**EE 371 Project 4**

**Extending a Simple Microprocessor**

**Building an Application**

**Qing Ran**

**Xuanlin Zhu**

**Siren Xu**

**Table of Contents**

Abstract3

Introduction3

Discussion of the lab3

Design specification3

Design procedure3

System description5

Software implementation 7

Hardware implementation 14

Test Plan15

Test Specification16

Test Cases17

Analysis of the Results18

Analysis of Errors21

Workload22

Summary22

Conclusion22

Appendices22

**ABSTRACT**

This report discusses the design and test of a network system between two microprocessors. The asynchronous serial network supports asynchronous full duplex data transmission and reception between two NIOS II microprocessors. The design and test of this project has two parts: first is to design the network and test serial communication with the network without microprocessor; second is to utilize the network on two microprocessors and test serial communication between the two processors. The result of the test is the same as we expected and the two microprocessors could communicate with the each other successfully.

**INTRODUCTION**

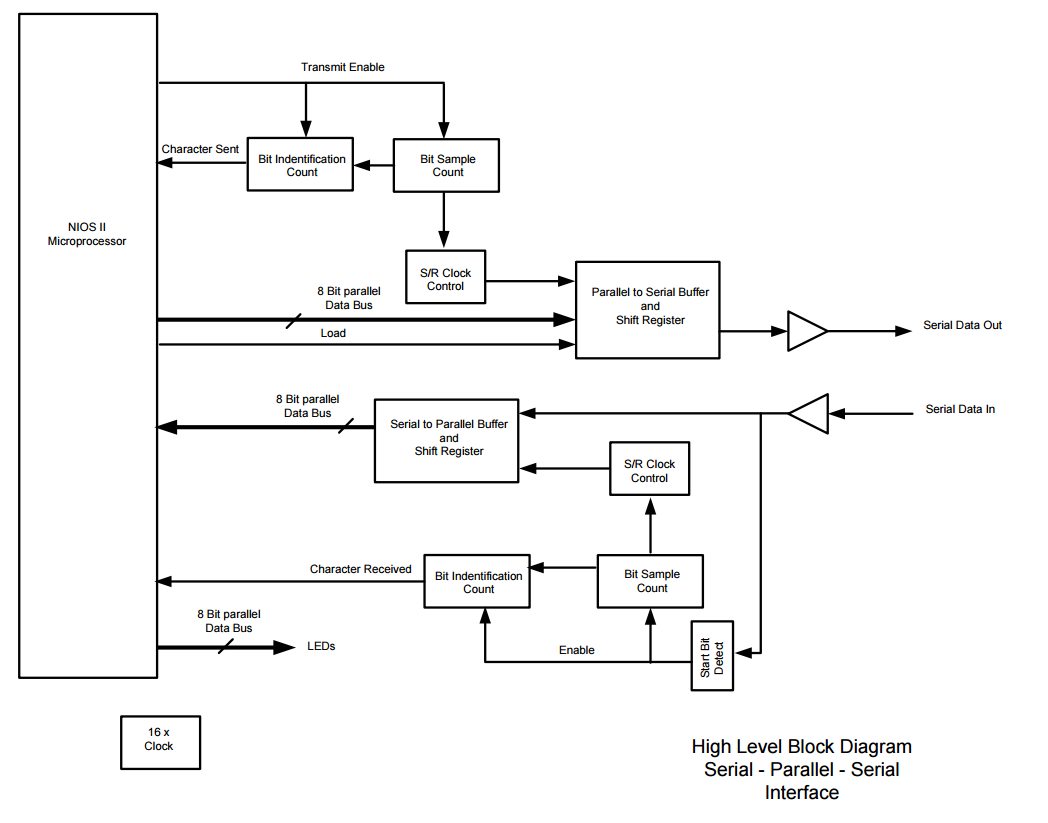
The purpose of this lab is to: first, develop and test a serial-parallel-serial network that will support asynchronous message exchange; second, design and develop a hardware system on the Cyclone V FPGA on the DE1-SOC board that comprises a NIOS II processor and utilize the network to communicate between two microprocessors; third, design a simple program in C language which can be played by two players on the two microprocessors. During the design procedure, the tools used are Quartus II IDE and NIOS II SBT for Eclipse. During the test, the Signal Tap II Logic Analyzer is used.

**DISCUSSION OF THE LAB**

**Design Specification**

The requirement of this lab is to first design and test an asynchronous serial network which supports asynchronous full duplex data transmission and reception. Second is to design and implement two NIOS II microprocessors on the Cyclone V FPGA on the DE1-SOC board and utilize the network system to communicate between the two processors. Third is to design a simple game which can be played between two players on two processors. For the network, two clock speed are specified, clk9600 and clk9600x16. The clk9600 will be the data transfer rate and the clk9600x16 will be the data sample rate. Also, two four-bit counters, bit sample count (BSC) and the bit identification count (BIC), will be used to indicate when to sample the data and how many data bit the system has read accordingly.

**Design Procedure**

****

**Figure 1.1 High Level Block Diagram Serial-Parallel-Serial Interface**

In terms of hardware, we used two DE1-SoC boards to clearly show the data communication. One board is used to send the data out from the NIOS II microprocessor, and another board is used to receive the data and display the output with the LEDs.

In terms of software, the top level design is developed using the high-level block diagram above. To accomplish the requirement for the serial-parallel-serial interface, we decided to build each module individually and then connect these modules to achieve the requirement for the entire dataflow. Also, when designing the BIC and BSC modules, the clock cycle for the BIC and the BSC are required to be not the same; so two clock cycles of different frequency are designed (9600Hz and 9600x16Hz). The detailed procedure is showed below:

**Clock**: This design requires two clocks. One clk9600 which has a frequency of 9600 Hz and the other on clk9600x16 which is 16-time faster than the clk9600. These two clocks are achieved by slowed down the internal CLOCK\_50. First in order to get clk9600x16 which has a frequency of 153600 Hz, CLOCK\_50 needs to be slowed down to a reasonable speed. By calculation, if CLOCK\_50 is slowed down by 2 to 5th square, a clock of frequency 1562500 could be achieved and this clock speed is nearly 10 times the clk9600x16. Therefore, clk9600x16 and clk9600 could be both implemented by slowing down CLOCK\_50 first by 2 to 5th square and then by 16.

