no matter what value was being stored. Once the counter reached nine, a “done” signal was sent to a buffer. Again, the “done” signal from the counter needed to be passed through the “doneDetector” module to allow the signal to be reduced in frequency. From there the buffer parallel-loaded the current value held in the shift register into a new register within the buffer. The I/O system between the shift register and the buffer can be seen below in Figure 3.4.5.

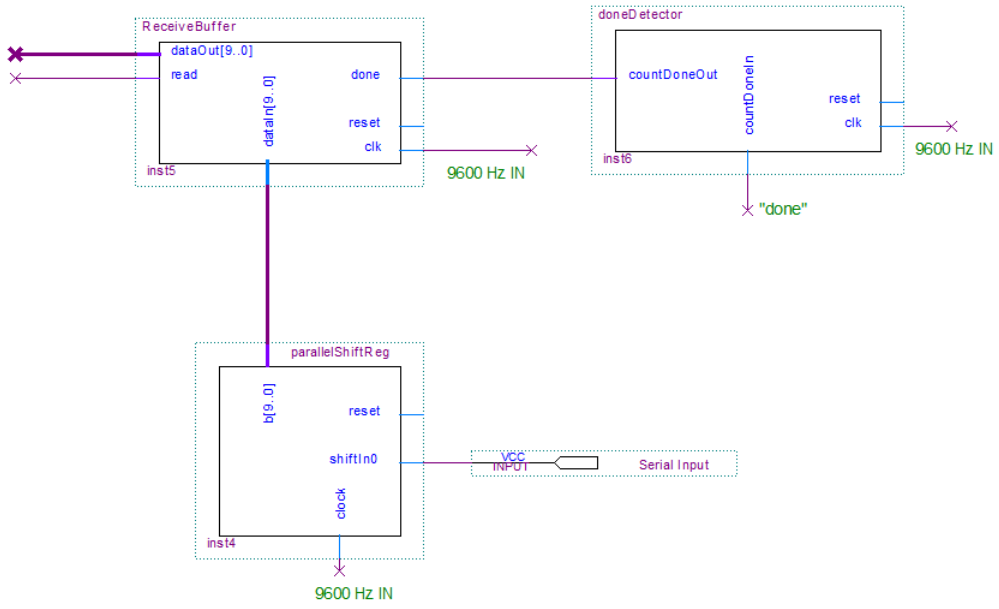

Figure 3.4.5 Serial Input Shift Register and Buffer

The conclusion of the counter allowed the parallel-load buffer to store the values seen in the shift register. This buffer allowed the system to continuously shift through new words while still maintaining the value of the previous word. This permitted the buffer system to run on a less-strict timing construct. While the shift register loaded in new data, the buffer was given ten full clock cycles to load data into the microprocessor. Once the software in the microprocessor was ready to receive data, the processor inputted the “read” signal from the buffer. If “read” was true, signifying the buffer currently held a word in its register, the parallel bus out of the buffer sent data to the processor to be stored. The construction of this safety net allowed for an easier transition between back-to-back serial loads, giving the system more time to process the serial input.

Back-to-back serial inputs required the implementation of a built-in system reset within the receiving construct of the design. The system needed to reset itself every time a word finished processing. Therefore, the “done” signal from the counter signified a rest to the system. Once the counter reached its max value and outputted a “done” signal, the StartBitDetector was reset to begin searching for a new start bit. Furthermore, the clock divider designed to center the clock cycles was reset once the counter was finished. This allowed the system to reconfigure the 9600 Hz clock to again be centered at the serial input bits.

Transmitting Data:

The project also required the system to transmit data with similar requirements for the serial data outputted from the system. Data transmitted needed to be outputted at a clock cycle of roughly 9600 Hz and needed to include a start bit, stop bit, and an even parity bit. Data also needed to be sent in little endian format.

To begin, the transmit system needed to parallel load data from the microprocessor into a buffer once the processor was ready to send data. Once data was loaded into the buffer, a signal was sent to a module called “ShiftRegControl” which regulated the shift register. Once the shift register control module received a “start” signal, the control told the shift register to parallel load the data stored in the buffer into the shift register. From there, the control commanded the shift register to output one bit at a time and began counting. With each clock cycle the register shifted out the lowest bit, sending this bit as the serial transmission from the system. The control module conducting the shifting of the register and commanded the register to hold its value once the word was completely transmitted. Once the transmission was finished, a “done” signal was sent to the processor and the shift register maintained a reset high, consistently sending true values through the serial output. Another “SixteenCounter” module was necessary in this system to allow the shift register and control module to run at 9600 Hz. This effectively sent the transmitted signal at a rate of 9600 Hz to be read by another arbitrary, receiving system. Another “doneDetector” module was necessary in order to reduce the timing of the “done” signal. The microprocessor ran at 50 MHz and required the “done” signal to be sent at that frequency. Therefore, a “done” signal sent at 9600 Hz needed to be reduced to a 50 MHz signal through the doneDetector. The full schematic of the transmitting system can be seen below in Figure 3.4.6.

