

EE/ CSE 371 Lab Report Final Project

Using a Simple Microprocessor - Building an Application

University of Washington

Department of Electrical Engineering

Spring 2017

Liyuan Wang

Zhouyuan Xu

Jiaqi Zhang

Abstract

The objective of this lab is to practice system integration of microprocessor and FPGA, as well as developing basic I/O for systems. Fundamental topics of this lab includes design of basic finite state machines, use of registers, HDL modeling, general purpose I/O of systems, integration of microprocessor, parallel and serial communication methods and protocols, and simple software MAC logic design. The project created in this lab is a communication system composed of a FPGA design with embedded NIOS II microprocessor and software defined MAC logic that provides a bi-directional chatting application.

Table of Contents

Introduction	4
System Description	4
Hardware Description	5
Software Description	8
Presentation and Results	9
Error/ Failure Analysis	14
Summary and Conclusion	15
Individual Contribution	15
Appendices	16
Appendix A. ASCII Binary Table	16

1. Introduction

The objective of this lab is to practice system integration of microprocessor and FPGA, as well as developing basic I/O for systems. In this lab, a parallel to serial communication port system was developed using FPGA, and a program was created using C and ran on NIOS II microprocessor, which is integrated with the FPGA, to provide some data processing and user interface functionalities. The FPGA system is connected to the NIOS II microprocessor through parallel data bus and convert the data stream to serial format using a specific protocol format, with start bit (1) and end bit (0), to transmit data to another identical system. Similarly, the FPGA system also receives data stream in the same serial format, convert to parallel transmission, and deliver to the NIOS II microprocessor. The final project achieved duplex communication between two instances of the system.

2. System Description

This entire communication system is composed of a hardware component and software component. The hardware component include a NIOS II microprocessor configured with a FPGA design. The software component is a simple C program, which runs on the NIOS II microprocessor , that provides a user interface through the Console that takes text input user and print text output from and to the user.

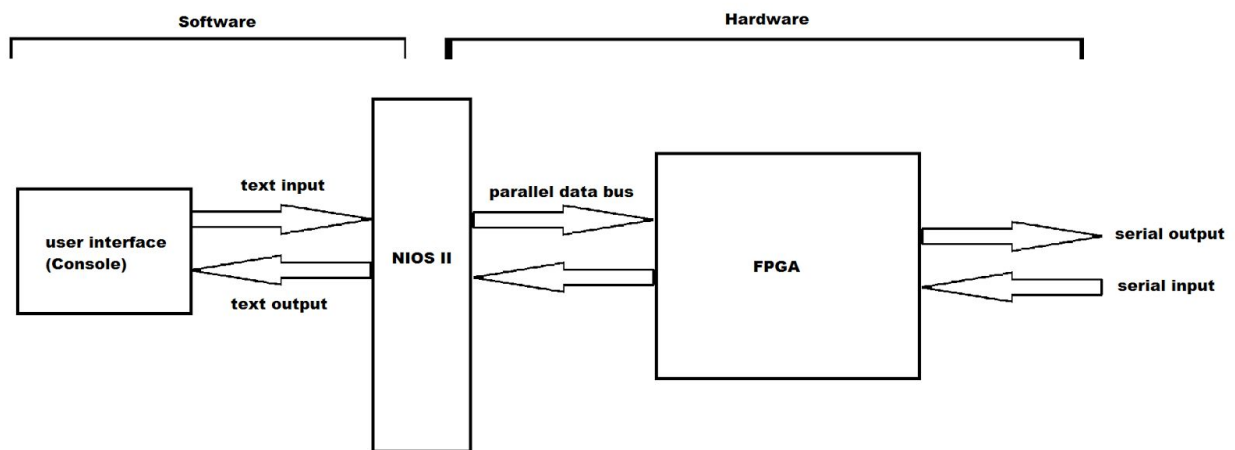


Figure 1 High Level Block Diagram of the Entire Communication System

The software takes text input from the user and convert it to binary form inside the microprocessor. Then the binary data of the text input is delivered to the FPGA system. Once received data from the microprocessor, the FPGA system put the data into frames and transmit the frames in serial through its serial output port. Similarly, the FPGA listen to its serial input port and decode any frames it receives, and then deliver received binary data to the microprocessor. When the microprocessor receives data from the FPGA side, it again converts the binary data into text form and prints the output to the console.

The FPGA component is supplied with a slower clock than the NIOS II microprocessor to

ensure signal integrity. The NIOS II is supplied with the default 50 MHz clock to ensure processing speed of the software code.