**Sending data out from the microprocessor**: When the Transmit Enable signal is enabled, the Bit Identification Count module and the Bit Sample Count module are triggered. The character in console is changed to its corresponding ASCII number by the NIOS II microprocessor. Then the S/R Clock Control is enabled which only samples the middle of the output to keep the output data valid. Then the S/R Clock Control module, the 8Bit Parallel Data Bus and the Load signal are processed through the PISO Buffer and Shift Register module which outputs the serial data, this data is connected to another DE1-SoC board in order to display the LEDs on another board.

**Sending data into the LEDs**: The serial data is firstly sent from the previous step, then, when the start bit (0) is detected by the Start Bit Sample module, Bit Identification Count module is enabled by receiving the enable signal sent by the Start Bit Detect module, and it counts how many bits are included in the input data. Also, in order to avoid unstable data input, we decided to connect the shift register to a clock control module, which makes the shift register only takes the middle one of the whole samples: the Bit Sample Count module counts 16 times and the S/R Clock Control module set only the 8th clock cycle being true. Finally, the output of the S/R Clock Control is sent to the SIPO Buffer and Shift Register, which transfers the serial input into 8 bit parallel data bus and sends the parallel data bus to the LEDs to display the parallel data.

**System Description**

**serialComCPU:**

Input: CLOCK\_50, SW, GPIOIn

Output: LEDR, GPIOout

This module is the top level module which connects the provided CPU, clock, input and output together. It firstly divide the cycle from 50MHz to 1562500Hz through the clock divider module, after that, two clock counter are used, the first counter outputs a frequency of 153600 which is very close to 10 times slower of 1562500Hz, and the second counter outputs a frequency of 9600Hz which is 16 times slower of 153600Hz. Then it sends the relative parameter to the provided CPU module; it also activates the transmit module to send data out from the microprocessor, and the receive module to receive the user input to proceed it to the microprocessor.

**Transmit**:

Input: tranEnable, clk 9600x16, clk9600, reset, dataIn

Output: dataOut, charSent

This module is a combination of the four modules: transmitEnable module, PISO module, BSC module and the BIC module. It transmits data from the CPU as serial data out through the PISO.

**Receive**:

Input: dataIn, clk9600x16, clk9600, rst

Output: dataToMem, charRecived

This module is a combination of the four modules: startBit module, SIPO module, BSC module and the BIC module. It receives serial data and passes to the CPU as the SIPO.

**transmitEnable**:

Input: enableIn, clock9600, charSent, clock, reset

Output: enableOut

This module uses a state machine to change the enable signal from the microprocessor to the two counters. The first state is an idle state which delays one cycle for the system to send data; the second state is the enable state which is the actual Transmit Enable signal sent to the Bit Identification Counter module and the Bit Sample Counter module. The third state is the empty state which is entered after one character is sent from the microprocessor.

**startBit**:

Input: data, clk, rst

Output: enable

This module detects the start bit (0) in the input data. If it reads a 0, it will enable the BSC and the BIC.

**SIPO**:

Input: data, clk, rst

Output: dataOut

This module is a serial in parallel out shift register with buffer. It will read data bit by bit and send out the data to the buffer after receiving all data.

**PISO**:

Input: load, clk, rst, dataIn

Output: dataOut

This module is a parallel in serial out shift register with buffer. It will read the data to be sent from the buffer and output the data bit by bit.

**BSC**:

Input: enable, rst, clk

Output: Count

This is a four bit counter with an enable signal. It counts from 0000 to 1111 when enable is true.

**BIC**:

Input: enable, rst, clk

Output: Count

This is a four bit counter with an enable signal. It counts from 0000 to 1111 when enable is true.

**clock\_divider**:

Input: clock

Output: divided\_clocks

This module slows down the input clock. The output is a 32-bit binary number, each bit of this number represents a different clock speed.

**Software Implementation**

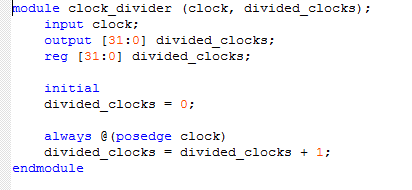
**Clock**

Figure 2.1 shows the overall implementation of the clk9600 and clk9600x16.

clock.PNG

**Figure 2.1 Implementation of clk 9600Hz and clk 9600x16 Hz**

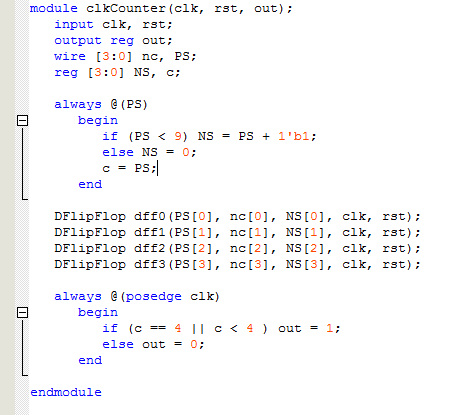
In order to slow down the internal CLOCK\_50. A clock divider is implemented as shown below in the figure 2.2.



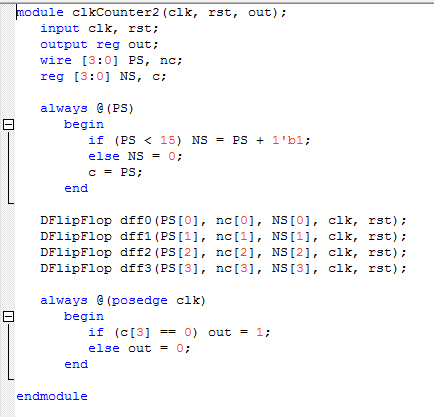
**Figure 2.2 Code for the clock divider module**

In this module, firstly, we create a 32-bit binary number and then increment this number by 1 at each positive edge of the original CLOCK\_50. To lower the frequency by a reasonable amount, set the output clock to be equal to the 5th bit of the binary number, which means the original clock rate is divided by 2^5. Therefore a new clock with a frequency of 50MHz / 2^5 = 1562500 Hz is achieved.

Second step is to slow down the clock with frequency 1562500 by 10 to get clk9600x16 and then slow down clk9600x16 by 16 to get clk9600. This is implemented by two clkCounters shown below in figure 2.3 and figure 2.4.