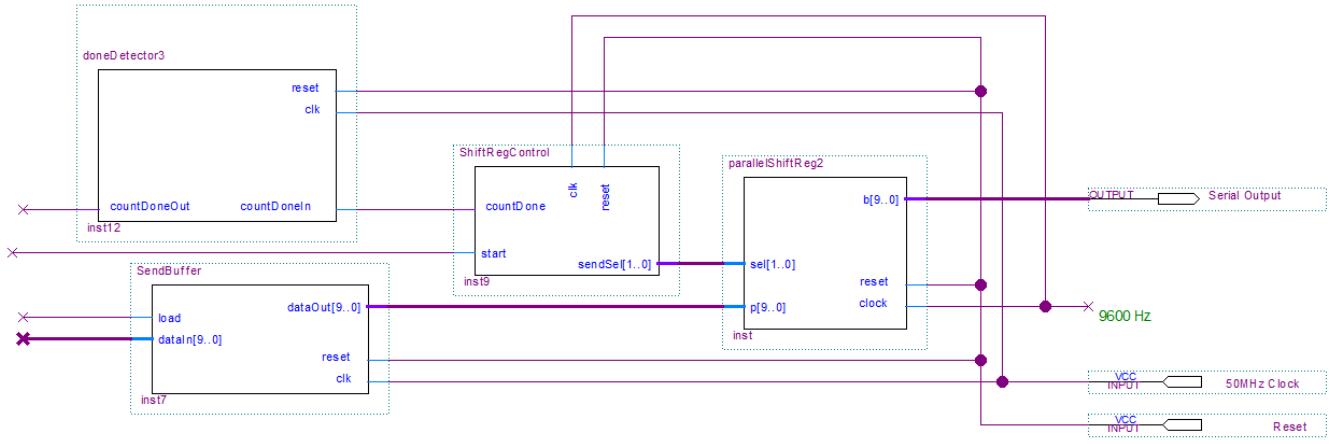

Figure 3.4.6 Transmitting Hardware

The C program prints “Hello from Nios II!” to the console to indicate the start of the program. The program then prompts the user to input a message of up to 160 characters. The message is stored in a character array then sent out one character at a time. When a character is sent the data_out port of the processor gets assigned to the value of the byte. Then a load signal is sent to the buffer to take in the byte. Once the buffer is loaded a start signal is sent telling the hardware to start shifting out the byte. Each time a character is sent an even parity bit is assigned to the byte being sent. Once the character has been sent the program immediately checks for an incoming character. An incoming character is found when the char_received input goes true this signals the processor to record a new byte. The characters’ parity bits are checked to be even to ensure that there was no errors in transmission. If a character is found to have an even parity the parity bit is removed and the character is saved in a character array. If the character received is corrupt it will show up as a “~” and stored in the character array.

3.5 - Hardware Implementation

The hardware implementation followed the basic outline described by the software design. The software design structured each module and allowed for the construction of all block diagrams analyzed in Section 3.4. Below is the overall block diagrams of the system. This diagram is the outline of the IO relationships of the top-level architecture. This diagram includes a simplified “NIOS II” block that signifies the NIOS II microprocessor. By implementing a NIOS II microprocessor, the receiving and transmitting systems described above were combined to represent the overall design of the project. The microprocessor needed to have a ten-bit parallel input to receive data as well as a ten-bit parallel output to transmit data. All functions of the microprocessor were defined by the hardware of each respective system. When receiving, a “read” signal was sent to the microprocessor telling the processor that data was ready to be stored from a buffer. When transmitting, a “done” signal was sent to the processor informing the processor that the system has finished transmitting the data sent from the microprocessor. The system itself has three inputs and one output. The inputs include the built-in, 50 MHz clock from the gate array, a hard reset represented by KEY [0], and the serial input obtained with specific construction requirements. The only output to the system is the transmitted, serial data that follows the same constructs as the input data. All together, the schematic for the system can be seen in Figure 3.4.7, including the receiving system, the transmitting system, and the NIOS II microprocessor.