3. Hardware Description

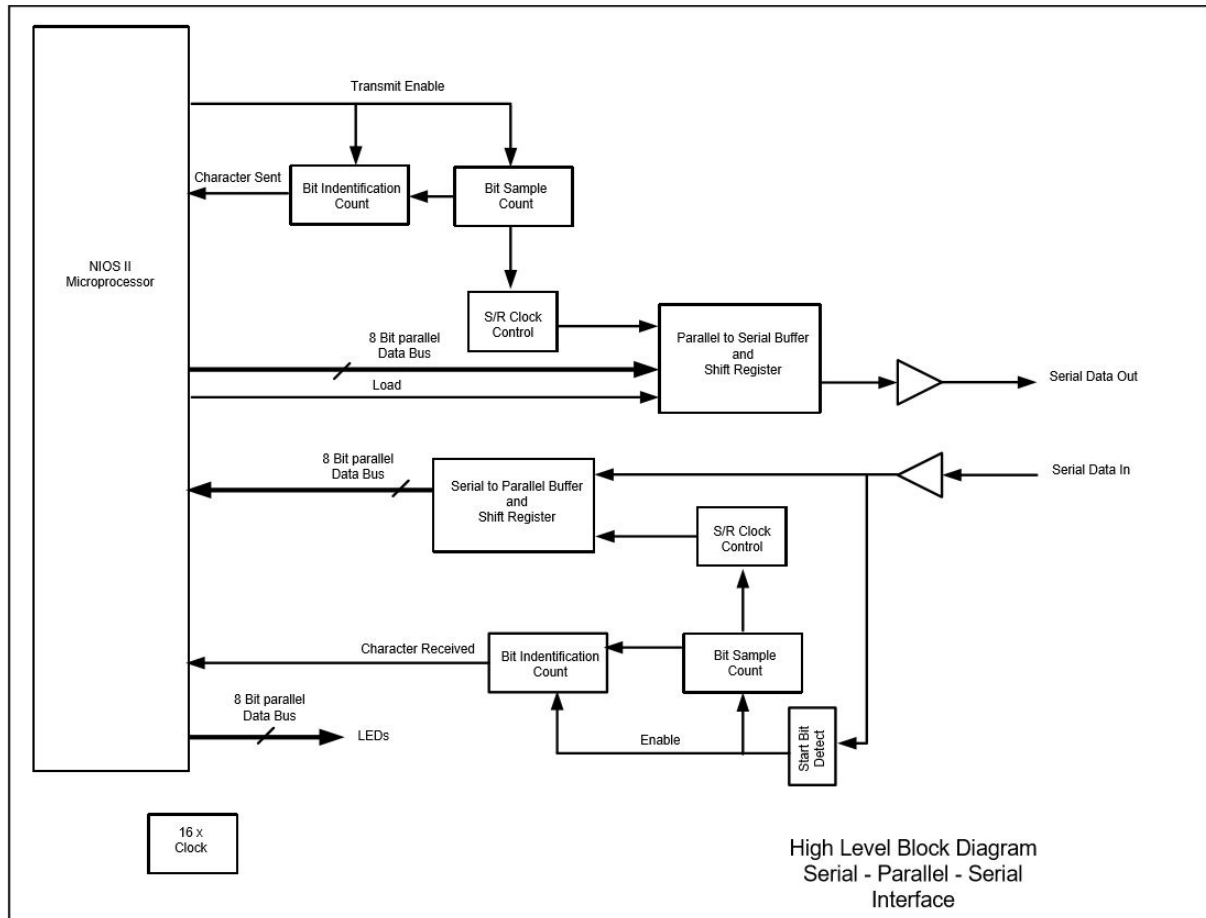


Figure 2 High Level Block Diagram of the Hardware of the Communication System

Communication System

The system consists three parts: the Nios II Microprocessor, the transmitting part, and the receiving part. The transmit and receive process are independent from each other. The microprocessor takes 8-bit Parallel Data Bus from the receiving port, and outputs data as 8-bit Parallel Data bus to the transmitting part. The handling of the parallel data is done in the Nios II Microprocessor.

Data comes in from or transfers out to outside connections such as an General Purpose Input/Output(GPIO) port, or bluetooth connections one bit at a time. Our system performs asynchronous serial communication, in which serial data is grouped into frames. These serial data are clocked by two 4-bit counters - *bit sample count (bsc)*, and *bit identification count (bic)*.

Framing

A frame starts with a *start bit*, consists 8 bits of data in the middle, and ends with a *stop bit*. The *start bit* enables receiving device to temporarily synchronize with the transmitting device. The *Stop bit* indicates the end of a frame and gets the receiving device ready for the next frame. Both the receiving device and the transmitting device must share the same framing setup to ensure successful temporal synchronization. In our system, the *start bit* will be 0 and the *end bit* will be 1.

0	8-bit data	1
---	------------	---

Figure 3 A frame with start bit and end bit.

Timing

Two 4-bit counters are used to clock data in both directions. *Bit Sample Count(bsc)* is triggered when the *start bit* is detected. *bsc* indicates whether we are looking at the start(0000), middle, or end (1111) of a bit. We choose to sample the 1-bit data at the halfway point, in other words, when *bsc* is 1000. *Bit Identification Count (bic)* is triggered when the first bit is received/sent, in other words, when *bsc* is 1000. There are 10 bits in each frame, the *bic* starts from 0000, 0001, 0010, ..., and starts again at 1111. When *bic* reaches 1010, a flag will be set up to tell the microprocessor that all 10 bits of data are received/sent.

Receive Data

Start Bit Detect - Transmit Enable

input - reset, clock, serial data input, character received.

output - receive start.

This logic block identifies the start of an input data frame. When it detects a '0' from the serial data input, it will generate a *receive start* signal to the control logic which starts counting the input data bits. It will stop the *receive start* signal once the 10 bits of the frame have been received, or when the whole system is reset. In other cases, the *receive start* signal will remain at its current state.

Bit Sample Count (bsc) - srClock

input - reset, clock, enable.

output - source clock.

register - (4 bit) bit sample count.

As noted before, *bsc* is used to keep track of whether we are looking at the start or end of the incoming bit. *bit sample count* is a 4-bit counter which starts from 0000 and ends at 1111. It will start counting when it receives the *enable* (receive start) flag. The *source clock* will be set to 1 every time when *bit sample count* counts to 1000, and it will be 0 in all other cases.

Bit Identification Count Receive (bic) - charReceived

input - reset, clock, source clock, receive enable.

output - character received.

register - (4 bit) *bit identification count*.

As noted before, *bic* is used to keep track of which bit it is present at the input frame. It will start counting at the *receive enable* signal. Starting from 0000, the *bit identification count* counter will increment at each *source clock* while *receive enable* is true. The value of the *bic* counter represents how many bits have been received by the microprocessor. When *bit identification count* counts to 1010 (which is 10 in decimal), the *character received* flag will be turned to 1, it will remain 0 in all other cases.