**Figure 2.3 Code for the clkCounter.**



**Figure 2.4 Code for the clkCounter2.**

clkCounter is a state machine which counts from state 0 to state 9. During state 0 to 4, the output is set to 1 and during state 5 to 9, the output is set to 0.

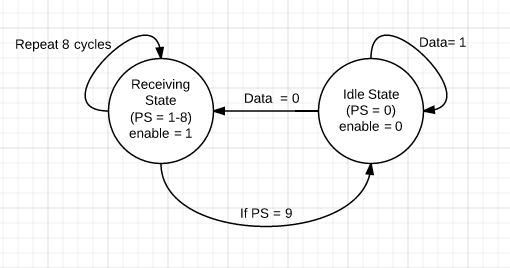
clkCounter2 is a state machine which counts from state 0 to state 15. During state 0 to 7, the output is set to 1 and during state 8 to 15, the output is set to 0.

**Network**

The design of our serial-parallel-serial network follows the block diagram in figure 1.1. The implementation of each major module is given below:

**Start Bit Detect:**

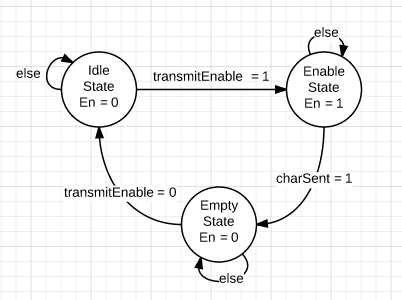
The start-bit detection function is implemented by a state machine as shown in figure 2.5. The state machine stays in the idle state (state 0) unless it receives an input data of 0 which means the next 8 bits are the input data we need. After receiving a 0, the state machine will go to the receiving state and stay there for 8 cycles. In the receiving state, an output called enable will be set to 1 and send to BIC and BSC to enable them counting input data bit. If enable is 0, BIC and BSC will be in the reset state and stop counting. After receiving all 8 bits, the state machine will go back to the idle state and set the enable signal back to 0.



**Figure 2.5 State machine diagram for the Start Bit Detect module**

**transmitEnable:**

This module shown in figure 2.6 is a state machine which controls the enable signal of the BCS and BIC for the transmitting procedure. In the Idle state, the output enable signal will be set to 0. Once transmitEnable is 1, the state machine will go to the Enable State and set the output enable signal to true. When all data is sent (charSent = 1), the state machine will go to the Empty state and set Enable to 0. This state is acting like a denouncer which requires the transmitEnable to be reset once between two output data.



**Figure 2.6 State machine diagram for the transmitEnable module**

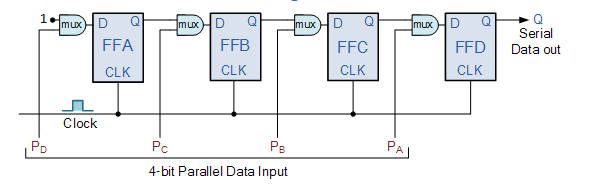
**Bit Identifier Count (BIC):**

The Bit Identifier Count (BIC) is a four-bit counter with an enable signal which can count from 0000 to 1111. The enable signal is received from the start bit detector. When enable is true, the BIC will start counting from 0000; when enable is false, the BIC will be reset to the 0000 state.

**Bit Sampling Count (BSC):**

The Bit Sampling Count (BSC) has the same implementation as the BIC. The difference is that the BSC is connecting to a clock that is 16x faster than the clock of the BIC.

**Parallel in Serial out Shift Register (PISO) with buffer:**

The diagram for the PISO shift register is below and we actually used an 8-bit register for the design 

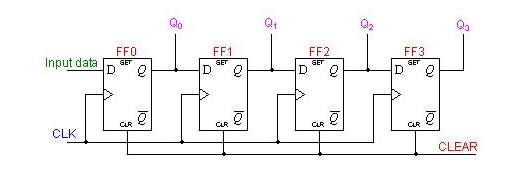
**Figure 2.7: Example diagram for a 4-bit PISO shift register**

The CPU first sends the data to the buffer, then when load is set to true, the PISO will load the data from the buffer simultaneously.

Then each bit is shifted to the right with the largest digits being 1. For example, if we want to shift the number 1111 with the PISO shifter, we can get 0001, 0011, 0111, 1111 (the least significant bit is firstly shifted out with the rest of the digits being 0, in the last cycle, 1111 appears as the output which is the same as the original number).

**Serial in Parallel out Shift Register (SIPO) with buffer:**

The diagram of the SIPO shifter is shown below :( we actually used an 8 bit shifter)



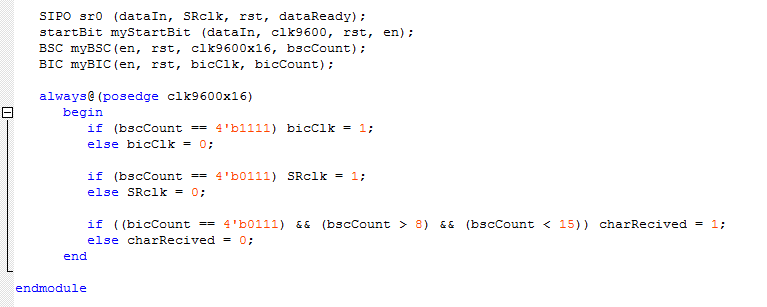
**Figure 2.8 example diagram for a 4-bit SIPO shift register**

We used the SIPO (Serial-In to Parallel-Out) shift Register to store the received data. For example, if the input data is 1111, the data stored in the SIPO shift register will be changed from 0000 to 1000 to 1100 to 1110 to 1111.

The output of the SIPO will be first send to an 8-bit buffer in parallel and then after all data is received (charRecoved set to 1), the buffer will be clocked and send the data to the CPU.

