EE 371 Lab 5
Building and Working with a Simple Microprocessor and Building an Application

Sailesh Suri
(Michael) Wenbing Zhang
Boren Li

The contents of this report is solely the work of the team members below, and the report cites outside references where applicable.

Boren Li

Wenbing Zhang

Sailesh Suri

TABLE OF CONTENTS

I.	ABSTRACT	2
II.	INTRODUCTION	2
III.	SYSTEM DESCRIPTION FOR Lab 4	3
IV.	DESIGN SPECIFICATIONS	4
V.	DESIGN PROCEDURE	5
VI.	SYSTEM DESCRIPTION	6
VII.	SOFTWARE IMPLEMENTATION	12
VIII.	HARDWARE IMPLEMENTATION	13
IX.	TEST PLAN	19
X.	TEST SPECIFICATIONS	21
XI.	TEST CASES	23
XII.	PRESENTATION OF RESULTS	30
XIII.	ANALYSIS OF ERROR	35
XIV.	SUMMARY	35
XV.	CONCLUSION	36
XVI.	CHART OF CONTRIBUTIONS	36
XVII.	APPENDICES	37

ABSTRACT

In lab 4 & 5, we work with Qsys to design, implement, and test a simple microprocessor based upon the Altera NIOS II core on the DEI-Soc board FPGA. We add GPIO(general purpose I/O) on the microprocessor to enable information exchange within and external to the FPGA. We also put our C language knowledge into play by using Eclipse based NIOS II IDE to develop applications that run on the microprocessor. We start our lab by building small applications, that are, *Count Binary*, *lights and switches*, and *hello world small*. We, then, integrate the scanner subsystem into the microprocessor to allow the system to receive and send the data between scanner and the station by developing and testing an asynchronous serial network.

INTRODUCTION

In lab4, we used Qsys software to create a NIOS II processor that is capable of executing C program through Eclipse for NIOS II processor. The processor consist of a processor core, on-chip memories, and Parallel IOs to interact with the peripherals. To test the functionality of the processor, three sample programs was made as mentioned in the abstract. In lab5, we created an asynchronous receiver and transmitter system in order for the NIOS II processor to send and receive data from other devices and network that follow the RS232 protocol. In addition, we improved the scanner subsystem from lab5. The scanner include a buffer that holds byte wide 10 entries memory each, and is able to receive and transfer data through the parallel IO bus. Many command and flag lines are also exported to grant more control of the system to an master device. In this case, we expand our NIOS II processor in order to control both system with C program. Three programs were made to demonstrate the functionality of the system: scanner_intergration, transmit and receive. The respectively test the function of the scanner communication with the Nios II processor, function of the system act as slave scanner taking command through the network and return data, and function of the system act as master station, commanding an remote scanner through the network and acquires data.

SYSTEM DESCRIPTION FOR Lab 4

We began this lab by building a simple NIOS II microprocessor using the Altera Qsys system integration tool in Quartus. We then used this microprocessor to run simple C programs.

Microprocessor

The microprocessor consists of 2 parallel input/outputs that can be used by the software implemented onto the NIOS II processor.

Inputs:

- Clk: 50 MHz system clock.
- Reset: System reset
- Switches: This is an 8-bit wide PIO (parallel input/output) representing Switches 0 through 7 on the DE1-SoC board.

Outputs:

- Leds: An 8-bit wide PIO representing LEDRs 0 through 7 on the DE1-SoC board.

Software

We ran the three following simple software on our NIOS II microprocessor:

1. Count Binary: This C program uses our microprocessor to output an incrementing count from 00 to ff in hexadecimal on the Eclipse NIOS II console.
2. Lights: This C program uses our switch and LED GIPOs and maps each LEDR to its corresponding switch on the DE1-SoC board. The end result is being able to turn on and off LEDRs 0 through 7 using Switches 0 through 7 respectively.
3. Hello World Small: This C program outputs a Hello World message on our Eclipse NIOS II console and then waits for user input. If the user inputs character ‘g’, the program will run the “Lights” program which can be reviewed in detail above.

DESIGN SPECIFICATION

The system can be divided into three parts: Nios II processor, scanner subsystem, and the Parallel-Serial-Parallel network system. The following discuss the specification of the three parts respectively.

Nios II processor: The Nios II processor controls both the scanner subsystem and the network system as well as acquire and sending data from and to them. The processor is programmable through Eclipse for Nios II using C.