Serial to Parallel Buffer and Shift Register

input - *reset, source clock, serial data input*.

output - (8 bit) *parallel data*.

register - (10 bit) *buffer*.

This serial to parallel shift register propagates the *serial data input* value to the left at each *source clock*. A 10-bit *buffer* is used to keep track of the *parallel data input*. The *start bit* and *end bit* of the parallel data bus is neglected because they do not contain useful information. The 8 bits in the middle are passed to the microprocessor.

Nios II Processor

input - (8 bit) *parallel data bus, character received*.

The microprocessor reads the parallel data bus when the character received flag is set to 1. This 8-bit parallel data bus is the binary Ascii code for one character. The c 'putchar()' function can print the data bus out as characters. For example, the ascii code for A is 0097, which is 01100001 in binary.

Transmit Data

Nios II Processor

input - *character sent*.

output - (8 bit) *parallel data bus, load, transmit enable*.

When user types characters in the Nios II console, the c function 'getchar()' reads the character and sends it in binary Ascii format. At the moment the frame of bits is sent, the processor also generates a *load* signal, which notifies the buffer to load the new data. Similarly, the *Transmit Enable* signal is sent to the *bsc* block and the *bic transfer* block to indicate the start of the transmission.

Bit Sample Count (bsc) - srClock

input - *reset, clock, enable*.

output - *source clock*.

register - (4 bit) *bit sample count*.

The *bsc* module used for transmission is the same as the one used for receive. *bsc* is used to keep track of whether we are looking at the start or end of the incoming bit. *bit sample count* is a 4-bit counter which starts from 0000 and ends at 1111. It will start counting when it receives the

enable (receive start) flag. The *source clock* will be set to 1 every time when *bit sample count* counts to 1000, and it will be 0 in all other cases.

Bit Identification Count Transfer (bic) - charSent

input - *reset, clock, source clock, transfer enable, load.*

output - *character sent.*

register - *(4 bit) bit identification count.*

bic is used to keep track of which bit it is present at the transmit frame. It will start counting at the positive *transfer enable* signal. Starting from 0000, the *bit identification count* counter will increment at each *source clock* while 1) *transfer enable* is true and 2) the shift register is not loading (negative *load*). The value of the *bic* counter represents how many bits have been sent by the microprocessor. When *bit identification count* counts to 1010 (which is 10 in decimal), the *character sent* flag will be turned to 1, it will remain 0 in all other cases.

Parallel to Serial Buffer and Shift Register

input - *reset, source clock, load, transfer enable, clock, character sent, [7:0] data.*

output - *serial data output.*

register - *(10 bit) buffer.*

When *load* is positive and *character sent* is false, the 8-bit parallel data is loaded into a 10-bit buffer, with a *start bit* - 0, and an *end bit* - 1. This parallel to serial shift register pushes the parallel data out from the left one bit at each positive source clock edge. If the *character sent* flag is positive, the buffer will be set to its default, which is 10 bits of 1.

4. Software Description

The software part of the communication system is a simple C program that runs on the NIOS II microprocessor. The C program provides the basic MAC logic for the duplex communication system. The fundamental logic of the C program is an always active while loop that constantly check for user input. If user input is provided, the MAC logic inside the while loop process the input immediately and deliver output to the FPGA side. When no user input is provided, the MAC logic listen to its parallel input data bus to check for received data from the FPGA side. Once data received from the FPGA side, the data will be processed and printed to the console.