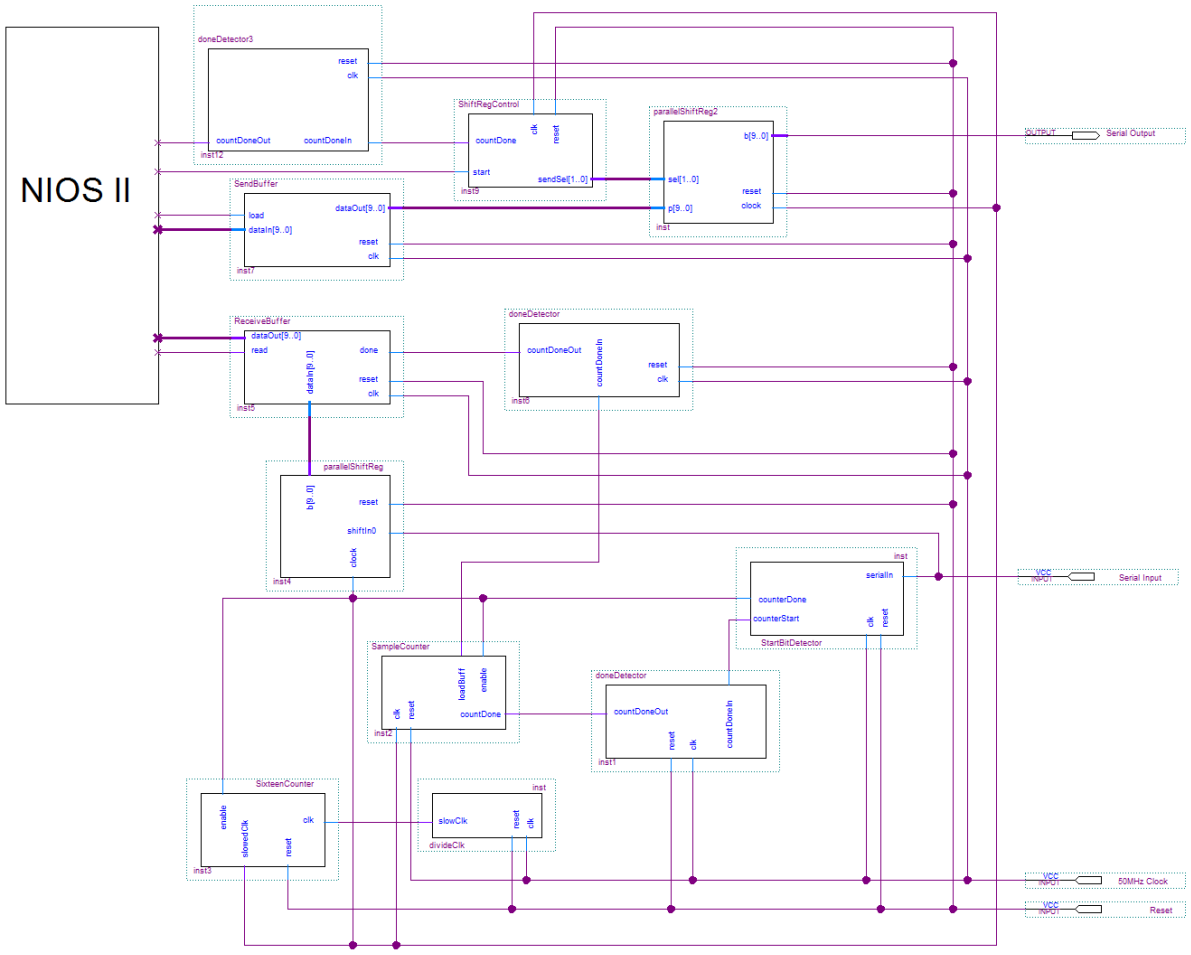

Figure 3.4.7 Universal Asynchronous Receive/Transmit Schematic

4 - TEST PLAN

Testing of every major component needed to be executed in order to ensure the system works according to original requirements. Initially, the clock needed to be tested to ensure it was functioning correctly as this allowed the sequential logic to properly execute the design requirements. Also, KEY [0] needed to be tested to ensure that the system fully reset, including clearing all flip flops. Next, the 9600 Hz clock construct needed to be tested to ensure that it was functioning at the proper speed and recording data within the center of the serial input bits. This testing included the divideClk module, verifying that the output was roughly 153 kHz, and the SixteenCounter module, verifying that the clock cycles were properly centered.