S/R Clock Control:

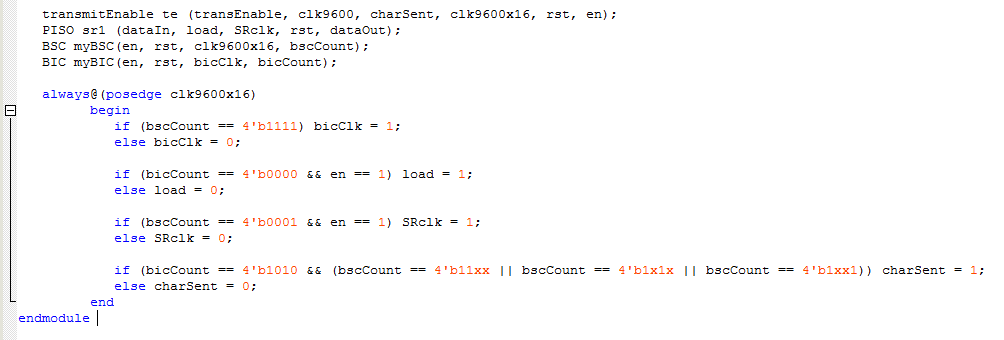
The S/R clock control of the receiving procedure is implemented by defining two different clocks as shown below in figure 2.9.



**Figure 2.9 Code for the S/R Clock Control**

As shown in the code, the BSC is connected to clk9600x16 and the start-bit detector is connect to clk9600, so BSC is counting at a rate 16-time faster than the data flow which means BSC counts 16 times every time a new bit is received. Another two clocks are defined based on the count of BSC. bicClk decides when BIC is incremented by 1 and SRclk decides when SIPO takes a new data bit and shifts by 1. SRclk is set to 1 when bscCount equals to 7 which means the SIPO will receive the data at the middle of the BSC counting cycle (max tolerance). bicClk is set to 1 when bscCount euqals to 15 which means BIC will be incremented by 1 every time BSC counts 16 times. The charRecieved signal is set to 1 when all data is received by the SIPO. When bicCount equals to 7 (the last data bit is received) and bscCount is between 8 and 15 (the SIPO has read the last data bit), the charRecieved signal will be set to 1.

The S/R clock control of the transmitting procedure is shown below in Figure 2.10.



**Figure 2.10 Code for the S/R clock control of the transmitting procedure**

The transmitting procedure basically has the same implementation as the receiving procedure. The enable signal for BSC and BIC is now generated by the transmitEnable function. bicClk and SR clk still have the same definition as in the receiving procedure. When bscCount equals to 15, bicClk will be set to 1 which means BIC will be incremented by 1. When bicCount equals to 0 (just start counting), the load signal will be set to 1 to let the PISO read the output data from buffer. When bscCount equals to 1, SRclk will be set to 1 which means PISO will output one bit data. charSent indicates if all data has been transmitted. When bicCount equals to 10 (includes two zeros at the begining and the end) and bscCount is between 8 and 15, the charSend signal will be set to 1.

**C Application**

1. Hardware serial communication test -- one system

The first program is designed to test the serial communication system on one NIOS II processor as shown in appendices B. While no new character is received from the console, transEnable will be set to 0. Once a new character is read from the console, the system will start sending the character until charSent is set to 1. Then the system will start reading the character and output to the LED. Also when reading the character from the console, the program will set the leftmost bit as the parity bit which means if the binary number has an odd number of 1’s, it will set the parity bit to 1 to make the number have an even number of 1’s.

2. Hardware serial communication test -- two system

The second program is designed to test the serial communication system between two NIOS II processors as shown in appendices C. In this program, the code is basically the same as the previous application, but the difference is that one processor is sending data only and the other one is receiving data only.

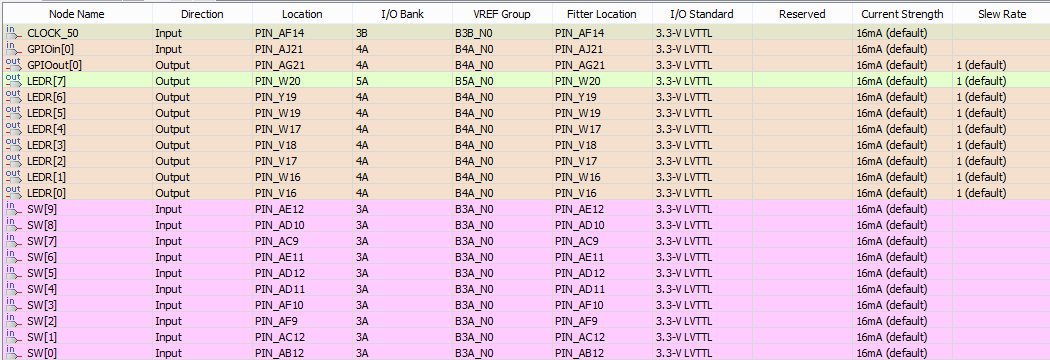
**Hardware Implementation**

In this lab, we used Qsys tool from Quartus to create two NIOS II processors on each board. After following the tutorial, the Qsys tool generates the Verilog modules of a NIOS II processor. We called the processor module in a top-level Verilog file to program our NIOS II onto the FPGA.

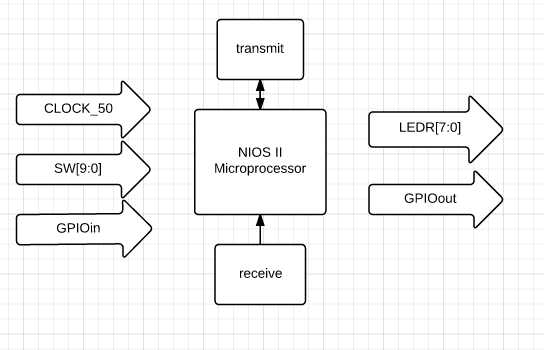
We connected CLOCK\_50 from the FPGA as the clock of the NIOS II processor. Also, we used LEDR [7:0] as the outputs for our processor. We then added SW [9] as our reset input signal. dataToMem and dataToSent are connected to the receive module and the transmit module accordingly, and they represent the data received and the data to be sent. charSent and charRecieved are signals that indicate all data has been received or sent, and they are connected to the receive module and the transmit module accordingly. transEnable indicates the CPU is outputing data. This signal goes from the CPU to the transmit function. The Serial Data In and Serial Data Out are the input and output to the receive module and transmit module. These two signals are connected to GPIOin and GPIOout accordingly.

After connecting two microprocessors through the network, the system will be able to run our C program. This program will run through our NIOS II processors, which are programed on the DE1-SoC FPGA board, and output the result through on board LEDR or Eclipse console.