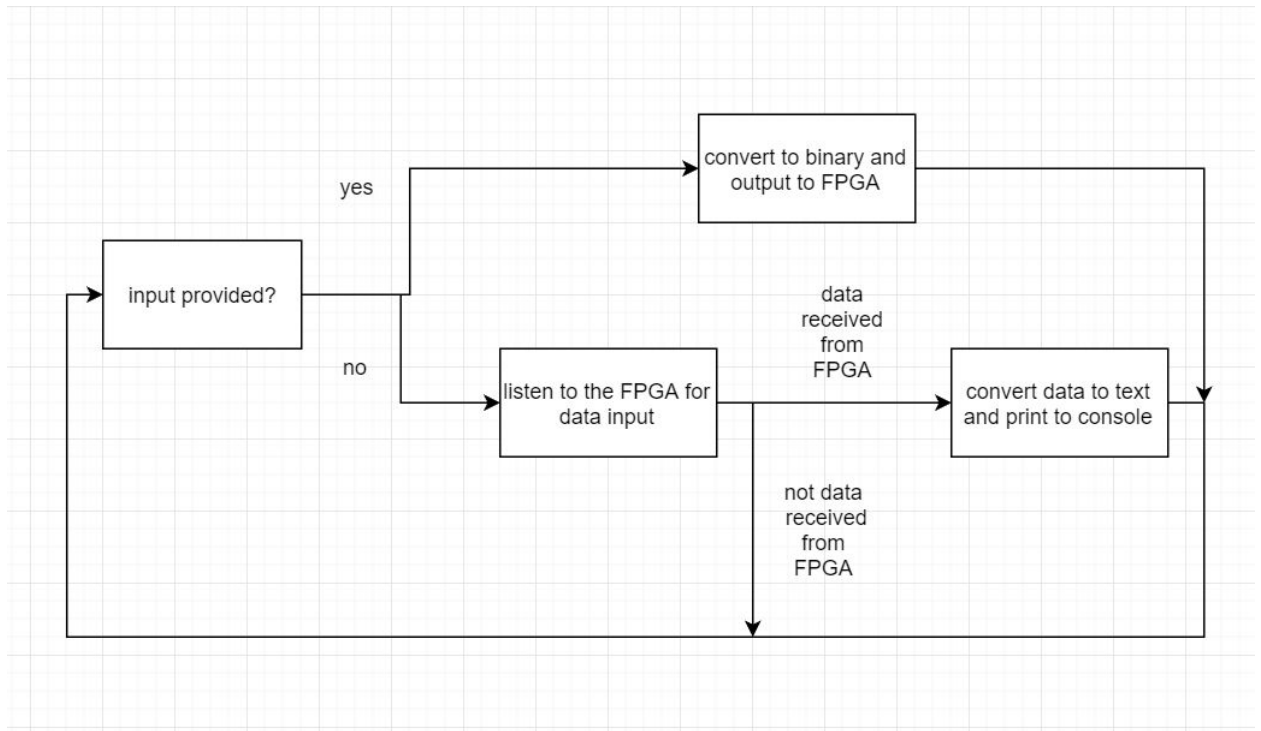


Figure 4 Flowchart of the MAC logic provided by software.

This MAC logic program essentially provides half-duplex functionality to the communication system. However, because the microprocessor is supplied with a clock source that is much higher in speed than the FPGA system, it processes input and output data fast enough to achieve full-duplex in most case scenarios.

5. Presentation and Results

5.1 Simulation of the Hardware Components

Simulation of submodules of the FPGA design were done with iVerilog and GTKwave.

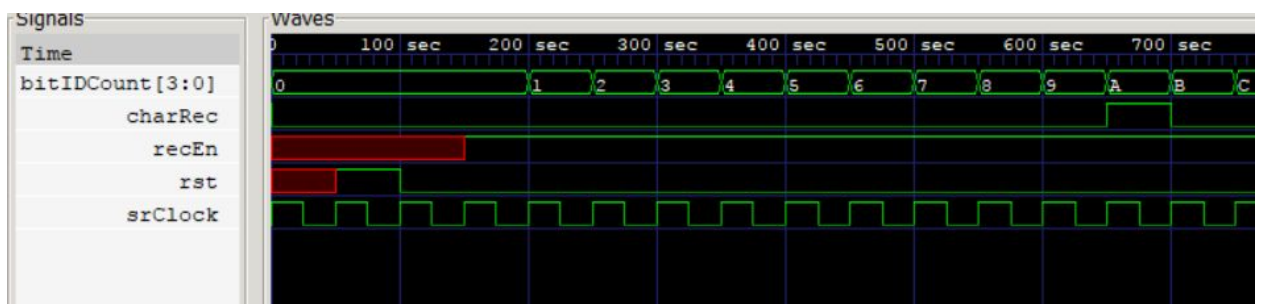


Figure 5 GTKwave of charRec signal simulation

This simulation validate the function of the bicReceive module and mainly test for the behavior of the output signal charRec. The module properly counts the number of bits received since activation of recEn signal and output active charRec signal for one cycle after receiving a

complete frame.

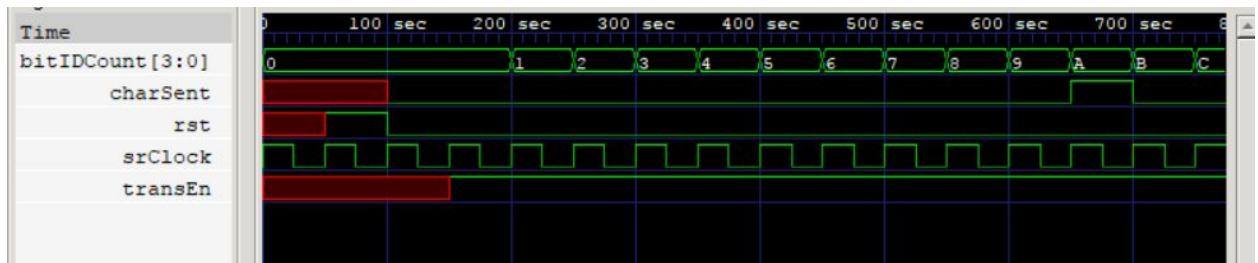


Figure 6 GTKwave of charSent signal simulation

This simulation validate the function of the bicTransmit module and mainly test for the behavior of the output signal charSent. The module properly counts the number of bits sent since activation of transEn signal and output active charSent signal for one cycle after sending a complete frame.

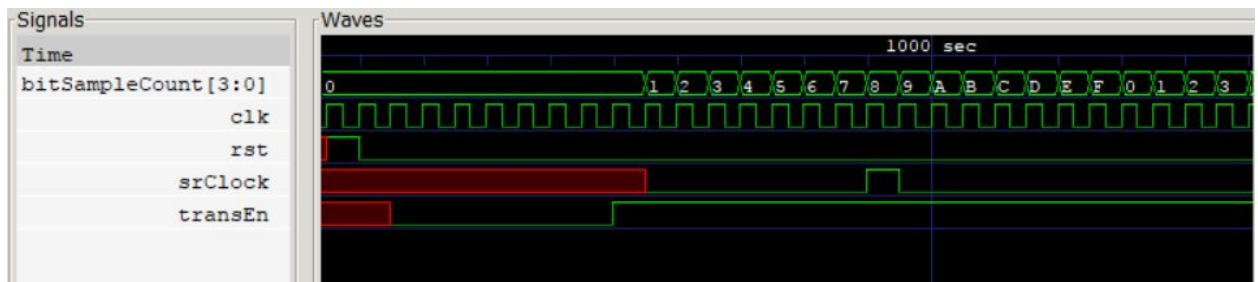


Figure 7 GTKwave of bscCounter simulation

This simulation validate the function of the bscCounter module and mainly test for the behavior of the output signal srClock. The module properly counts from 1 to 16 since activation of transEn signal and output active srClock signal for one cycle when the count reaches 8 (mid point).