The testing of the receiving system began with the StartBitDetector. The StartBitDetector needed to be tested to ensure that the system acknowledged the reception of a new word. Once the start

bit was properly detected, the SampleCounter needed to keep track of how many bits have been stored to the register before notifying the ReceiveBuffer module that it needed to store the current value held in the shift register. The shift register needed to be tested to verify that it was constantly shifting in new values in little endian format. Once the ReceiveBuffer correctly held the received word in a register, the microprocessor needed to be tested to ensure that the word was properly received into the memory chip within the processor.

The testing of the transmitting system began with the communication between the microprocessor and the SendBuffer module. The SendBuffer needed to grab the value held in the memory of the processor through a parallel bus once the processor signaled the buffer to load in a new value. Loading in the new value needed to initiate the ShiftRegControl module through the simple DoneDetector module previously explained. The ShiftRegControl module needed to be tested to ensure that the commands given to the transmitting shift register followed the proper sequence. More information about this sequence can be seen in Section 5: Test Specification. Once the parallel load from the SendBuffer reached the shift register, the shift register needed to be tested to ensure that the proper values were being serially transmitted out of the system in a format that matched that expected by the receiving system.

The C program needed to be tested to ensure that the microprocessor correctly received data from the ReceiveBuffer and correctly transmitted data to the SendBuffer. This functionality was seen by testing the overall system. Compilation of the C program confirmed proper syntax and the correctly executed transmit/receive functions were seen through outputs on the gate array and synthesis to Signal Tap II.

5 - TEST SPECIFICATION

As previously stated, the first test needed to ensure the input clock functioned correctly. Tested needed to confirm that the clock alternated high/low at a consistent rate and adhered to the divisional cuts in frequency if necessary. Also, KEY [0] needed to be tested to ensure a functional, hard reset. Initially composed of 50 MHz cycles, the 9600 Hz clock-dividing system needed to be tested to ensure that the clock cycles alternated values at a frequency close to 9600 Hz and that the positive edges of these cycles were offset to the original serial input rate by one fourth the period, essentially centering the 9600 Hz clock to the middle of the serial input bits to be read. This construct was necessary because centering the positive-edge reading of the serial input allowed the system a larger margin of error between serial bits. This clock division was a result of communication between two modules, the divideClk module and the SixteenCounter module. The divideClk module inputted the 50 MHz clock and divided the frequency to 10 MHz using the logic seen in Figure 3.4.1. From there a simple clock divider was used to cut 10 MHz down to 153 kHz. This output needed to be tested to ensure the SixteenCounter was manipulating a clock input of 153 kHz to accurately result in a 9600 Hz frequency. The SixteenCounter inputted the 153 kHz clock from the divideClk module. This counter needed to

be tested to ensure that the output of the module resulted in a 9600 Hz clock signal that was offset by one fourth the period of the serial input clock.

The StartBitDetector in the receiving system needed to run at 50 MHz and output a true “start” value once the module saw a negative edge on the serial input, denoting a “start” bit. Once the start bit was detected and the system was notified, the StartBitDetector needed to be tested to ensure that the module was held in a hold state while the receiving system processed the new word. Once the word was fully processed, the StartBitDetector needed to receive a “done” signal from the system. From there, the detector needed to reset itself and begin searching for a start bit all over again. This process allowed back-to-back serial transmission into the system. Once the start bit was detected, the SampleCounter needed to be tested to ensure that the counter correctly incremented from zero to nine, representing the amount of cycles it took to fully load the serial word into the parallel shift register. The parallel shift register needed to be tested to confirm values were continually shifted in at 9600 Hz from the highest bit. This stored the word in little endian format. Once the counter reached nine, a signal from the counter to the buffer needed to be tested that represented the conclusion of the counter. Comparison between the counter state and the bit value of this signal needed to be compared in simulation to confirm that the value stored into the buffer was indeed the serial input word initially seen by the system. Any contingencies of an incorrect bus needed to be resolved within the state machine of the counter. Furthermore, the signal informing the buffer to parallel load the values of the shift register need to be tested to validate the period of this signal lasted one single clock cycle. A signal lasting more than a clock cycle at 50 MHz would result in the parallel loading of an incorrect word into the buffer. This is because, since the shift register constantly shifted in new values, an addition clock cycle introduced the possibility of an invalid bit being shifted in to the register. From there, communication between the processor and the buffer needed to be tested. Once the buffer received the word in a parallel input bus, a signal from the buffer to the processor needed to be sent out to inform the processor that there is a word ready to be received inside the register of the buffer. This value needed to be held true until the processor completely received the word into memory. This is because the software within the processor needed to be given enough time to recognize the “load” signal. A single clock cycle at 50 MHz was too fast of an input rate for the processor to recognize the load signal. Once the processor successfully loaded the word from the buffer, a “done” signal needed to be sent to the buffer to allow the buffer to accept a new word into the parallel register.