Scanner subsystem: The scanner subsystem contains two scanners that is able to read and hold the data, and transfer the data on command of the processor. Each scanner contain a 10-entry byte wide buffer to hold the scanned in data. The behavior of the scanner subsystem is as follow.

1. When StartScan signal is received , the first scanner start to take in data and store them into the buffer.
2. When the buffer is 80% full, flag high Ready_to_Transfer signal to the processor, and send Ready_Second_Scanner signal to the other scanner, which should exit Low Power state and enter Standby state.
3. When receive a Transfer signal, the first scanner will transfer its data to the processor consecutively.
4. When the buffer is 90% full, signal StartScan signal to the other scanner and it starts taking in data.
5. When the buffer is 100% full, the scanner goes into Idle stage and wait for a Transfer signal
6. When the second scanner is 50% full, the first scanner Flush all its data and enter Low Power state.
7. This procedure is then alternately executed by the scanners.

In this case, the data for the scanners are provided by the Nios Processor with the C program. We write 0 to 9 to buffer 0 to 9 to test functionalities.

Network system: The network system is responsible for sending and acquiring data through a serial network with other devices. When sending data, it converts the parallel byte wide data to serial data and send it to the network. When receiving data, it converts the serial data into a parallel byte wide data for the processor to read. The Network system have the following specifications.

1. The serial data follows RS232 protocol.
2. The baudrate of sending and receiving is 9600 bps.
3. The bits are sampled at 16x clock.
4. Loading data and transmit enable are controlled by the processor.
5. The system notify the processor when a byte of data is sent or received.

The three parts are then integrated, and C programs are developed to synergize the functionality of the entire network enabled scanner control and data system.

DESIGN PROCEDURE

NIOS II CPU

We work with Qsys to develop and test the microprocessor. We first pick the NIOS II/e core as the centerpiece of the processor. Then, in order to work with Eclipse and the library it provides, we selected 20KB of on-chip memory to the RAM of the system. We then add the parallel I/O (PIO) in the microprocessor, setting up the width and direction of each PIO, according to the utilization and requirements of the scanner subsystem and the network system .We make sure that clock input and reset input of each PIO are in the same phase with that of the clock source. After assigning base address to the processor, we then use the Generate HDL (verilog) in the toolbox of Qsys to model the microprocessor.

Scanner subsystem with buffer modification

The general logic of the scanner subsystem inherited the design from Lab 3, with the only difference that the buffers are now real memories with 10 Byte of memory space instead of counters. The buffer is designed with two address counters: write address and read address. The two address counter dictate which memory address to write to or read from as well as indicate the fullness of the buffer through the scanning and transferring stage. We designed the buffer to store data from the data input buss at the proper address indicated by the counter at every posedge of write enable, and the counter would increment at every negedge of write enable. Therefore a consecutive writing process is accomplished. The buffer will also always output data according to the read address counter, and will increment every posedge of read increment signal. On negedge of reset signal, all data stored in the buffer are discarded and both counter will be reset at address zero.

Network system

In this lab,we add serial-parallel-serial network interface to the project. This system supports asynchronous full duplex data transmission and reception. We divide the network system into two parts, receive part (data propagation serial to parallel) and transmit part (data propagation

parallel to serial). Two counters, bit identifier count (BIC) and bit sampling count (BSC) are used to keep track of the incoming and outcoming serial data. On the receive side, on each transition, the bit is sampled at the halfway point. On the transmit side, the entire frame of data is shifted out serially. On both sides, the clock control controls when the data is clocked.

Software and Top level Integration

In the first phase of this lab, we wrote a C program for the microprocessor to act as the scanner subsystem controller and the canal-side tracking station. Therefore, our microprocessor will be sending data to the scanner subsystem, and receiving the transmitted data from the scanners. This requires our microprocessor to be able to send the scanner subsystem a *start_scan* signal and a *transfer* signal, while waiting to receive a *ready_to_transfer* signal from the scanner subsystem.

In the second phase, our board will be connected to another DE1-SoC board. We will alternate the roles of the station and the canal scanner for each board. The canal scanner board will await a *start_scan* signal from the station board. Once the scanner board receives this signal, it will start scanning and collection data. Once it has reached 80% capacity, the scanner board will send out a *ready_to_transfer* signal to the station board. Upon receiving this signal, the station board will issue a *transfer* signal to the scanner board and the scanner board will begin transferring its data to the station board.