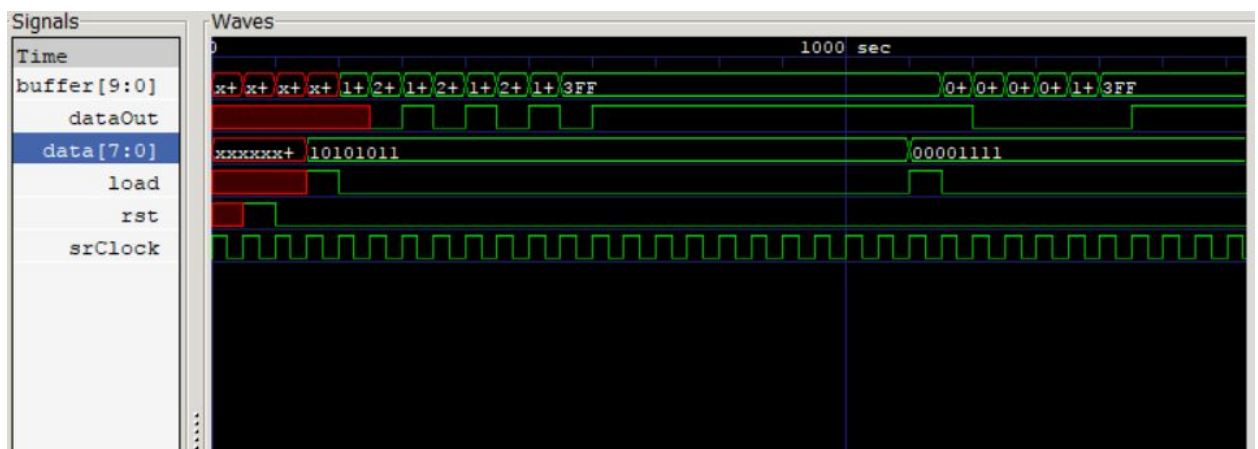


Figure 8 GTKwave of P2S_buffer simulation

This simulation validate the function of the P2S_buffer module. The module properly loads the buffer with supplied data when load signal is activated and pushes the bits in the buffer out in serial through dataOut.

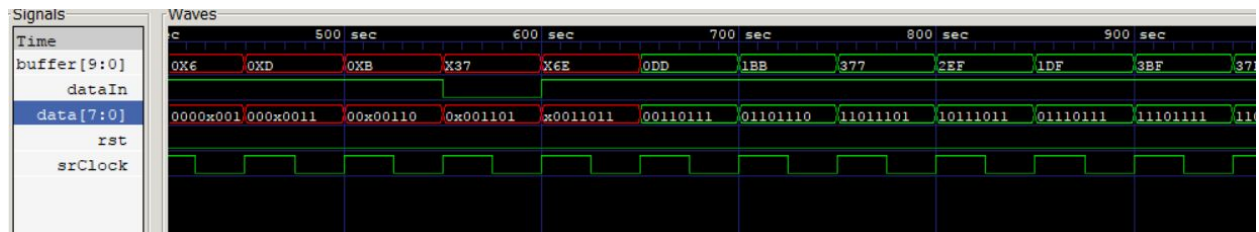


Figure 9 GTKwave of S2P_buffer simulation

This simulation validate the function of the S2P_buffer module. The module properly updates the parallel buffer and data every cycle of the srClock with the value from dataIn.

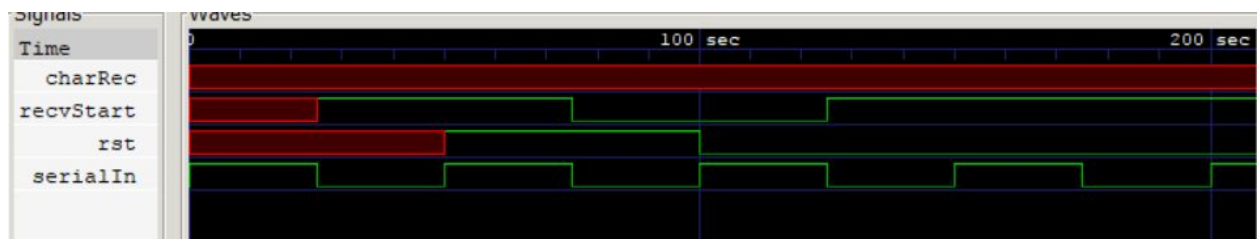


Figure 10 GTKwave of startBitDetection simulation

This simulation validate the function of the startBitDetection module and mainly test for the behavior of the output signal recvStart. The module properly set the recvStart signal to active one cycle after receiving the beginning bit of a frame (1) from serialIn.