The pin assignment of the LEDR [7:0] and the SW [9:0] are shown in the figure 3.1.

****

**Figure 3.1 Pin assignment for the system**



**Figure 3.2 Hardware implementation block diagram**

**TEST PLAN**

We have to verify three parts when we test the outputs;

The first test output is displayed on the Signal Tap. The waveforms of three clocks with different frequency (9600Hz, 9600x160Hz and 9600x16Hz) are tested. The Signal Tap output for three clocks should show that the 9600x16Hz clock counts 16 times faster than the 9600Hz clock. There should be 16 cycles of the 9600x16Hz clock in one clock cycle of the 9600Hz clock.

The second test is testing serial communication without NIOS II. The output display on the LEDs is tested, and we can check whether the LEDs are displaying the correct input corresponding to switches.

The third test is testing serial communication with one NIOS II microprocessor and check if the LED is displaying the correct ASCII number in binary on a single system based on the NIOS II microprocessor.

The fourth test is testing serial communication with the console and two microprocessors. The LED display as the output is tested to see if the first system’s ASCII number is correctly sent. Also, by displaying the ASCII number on the LEDs, the output is tested to see if the ASCII number is correctly received.

**TEST SPECIFICATION**

1. Testing the clocks

In this part, only the outputs are need to be tested to make sure the clock cycles meet the requirement. The frequency of the 1.5625MHz (50MHz / (2^5)), 9600x16Hz and the 9600Hz clock should be tested. The first frequency is 10 times faster than the second frequency, and the second frequency is 16 times faster than the third frequency. The clock cycles can be tested by counting them and compared.

2. Testing serial communication without NIOS II

In this part, the inputs are eight switches; an extra switch is used to represent Transmit Enable signal. The outputs are eight LEDs. After pressing a switch, we can check whether that data is correctly received by checking which LED is lit. To individually test if the transmit module and the receive module are transmitting the correct data without being affected by the microprocessor, the microprocessor part is brought out from the whole system. The input from the switches is used to represent the 8 bit parallel Data Bus which originally the data is being sent out from the microprocessor. This procedure should generate serial data in the transmit part. Then this generated serial data is sent to the receive module, this generates 8 bit parallel Data Bus. Finally, to display the output on the LEDs, the 8 bit parallel Data Bus is connected to the LEDs, and by pressing the switch and checking the LED’s status, we can verify if the receive and transmit module work correctly .

3. Testing serial communication with one NIOS II microprocessor:

In this part, the input is the character written in console, and the output is the LED display (actually displaying the ASCII number for that character). The microprocessor is added to accomplish the character feature. By verifying if the LEDs are displaying the correct binary ASCII number for the character in council, the correctness of the whole system can be checked. The microprocessor transfer the character into its corresponding ASCII number, and send it to transmit module. Then, in the receive module, the output is finally generated and by sending the 8 Bit Parallel Data Bus to the LED display, we can then  check whether the LED display is correct.

4. Testing serial communication with the console and two microprocessors:

In this part, the input is the console character prewritten; the output is the LEDs LED [7:0]. Two microprocessors are added to the two systems. By verifying if the LEDs are displaying the correct binary ASCII number for the character in console, two things are verified. 1. The data is correctly sent. 2. The data is correctly received. The whole system’s serial communication correctness is confirmed if the LEDs display the correct pattern corresponding to the character.

**TEST CASES**

1. Testing the clocks

To set up the test, the 50MHz is firstly used as a basic frequency. One clock divider is used which can divide the 50MHz into specific frequency, and two clock counters are used to modify the frequency with specific number. Then, to meet the requirement of the 153600Hz, the clock with 50MHz frequency needed to be slowed down by about 32 times, which is 2 to the 5th square, this step produces a frequency of 15625000Hz, and this is about 16 times of 9600Hz. Then the frequency of 9600Hz is obtained by slowing down the 9600x16 Hz by 16 times. After obtaining 9600Hz and 9600x16Hz, 9600x160Hz, we can display the three frequencies on the Signal Tap and compare the results

2. Testing serial communication without NIOS II

To set up the test, we simply manually substitute the microprocessor’s 8 Bit Parallel Data Bus with the switch SW [7:0], and connect the output directly to the 8 Bit parallel Data Bus of LED [7:0]. Also, we used another switch SW [9] to substitute the Transmit Enable signal from the microprocessor. The input for the whole system will be SW [7:0] plus SW [9] and the output signals to be measured is which LED is lit. The expected result should be: when we turn on one of the switches, the LED with the corresponding number is lit. For example, if SW [1] is pressed, the LED [1] should lit which indicates the correct data is receiving and sending between the receive module and transmit module. If all switches light up the corresponding LEDs, we can conclude that the test is passed. Elsewise, the test fails.

3. Testing serial communication with one NIOS II microprocessor:

To set up the test, we added the NIOS II microprocessor and connected it to a DE1-SoC board in order to show the ASCII value for the character in console. The inputs to the system is the character in console which is later transferred to the corresponding ASCII number. A switch SW [9] is used as the transmit enable signal. The output of the system is the LEDs (LED [7:0]). The expected results should show that the LEDs are displaying the correct ASCII number for that character. For example, when the council is written to a character ‘D’ as input, the LED display should be the ASCII number of ‘D’ which is 68, and the actual display of the LEDs should be: only LED[2] and LED[6] are expected to lit, which represents the binary number of 68 (1000100). If the LEDs are actually displaying correct ASCII number for the character written in console, we can conclude that the test is passed.

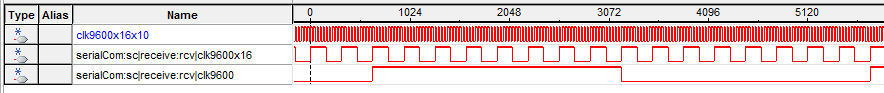
4. Testing serial communication with the console and two microprocessors:

To set up the test, we connected two microprocessors together (input to output and output to input) and set SW [9] as reset on both processor. The input entered to the console is b and the output to be measured are the LED [7:0] on the receiving processor. If data is transferred correctly, the ASCII number of b would be displayed on the LEDs as binary number (66) on the receiving processor. If the displayed LED patterns matches 66 in binary, the test is passed.