The SendBuffer in the transmitting system needed to be tested to confirm it ran at a 50 MHz frequency, alongside the microprocessor. If the buffer received a “load” signal from the processor, the buffer needed to be tested to verify that the value stored in the processor was loaded through a parallel bus into the buffer. The buffer had a parallel output to the shift register that represented the word transmitted out of the processor. This signal needed to be tested to

match the value sent from the processor. The shift register was controlled by the ShiftRegControl module. This module needed to get the same “load”/”start” signal that the SendBuffer received from the processor. This signal needed to be tested to align with the buffer. Once the control module received the “start” signal, representing the transmission of a word, the control module needed to send a two-bit signal to the shift register in a specific sequence. Initially, the control needed to send the shift register 2’b11, denoting a “load” control. Here the shift register loaded in the value from the buffer. Next, the control module needed to send the shift register 2’b01, denoting a “shift left” control. The shift register then shifted out the values in the DFF’s in little endian format. Once the state machine concluded shifting in the control module, the shift register was sent 2’b00 to allow the register to hold and shift out true until another word was transmitted. All of these signals in this sequence needed to be tested and verified to have a functional output and correct timing. These modules were controlled by the 9600 Hz clock to allow the serial output being transmitted to run at 9600 Hz.

When testing the C program all the stages needed to be verified to move on. First the user input needed to be taken in correctly. The characters in the character array needed to be checked to be correct. Certain functions that were created in the C program needed to be verified in order to continue with the design process. The way the program communicated with the hardware needed to be verified. The outputs needed to communicate successfully with the hardware in order to send a character. The inputs to the processor were then tested and used to get the receiving part of the C program working properly. The C program was then tested to correctly send and receive characters. The C program worked as it was specified.

6 - TEST CASES

The test set up included creating testbenches for each module, declaring input values for each module, verifying the output values through simulation, and altering the designs based on output values, if necessary. Each module contained inputs and outputs that were necessary to test under simulation. These IOs are described in detail in Section 5. The testbenches constructed for each module tested expected inputs, timing and variable constraints, and “don’t care” cases. Testbenches constructed all implemented common code block that created a simulated clock. This code is reconstructed below.

```
// Set up the clock.
parameter CLOCK_PERIOD=100;
initial clk=1;
        always begin
                #(CLOCK_PERIOD/2);
```

```
        clk = ~clk;
    end
```

Once the clock was constructed in the testbenches, an initial block instantiated the step-by-step simulation of the system. Within the initial block, inputs were assigned values and given @ (posedge clk) commands to increment the clock. One of the initial commands within this block included assigning reset a false value, effectively resetting the system and kick starting the software. This block was also where the varying IO cases were implemented, such as expected inputs, timing restraints, and variable restraints (e.g. testing that the division operator does not correctly compute the value if the divisor exceeded the dividend).

To verify that the designs met the specifications, the test case needed to measure the output signals. The expected output signals, as previously described, were analyzed in simulation based on the testbenches. Simulation and construction of gtkwave files allowed for visual interpretations of the behaviors of the modules through Icarus Verilog. These waveforms can be seen in the Appendix. SignalTap II analyzed the hardware implementation of the design by specifying input/output values measure on the FPGA. These values were measured to match what was expected of the system

7 - ANALYSIS OF RESULTS

Overall, the results of this project were successful. There were many components to this assignment. To begin, the receiving system of the UART successfully received data transmitted serially as an input. The StartBitDetector correctly identified the beginning of the serial data by noting the negative edge of the input. The waveform of this simulation can be seen in Figure 11.3.1 of the Appendix. Once the start bit was detected, the SampleCounter correctly managed the input bits to the shift register, maintaining a systematic approach to concluding the serial reception. Once the counter incremented from zero to nine, the shift register held the correct input word and the buffer was commanded to store the register value through a parallel bus. The waveform of the SampleCounter simulation can be seen in Figure 11.3.2.The successful identification of every input bit would not have been possible without a function clock divider that created the 9600 Hz frequency from the 50 MHz built-in clock. The dual-module system in charge of dividing the clock to 9600 Hz was also simulated. The clock needed to reset itself once the word was completely received. This allowed the clock to again fall at the center of the set of bits (in the case of back-to-back words). This waveform can be seen in Figure 11.3.3. Furthermore, the ReceiveBuffer was required to grab the ten-bit value inside the shift register once the counter reached its max value of nine. This allowed the microprocessor to receive this input word directly from the buffer, strengthening the system as a whole. When the microprocessor received data from the buffer, the output LEDR [8:1] successfully showed the word to be stored. The waveform of this module can be seen in Figure 11.3.4 of the Appendix.