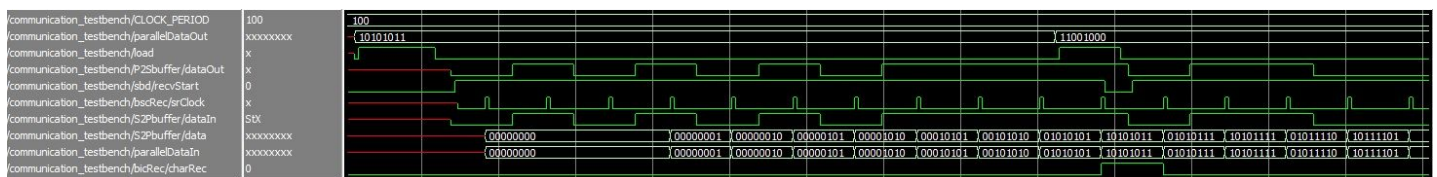


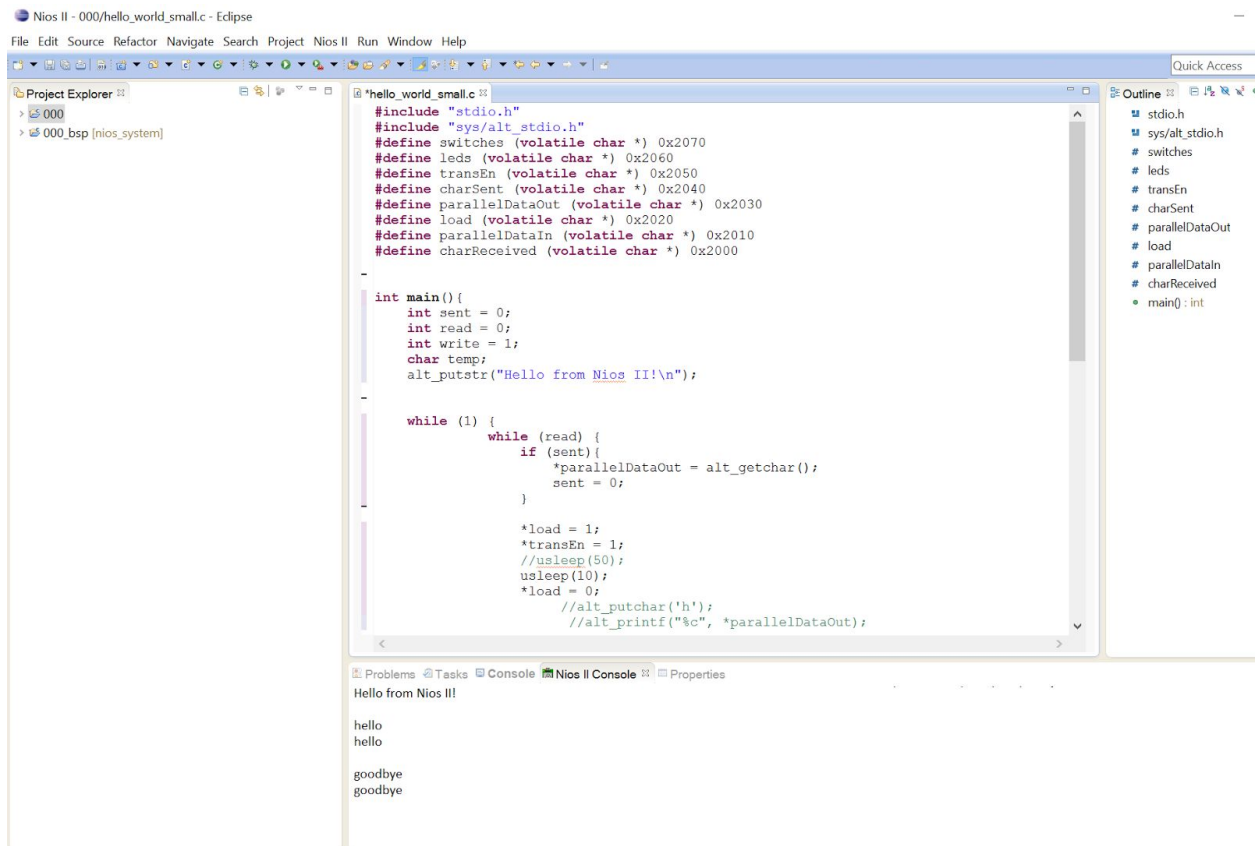
Figure 11 GTKwave of overall FPGA design simulation

This simulation validate the function of the overall FPGA design and mainly test for sending and receiving functions. To test the communication functions in one top module, the serial input and output signals and connected together. The load signal is properly updated to active for one cycle every time a frame is sent, and the dataIn signal properly react to the output data in serial. The charRec signal also properly update to active for one cycle every time the system receives a frame from the loopback.

After successful simulations, test with hardware were implemented. We first tested the functionalities using only one board with the serial output and input ports connected with a wire.