SYSTEM DESCRIPTION

NIOS II CPU

We integrate the scanner system in the microprocessor. We add the input, output and wire in the scanner system to the PIO so that we can build the scanner system into the microprocessor.
Input:

- cpu_data_in_0: 8-bit data output bus for the first scanner buffer
- cpu_data_in_1: 8-bit data output bus for the second scanner buffer
- Ready_to_transfer_in_0: ready to transfer signal output for the first scanner buffer. Will signal high when buffer is more than 80% full.
- Ready_to_transfer_in_1: ready to transfer signal output for the second scanner buffer. Will signal high when buffer is more than 80% full.
- char_sent: 1 bit signal to indicates that 1 bit of parallel data is sent out.
- char_received: 1 bit signal to indicates that data is received.
- net_data_in: 8-bit data received by the CPU.
- start_scan_receive: 1-bit signal allowing our scanner to begin scanning. Received from another DE1-SoC board acting as the station.

- ready_transfer_receive: 1-bit signal indicating to our station that the network scanner is ready to transfer. Received from another DE1-SoC board acting as the scanner.
- Transfer_receive: 1-bit signal indicating to our scanner to transfer its data to the station. Received from another DE1-SoC board acting as the station.

Outputs:

- cpu_data_out_0: 8-bit data input bus to the first scanner
- cpu_data_out_1: 8-bit data input bus to the second scanner
- Start_scanning: 1-bit signal that starts scanner 1.
- start_transfer:1-bit signal that permits either scanner to begin transferring data to the station.
- scanner_rst: KEY0. Reset scanner and dump collected data.
- wr_en1: 1-bit signal required to be pulsed to allow scanner 1 to write data to its buffer.
- wr_en2: 1-bit signal required to be pulsed to allow scanner 2 to write data to its buffer.
- read_inc1: 1-bit signal required to be pulsed to allow scanner 1 to transfer data from its buffer to the station.
- read_inc2: 1-bit signal required to be pulsed to allow scanner 2 to transfer data from its buffer to the station.
- Transmit_enable: 1-bit signal indicating to the ART that the microprocessor is transmitting data.
- Load: 1-bit signal indicating to the ART that it should load parallel data in.
- net_data_out: 8-bit data sent by the CPU
- Led_data: 8-bit signal that connected to the leds on the board
- Start_scan_send: 1-bit signal commanding the scanner system to begin scanning. Sent to another DE1-SoC board acting as the scanner.
- Ready_transfer_send: 1-bit signal indicating to the station that the scanner is ready to transfer. Sent to another DE1-SoC board acting as the station.
- Transfer_send: 1-bit signal indicating to the scanner to transfer its data to our station. Sent to another DE1-SoC board acting as the scanner.

Scanner subsystem with buffer modification

In order to grant more control of the scanner system to the NIOS II processor and have it work more precisely and friendly with the C programs, we added more control input to the scanners compare to previous design. The following is a list of all the IOs of the system.

Inputs:

- data_in1: 8-bit data input bus to the first scanner.
- data_in2: 8-bit data input bus to the second scanner.
- start_scan: 1-bit start scan signal line.
- transfer_input: 1-bit Transfer signal line
- wr_en1: write enable line for the first scanner. Scanner register data on the data bus on posedge of the signal and increment write address counter on negedge.
- wr_en2: write enable line for the first scanner. Scanner register data on the data bus on posedge of the signal and increment write address counter on negedge.
- read_inc1: 1-bit signal to increment the reading address counter of the first scanner buffer. Address increment on the posedge of the signal
- read_inc2: 1-bit signal to increment the reading address counter of the second scanner buffer. Address increment on the posedge of the signal
- clk: system clock input
- rst: scanner subsystem reset, Active Low.

Outputs:

- data_out1: 8-bit data output bus for the first scanner buffer. Will always display the data of the buffer memory address at the value of the reading address counter.
- data_out2: 8-bit data output bus for the second scanner buffer. Will always display the data of the buffer memory address at the value of the reading address counter.
- data_out_cpu1: same function as data_out1, reserved to connect to NIOS II processor.
- data_out_cpu2: same function as data_out2, reserved to connect to NIOS II processor.
- state: 3-bit indicator showing to current state of the first scanner. Connected to LEDs.
- state2: 3-bit indicator showing to current state of the second scanner. Connected to LEDs.
- ready_to_transfer: ready to transfer signal output for the first scanner buffer. Will signal high when buffer is more than 80% full.
- ready_to_transfer: ready to transfer signal output for the second scanner buffer. Will signal high when buffer is more than 80% full.