These results and their respective waveforms proved the receiving end of the design worked successfully.

Next, the UART successfully transmitted data serially out of the microprocessor. In order for the microprocessor to load a parallel bus into the SendBuffer module, the processor needed to know that the buffer was currently empty. Therefore, the “done” signal sent from the ShiftRegControl module, through the DoneDetector needed to successfully reach the processor as an input. This effectively told the processor that transmission is permitted. This functioned properly and the waveform of this simulation can be seen in Figure 11.3.5 in the Appendix. Once the microprocessor saw that the buffer was empty and able to load a new word, the processor then sent out the word from storage into the SendBuffer. Along with the parallel bus, the SendBuffer needed to accept a “load” signal into the system, enabling transmission into the buffer’s register. The SendBuffer successfully loaded the word from the processor once the system allowed a transmission. The success of this function of the system can be seen in Figure 11.3.6 of the Appendix. The ShiftRegControl module was in charge of managing the input bits to the shift register, maintaining a systematic approach to concluding the serial transmission. When the data was loaded into the SendBuffer, a “start” signal was also sent to the ShiftRegControl module. From there the control module successfully loaded the values seen in the buffer into the parallel shift register through a parallel bus. Once the shift register held the full word needed to be transmitted, the control module commanded the shift register to begin shifting out values to the right in little endian format. This sequence of command governed by the control module was represented by a two bit signal described in more detail in Section 3. The ShiftRegControl module successfully sent values into the shift register from the buffer. It also successfully followed the required sequence of commands to shift out the transmitted data from the shift register. The shift register successfully maintained a true bit in each flip flop while not transmitting a word. This allowed the system to maintain a high value on the output line, as specified by the project description. The values seen in the shift register as well as the two-bit command sent from the control module can be seen in Figure 11.3.7 of the Appendix. In order for the data to be sent at a clock rate visible to the receiving system, specifically at 9600 Hz, a new clock divider needed to be functioning for the transmitting system. With the same logic described in Section 3 and above in Section 7, the SixteenCounter and the divideClock modules needed to create the 9600 Hz frequency. The success of this functionality in the system can be seen in Figure 11.3.3 as previously stated. Therefore, the combined success of each component of the transmitting system allowed the data stored in the microprocessor to be sent serially out of the system in the little endian format. Finally, KEY [0] successfully reset the system and cleared all flip flops.

The C program successfully sent messages to the buffer and then told the rest of the hardware to shift it out to send it. The character or message being sent was inputted in the console. The

console also gave prompts to send and showed what had just been sent. The console also shows what character or message was just received. We chose to use the NIOS II processor because it was easy to use. The processor had 20kbytes of on-chip memory. The System clock used in the processor was 50MHz. Overall the system worked as specified.

8 - ANALYSIS OF ERRORS

The design process did not meet any errors that were not resolved through simulation and analysis of results. All of the functions of the system, including a very low-level view, resulted as expected. One speed bump that occurred during the design of the project was that the gate array required specific tweaks in order to function. Given the strict timing of the system, synthesis into hardware introduced a new problem in constructing the UART. After discussing a few glitches seen on the board, the design team was informed by J. Peckol that the gate array often has tweaks and irregularities while working with SW [8] and SW [9]. SignalTap II showed a few irregularities surrounding these switches on a couple clock cycles. However, the error ceased to exist after more hardware logic was implemented. No further action was necessary.