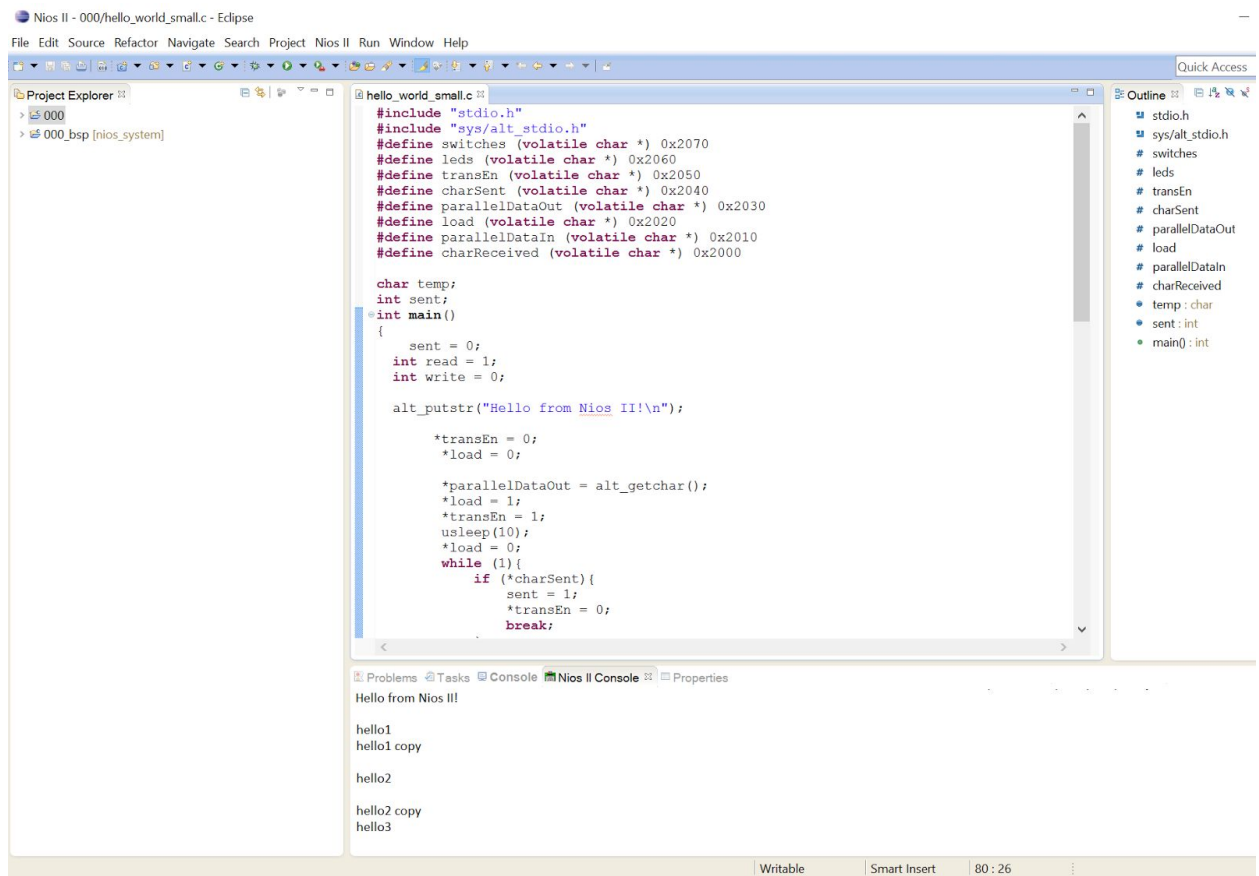


This waveform shows that the load signal was properly updated for every character sent from the microprocessor, transmit buffer in the FPGA design were properly updated, and the dataOut signal is also properly updated in a serial format of the buffer.



Once the single board is able to successfully send and receive messages through loopback,

two DE1 boards with identical design programed were used for further tests. The serial output port of one system is connected with wire to the serial input port of the other system.



The screenshot shows the Eclipse IDE interface for a Nios II project. The main editor window displays the source code for `hello_world_small.c`. The code includes standard headers, defines hardware registers, and implements a `main` function that prints "Hello from Nios II!\n" and then enters a loop that reads and writes data to hardware registers. The `Console` window at the bottom shows the output of the program, which includes the greeting and several "hello" messages. The `Outline` window on the right lists the symbols in the file, including the `main` function.

```
#include "stdio.h"
#include "sys/alt_stdio.h"
#define switches (volatile char *) 0x2070
#define leds (volatile char *) 0x2060
#define transEn (volatile char *) 0x2050
#define charSent (volatile char *) 0x2040
#define parallelDataOut (volatile char *) 0x2030
#define load (volatile char *) 0x2020
#define parallelDataIn (volatile char *) 0x2010
#define charReceived (volatile char *) 0x2000

char temp;
int sent;
int main()
{
    sent = 0;
    int read = 1;
    int write = 0;

    alt_putstr("Hello from Nios II!\n");

    *transEn = 0;
    *load = 0;

    *parallelDataOut = alt_getchar();
    *load = 1;
    *transEn = 1;
    usleep(10);
    *load = 0;
    while (1){
        if (*charSent){
            sent = 1;
            *transEn = 0;
            break;
        }
    }
}
```

Console Output:

```
Hello from Nios II!
hello1
hello1 copy
hello2
hello2 copy
hello3
```