**ANALYSIS OF THE RESULTS**

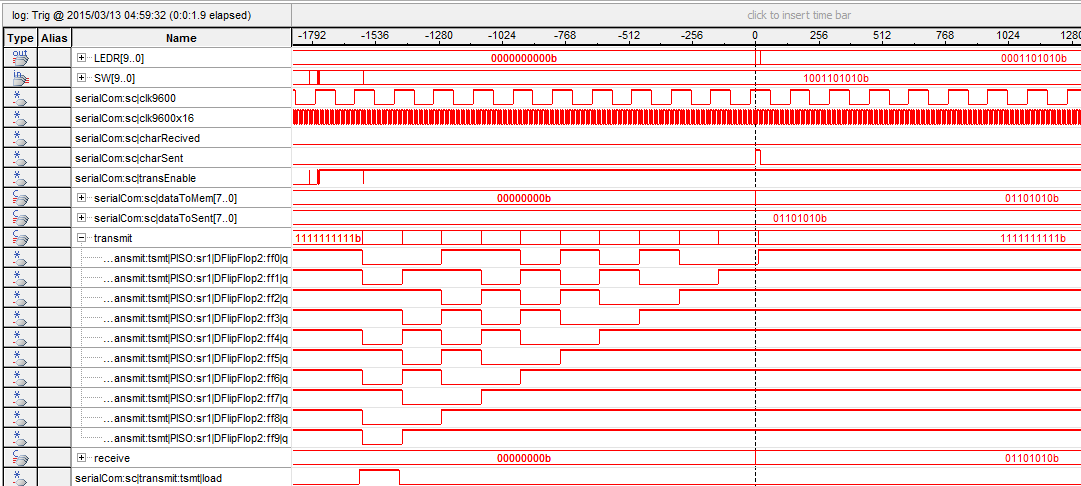
1. Testing the clocks

The figure 4.1 below shows the Signal Tap screenshot of the three clocks we got from the internal CLOCK\_50. As shown in the figure, clk9600x16x10 is 10-time faster than clk9600x16 and clk9600x16 is 16-time faster than clk9600. This screenshot shows that the clock divider and clockCounter we designed is working as we expected and the clocks we used in the system is correct.

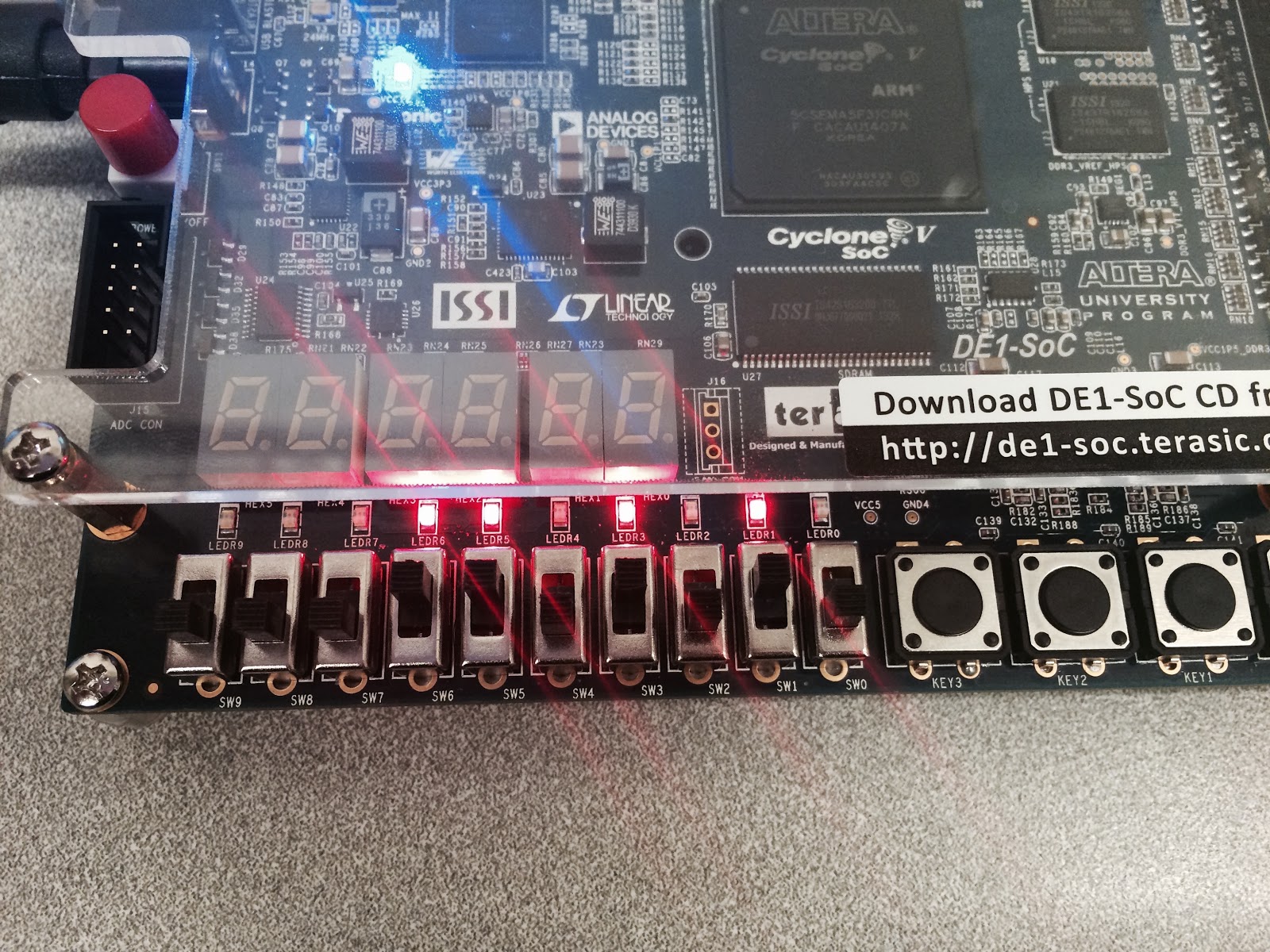
**Figure 4.1 Signal Tap screenshot for the three clocks of different clock cycle**

2. Testing serial communication without NIOS II

The picture given below in figure 4.2 shows the test of serial communication without connecting to the NIOS II microprocessor. As shown in the picture, the input data is given by the switch and the output data is displayed on the LEDs. According to the picture, the LED is turned on if the corresponding switch is turned to high which means that the network is sending and receiving data correctly. The screenshot in figure 4.2 shows the Signal Tap screenshot of this test.