9 – SUMMARY

The final project involved the construction of a functional Universal Asynchronous Receiver/Transmitter (UART) for a microprocessor-based application on the FPGA. This assignment implemented basic I/O processes for a complex system defined in C. Techniques involving framing and timing allowed for the asynchronous system to correctly implement I/O functionality to receive data. The transmission of data out of the system followed the same framing and timing techniques as well as the general structure of the serial data. Furthermore, the system required network communication between the gate array, the console, and the microprocessor. Communication between hardware and the microprocessor was constructed through software in C in the Altera environment. This system was initially tested by creating a testbench simulated using ModelSim and gtkwave. Synthesis into hardware was verified using SignalTap II. In hardware, the design allowed for both receiving and transmitting ten-bit words to memory, as specified in Section 7: Analysis of Results. The second part of this project required the implementation of an application designed using the C language. This program allowed for console I/O interaction with the gate array.

10 – CONCLUSION

To conclude, the construction of the Universal Asynchronous Receive/Transmit system was very educational. The design introduced new aspects to simulation and synthesis and required a lower-level understanding of proper timing. Synthesis into hardware introduced new problems in designing the project, including simple tweaks about the gate array and where the board is located in respect to the CPU during synthesis. No major errors arose during the design and synthesis of the project. The most time-demanding component to the project was the C language

implementation on the microprocessor. With only basic knowledge of the C language and very few guidelines to designing the application, constructing the C logic required a lot of time and research. The project was very informative and interesting. A more-complicated relationship between hardware and software was exciting to construct.

11 - APPENDICES

11.1 - Requirements Specification

Abstract – This specification describes and defines the design requirements for a basic UART.

Inputs to System - This design requires that serial input be stored to memory in a microprocessor. These inputs are defined the C application within the Altera environment. There is no hardware involved as input besides the system reset, KEY [0]. The serial input needed to follow a specific format in order to be comprehended by the system. When not receiving a word through the input line, the line should maintain a high value or 1. The start of a serial transmission should be denoted by a zero bit and seven bits to follow. The seven bits that proceed the start bit represent the data to be inputted in to memory. The values of these seven bits must be in little endian format. The eighth bit represents the parity bit. The transmitted word should hold an even parity. The parity bit to correct any deviation to this requirement. The end bit, the ninth bit, should be a zero. Again, the entire word inputted to the system must be comprised of the ten bits described.

The only input to the C program are the inputs from the base addresses which come from the the hardware designed in Verilog and the console input. The message being sent is the only input given the console.

Outputs to System – The design requires that the serial output from memory be sent in a specific format. This format must exactly match that described above according to the Inputs to System section. The parallel shift register is automatically programmed to output a high value or 1 when no word is being transmitted from the microprocessor. The hardware logic also hardcode the start and stop bits to be low or zero. The shift register shifts values out to the left, effectively assigning the serial data to be in little endian format. Furthermore, the C application verifies the parity bit in the transmitted data and corrects the value of that bit if the data does not have even parity. Overall, this system automatically creates a serial output in the correct format. The data being passed through the microprocessor can be seen on the output LEDS from LEDR [8:1].

Major Functions - The UART is designed to receive and transmit ten-bit words. Data received by the system is stored in memory in a NIOS II microprocessor. Once the processor is ready to transmit data, the ten bits stored in memory can be transmitted out of the system in the same format that it was received. This functionality allows multiple systems to communicate between each other as long as they share a common expectation within the serial data being received. All

UART systems should expect to receive serial data and transmit serial data all in the previously described format. The data being passed through the processor can be seen on the output LEDS.

11.2 - Design Specification

Abstract – This specification describes and defines the design requirements for a basic UART.

System Description – The UART receives and transmits data in a common format described in Section 11.1 Requirements Specification. Logic in the gate array that control the receive/transmit systems are tied to the microprocessor loaded on to the FPGA. The C language controls the processor through the console. The console, as well as the output LEDS, display the received data and the transmitted data.

Specification of External Environment – The UART is to operate in an industrial environment or room temperature surroundings. This unit is portable but operates on an outlet.

System Input and Output Specification - The system shall take no hardware inputs besides the KEY [0] hard reset. Serial inputs to be received through the system are built in to the C logic. Furthermore, the transmitted outputs are written to console and sent to the LEDs to be displayed. The system is based on a rising edge clock and is reset on a falling edge reset signal. Frequency, period, and time interval ranges are not specified. The system outputs values to be displayed on the LEDs [8:1] and to console in the Altera environment.

User Interface – The user shall control the reset value, KEY [0], and the console commands in the Altera Environment. The functions of these interfaces are described above in Section 3.3.

Time Base – The time base system is driven by an FPGA built in 50MHz clock.