Figure 14 Console of board number 1

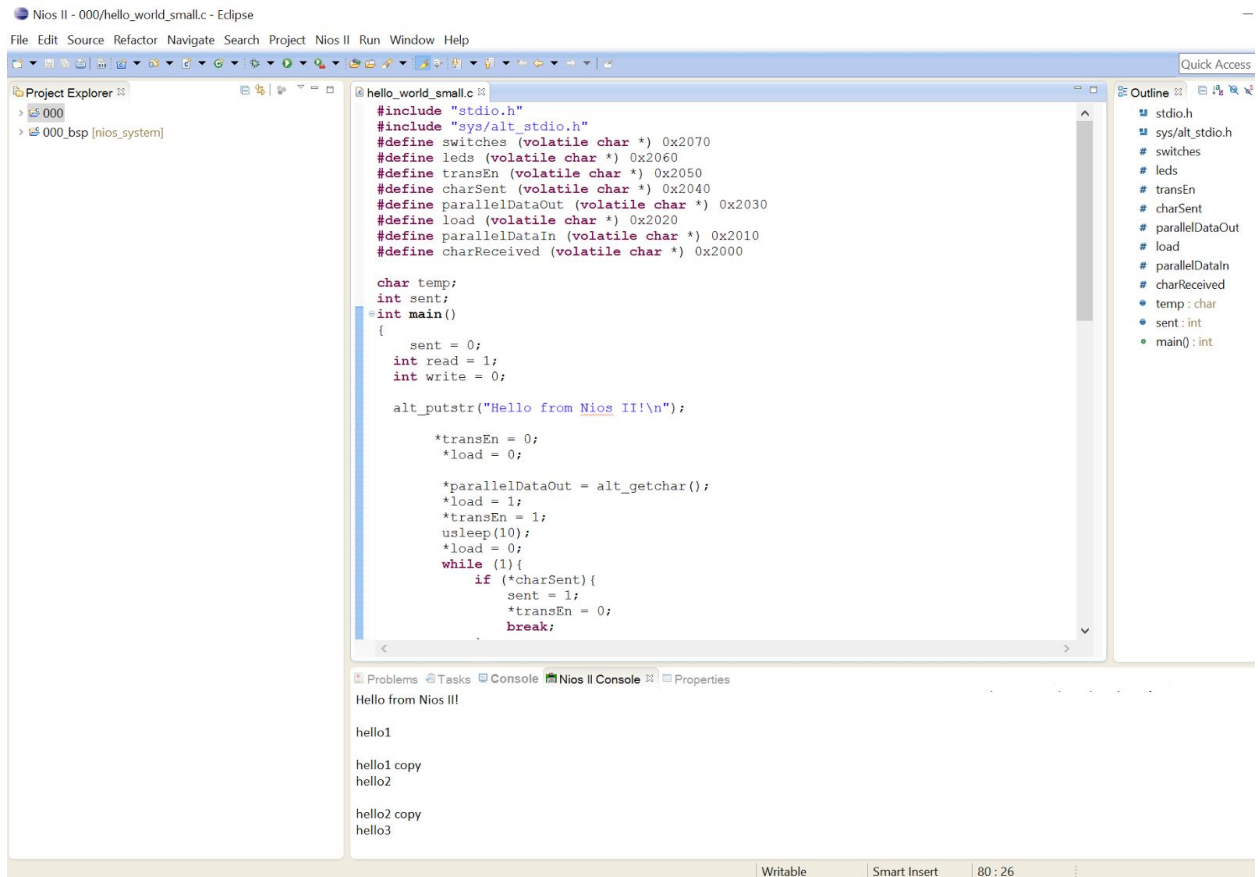


Figure 15 Console of board number 2

6. Error/ Failure Analysis

One of the most frustrating problem encountered during the development of this communication system is working with the NIOS II microprocessor. The error of failed to load EFL file happened multiple times when trying to run the C code on the hardware. The first few times of the error were caused by improper reset in the FPGA design and were easily fixed by going through the verilog codes and correcting for unexpected behavior of the reset signal. The cause of the last occasion of the very same error was not clear. The problem occurred after a successful run without any modification to the verilog codes. However, we were able to eliminate the problem by rebuilding the project base. Possible cause for the last occasion could be unintentional change to one of the microprocessor setting file.

Another error that lead to possible failure is to use the same clock source for both the microprocessor and the FPGA design. When the clock source for these two component are the same, very unstable and unpredictable behavior of the load signal was observed using SignalTap. Analysis for this problem is that the C program requires higher clock speed to implement the much more complicated structure in instructions. When using the same clock source with the FPGA, the microprocessor simply does not have enough speed to run all the instructions that are required to process one frame of data within the same time that the FPGA process and transmit one frame of data.

The hardware connection through the GPIO socket could also be a source of failure. The

serial output data were not able to be detected at all in the beginning of our hardware test. The problem was caused by connecting the wrong pin in the GOIP socket. The prototype board that came with the DE1 board has less pin in its GOIP connector, and it was initially connected to the GPIO port on the DE1 board one row misaligned. The problem was discovered by looking up the pin layout of the DE1 GPIO port from the manual. Also, the connection port on one of the board was very loosen and does not hold the connection wire very stable. Failures occurred multiple times during our test because of disconnection.

One last error that is worth mentioning is overloading the C program. The LED function that indicates the state of input and output buffer of the FPGA were added the last after all the major communication functionalities were implemented and tested. Because we have integrated the LED signals into the microprocessor, the intuitive solution to add the LED function was to control the signals in the C program. However, it seemed like the C program was overloaded with the extra LED function and not able to keep up with the FPGA side in terms of processing speed. The same situation also happened when trying to use complicated data storage method in the C codes. The solution to the C overloading problem is to optimize code efficiency. For the LED function specifically, our final solution was to implement it in hardware (verilog and FPGA) instead. Extra dangling wires were created to connected with the microprocessor as pseudo LED signals to avoid multiple assignment for the actual LED signals.

7. Summary and Conclusion

In this lab, students learned and practiced development of project with microprocessor and software. To be specific, this lab provided the chance for student to practice designing a communication system with FPGA and the embedded NIOS II microprocessor. Fundamental topics of this lab includes design of basic finite state machines, use of registers, HDL modeling, general purpose I/O of systems, integration of microprocessor, parallel and serial communication methods and protocols, and simple software MAC logic design. The project created in this lab is a serial transmission communication system including parallel communication between the FPGA and microprocessor with MAC logic defined in a C program. The final application provides bi-directional communication between two instances of the same system and allows users to chat back and forward in text inside the console.

8. Individual Contribution

Name	Contribution
Liyuan Wang	RTL design & Test & Verification
Jiaqi Zhang	RTL Design & Test & Design Specification
Zhouyuan Xu	Test & Lab Report & Simulation

Appendices

Appendix A. ASCII Binary Table

Letter	ASCII	Binary	Letter	ASCII	Binary
a	097	01100001	A	065	01000001
b	098	01100010	B	066	01000010
c	099	01100011	C	067	01000011
d	100	01100100	D	068	01000100
e	101	01100101	E	069	01000101
f	102	01100110	F	070	01000110
g	103	01100111	G	071	01000111
h	104	01101000	H	072	01001000
i	105	01101001	I	073	01001001
j	106	01101010	J	074	01001010
k	107	01101011	K	075	01001011
l	108	01101100	L	076	01001100
m	109	01101101	M	077	01001101
n	110	01101110	N	078	01001110
o	111	01101111	O	079	01001111
p	112	01110000	P	080	01010000
q	113	01110001	Q	081	01010001
r	114	01110010	R	082	01010010
s	115	01110011	S	083	01010011
t	116	01110100	T	084	01010100
u	117	01110101	U	085	01010101
v	118	01110110	V	086	01010110

w	119	01110111	W	087	01010111
x	120	01111000	X	088	01011000
y	121	01111001	Y	089	01011001
z	122	01111010	Z	090	01011010