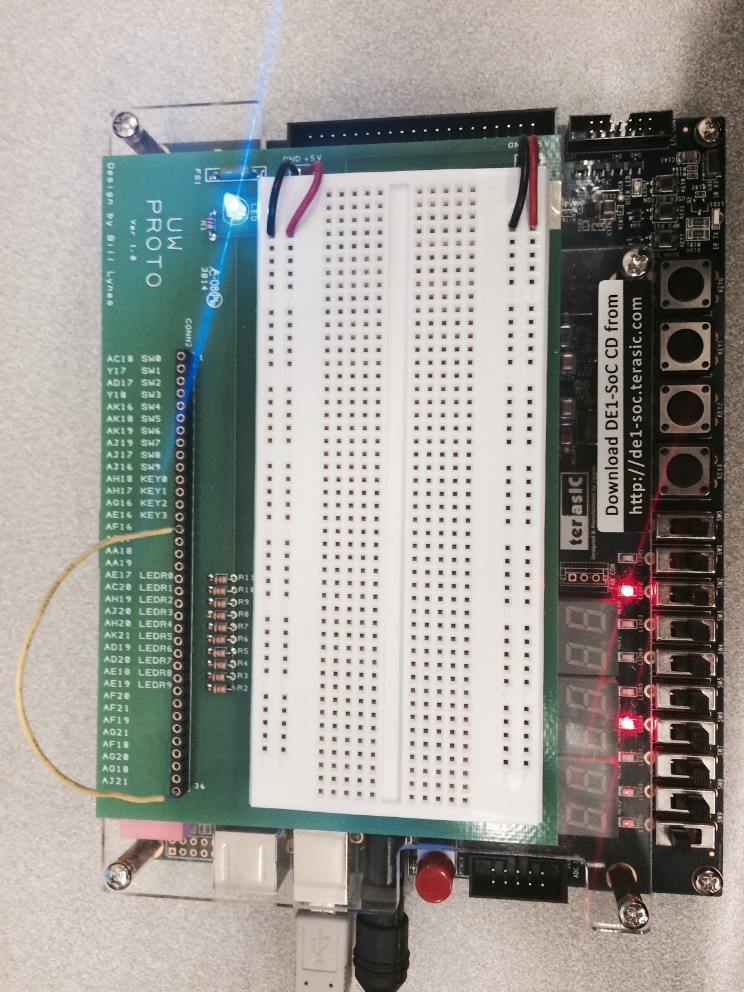
**Figure 4.2 Signal Tap screenshot for the hardware test without NIOS**



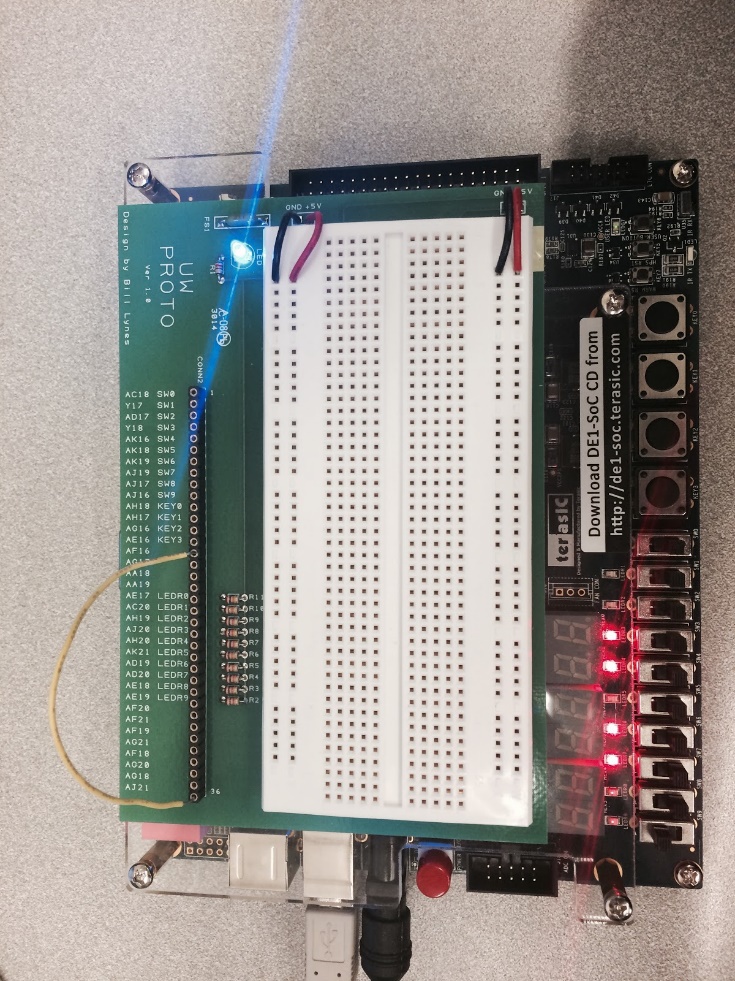
**Figure 4.3 LED test for the serial communication with NIOS**

3. Testing serial communication with one NIOS II microprocessor:

The two pictures below show the LED display of testing serial communication with one NIOS II processor. As shown in figure 4.4, the character D is entered from the computer and LEDs are displaying the ASCII number of character D (68). As shown in figure 4.5, the character X is entered from the console and LEDs are displaying the ASCII number of X (88). Also in the test, the parity bit is shown in the LED 7 to test the correctness of the parity bit. This test shows that after implemented to the microprocessor, the network still receiving and sending data correctly and the C application we design is working as we expected.

.