11.3 – Waveforms

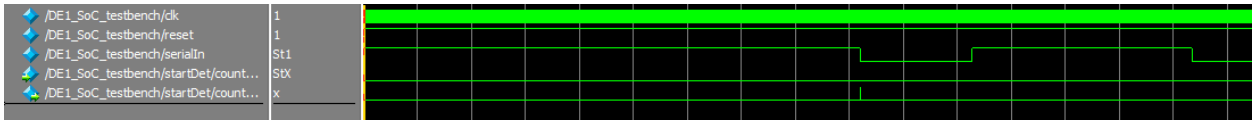

Figure 11.3.1 StartBitDetector Waveform

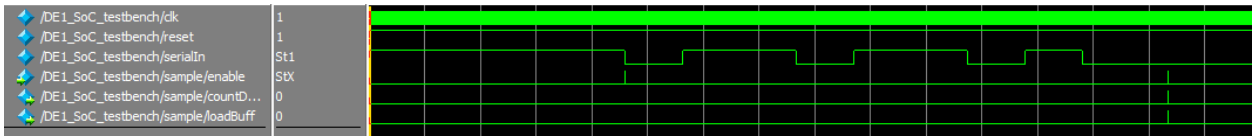

Figure 11.3.2 SampleCounter Waveform

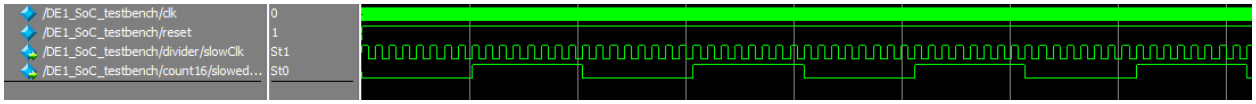

Figure 11.3.3 SixteenCounter and divideClock Waveforms

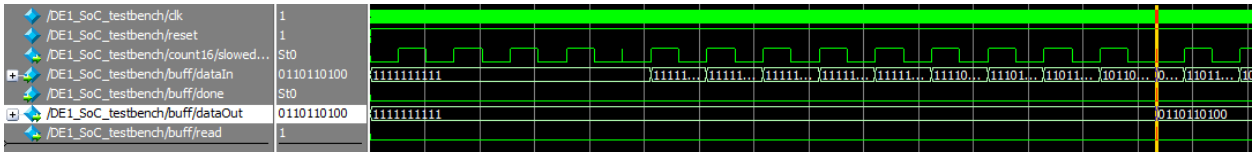

Figure 11.3.4 ReceiveBuffer Waveform

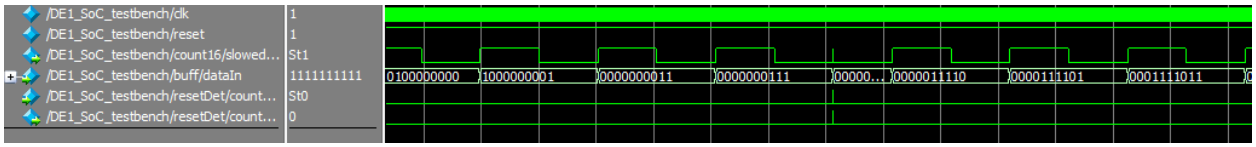

Figure 11.3.5 DoneDetector Waveform

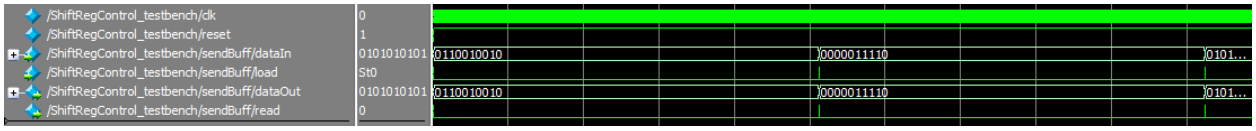

Figure 11.3.6 SendBuffer Waveform

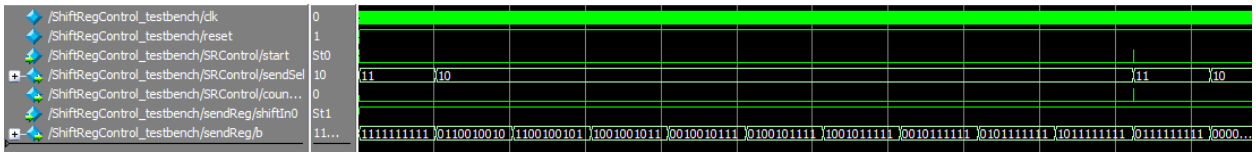

Figure 11.3.7 ShiftRegControl and Shift Register Waveform