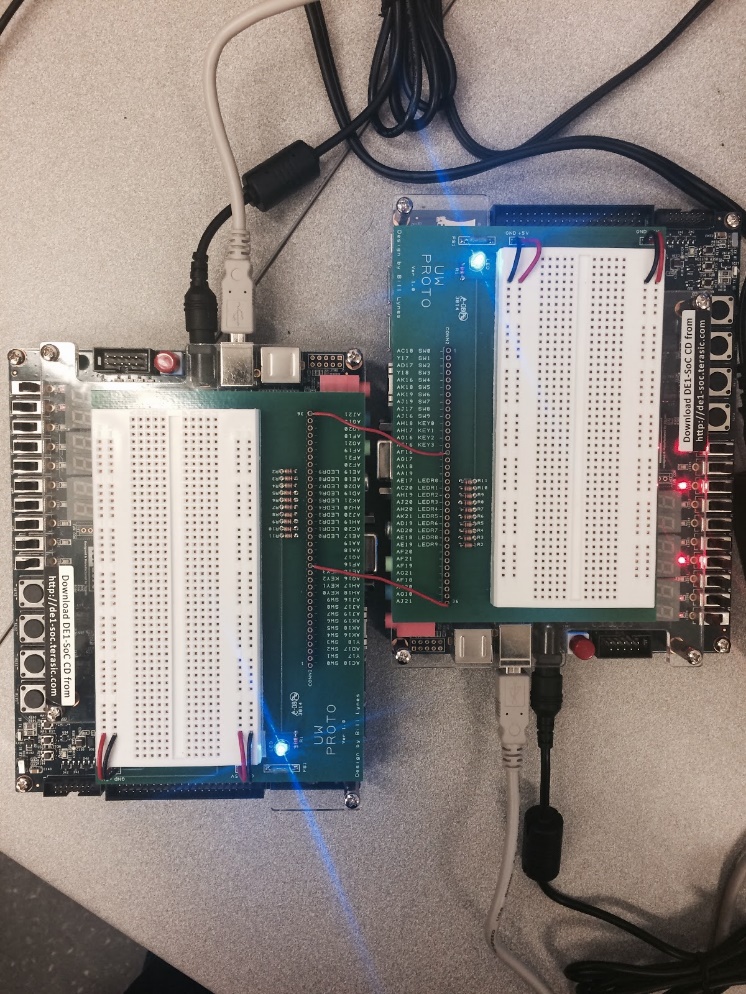
**Figure 4.4 LED display for serialCom testing w/ NIOS – character ‘D’**



**Figure 4.5 LED display for serialCom testing w/ NIOS – character ‘X’**

4. Testing serial communication with the console and two microprocessors:

This picture below shows the LED display when a character is entered in the console and the ASCII number of this character is shown on the LED. As shown in the picture, the character entered is b and the ASCII number 66 is displayed as binary number on the LEDs. This test shows that data could be successfully transferred between two microprocessors and the C application we design is working as we expected.

****

**Figure 4.6 LED display for serialCom testing between 2 systems**

**ANALYSIS OF ANY ERRORS**

When designing the clock control, we was trying to use assign method to set up the SRclk and bicClk at first, but this method doesn’t work properly and leading to unknown errors when we debugging the program. We found that when using the assign method, the SRclk and bicClk is counting in a wired behavior and they were counting at a changing rate. After changing this part of the code to behavioral Verilog, the problem is solved and these two clocks were counting as we expected.

**WORKLOAD**

|  |  |  |
| --- | --- | --- |
| Design | Xuanlin Zhu, Qing Ran, Siren Xu | 5 hours |
| Coding | Xuanlin Zhu, Qing Ran, Siren Xu | 15 hours |
| Debugging | Xuanlin Zhu, Qing Ran, Siren Xu | 30 hours |
| Documentation | Xuanlin Zhu, Qing Ran, Siren Xu | 5 hours |

**SUMMARY**

In this lab, we successfully designed and tested the asynchronous serial network and after implementing the network on two NIOS II microprocessors, the two processors could successfully exchange data. For the network, we followed the block diagram from the design specification and it successfully supports data exchanging between two microprocessors. For the microprocessors, we followed the tutorials and added several input and output ports to connect the network.

**CONCLUSION**

After finishing this lab, we get more familiar with the Qsys and the NIOS II SBT for Eclipse by designing this more complex project. Also by following the specification and the block diagram, we developed an asynchronous serial network. The experience gives us a better idea about data transporting, serial communication and especially timing. While building the system, we learned the correct process of designing, implementing and testing a complex system. This knowledge will help us in the feature when we are designing any other project.

**APPENDICES**

Appendix A

Requirements Document

* a serial-parallel-serial network that will support asynchronous message exchange
* Design and develop a NIOS II processor on the Cyclone V FPGA on the DE1-SOC board
* Utilize the network on the NIOS II processor and communicate between two processors
* Design and test a program in C which can be played on two separate microprocessors by two players.

Appendix B

C Application for testing the serial communication system with NIOS

int main()

{

alt\_putstr("Hei Hei Hei\n");

int out = 0;

int oldOut = 0;

int sent = 0;

int in = 0;

while (1){

while (out == oldOut){

IOWR\_ALTERA\_AVALON\_PIO\_DATA(transEnable, 0);

out = parity ((int) alt\_getchar());

}

while (sent == 0) {

IOWR\_ALTERA\_AVALON\_PIO\_DATA(dataOut, out);

IOWR\_ALTERA\_AVALON\_PIO\_DATA(transEnable, 1);

sent = IORD\_ALTERA\_AVALON\_PIO\_DATA(charSent);

}

in = IORD\_ALTERA\_AVALON\_PIO\_DATA(dataIn);

IOWR\_ALTERA\_AVALON\_PIO\_DATA(LED, in);

sent = 0;

oldOut = out;

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int parity (int x){

int one = 0;

int temp = x;

while (x > 0) {

one = one + x % 2;

x = x / 2;

}

return (temp + 128 \* (one % 2));

}

Appendix C

C Application for testing serial communication between two systems

int main()

{

alt\_putstr("Hei Hei Hei\n");

int out = 0;

int oldOut = 0;

int sent = 0;

while (1){

while (out == oldOut){

IOWR\_ALTERA\_AVALON\_PIO\_DATA(transEnable, 0);

out = (int) alt\_getchar();

}

while (sent == 0) {

IOWR\_ALTERA\_AVALON\_PIO\_DATA(dataOut, out);

IOWR\_ALTERA\_AVALON\_PIO\_DATA(transEnable, 1);

sent = IORD\_ALTERA\_AVALON\_PIO\_DATA(charSent);

}

sent = 0;

oldOut = out;

}

return 0;

}