Network System

On transmit side, we implemented three modules. Module serial_buffer uses parallel to serial shift register to shift out the entire frame of parallel data serially. Module serial_out_ctrl sends *char_sent* signal back to the CPU when data is sent out. It also sends *SR_clk* signal to the serial_buffer module. Module serial_out makes serial_buffer module and serial_out_ctrl module work together in order to shift the stored parallel data out serially when the data is clocked and send signal back to the CPU when data is sent out.

serial _buffer

Inputs

- rst: 1 bit signal to rest the each bit of data in the parallel_in reg to be 1 when 1 bit load signal is received.
- sr_clk: 1 bit active high clock control signal.
- parallel_in: 8 bit parallel data
- load: 1 bit signal that fill the reg which contains 8 bit parallel data that is ready to sent out serially.

Outputs

- bit_out: 1 bit data is sent out when the parallel data is clocked.

serial_out_crtl

Inputs

- rst: reset the tBase counter
- clk: standard clock signal coming from the top level module.
- Trans_en: 1 bit signal to enable the BIC and BSC counter

Outputs

- SR_clk: 1 bit active high clock control signal.
- char_sent: 1 bit signal to indicates that 1 bit of parallel data is sent out.

serial_out

Inputs

- rst: reset the tBase counter
- clk: standard clock signal coming from the top level module.
- trans_en: 1 bit signal to enable the BIC and BSC counter
- load: 1 bit signal that fill the reg which contains 8 bit parallel data that is ready to sent out serially.
- data_in: 8 bit parallel data that is ready to sent out serially.

Outputs

- bit_out: 1 bit data is sent out when the parallel data is clocked.
- char_sent: 1 bit signal to indicates that 1 bit of parallel data is sent out.

On receive side, we implemented three modules. Module parallel_buffer uses serial_to_parallel shift register to propagate the serial input data when the data is clocked . Module serial_in_ctrl sends *char_received* signal to the CPU when parallel data is received by the CPU. It also sends *SR_clk* signal to the parallel_buffer module. Module serial_in makes parallel_buffer module and serail_in_ctrl module work together in order to received parallel data when the data is clocked and send signal to the CPU when parallel data is received.

parallel _buffer

Inputs

- rst: 1 bit signal to rest the each bit of data in reg to be 1
- sr_clk: 1 bit active high clock control signal.
- ser_data_in: 1 bit data that is received

Outputs

- parallel_out: 11 bit data

serial_in_crtl

Inputs

- rst: reset the tBase counter
- clk: standard clock signal coming from the top level module.
- ser_data_in: 1 bit signal that indicates if there is data available to be received

Outputs

- SR_CLK: 1 bit active high clock control signal.
- char_received: 1 bit signal to indicates that data is received.

serial_in

Inputs

- rst: reset the tBase counter
- clk: standard clock signal coming from the top level module.
- ser_data_in: 1 bit signal that indicates if there is data available to be received.

Outputs

- data_out: 8 bit data is received by CPU when the serial data is clocked.
- char_received: 1 bit signal to indicates that data is received.

Software and Top Level Integration

The following is a detailed breakdown of the top level C program's inputs and outputs:

Phase 1: scanner_integration.c

Inputs

- Data_in_0: 1-byte incoming data from scanner 1 to the station.
- Data_in_1: 1-byte incoming data from scanner 2 to the station.
- Ready_to_transfer_0: 1-bit signal coming from scanner 1 indicating it is ready to transfer.
- Ready_to_transfer_1: 1-bit signal coming from scanner 2 indicating it is ready to transfer.

Outputs

- Data_out_0: 1-byte data representing data collected by scanner 1.
- Data_out_1: 1-byte data representing data collected by scanner 2.
- Start_scanning: 1-bit signal that starts scanner 1.
- Start_transfer: 1-bit signal that permits either scanner to begin transferring data to the station.
- Wr_en_1: 1-bit signal required to be pulsed to allow scanner 1 to write data to its buffer.
- Wr_en_2: 1-bit signal required to be pulsed to allow scanner 2 to write data to its buffer.
- Read_inc_1: 1-bit signal required to be pulsed to allow scanner 1 to transfer data from its buffer to the station.
- Read_inc_2: 1-bit signal required to be pulsed to allow scanner 2 to transfer data from its buffer to the station.
- Scanner_rst: KEY0. Reset scanner and dump collected data.

Phase 2: receive.c and transmit.c

The inputs/outputs for this phase are identical to the ones in phase 1, with the following additions:

Inputs

- Char_sent: 1-bit signal indicating that we have sent data to the ART (asynchronous receiver/transmitter) system.
- Char_received: 1-bit signal indicating that we have received data from the ART system.
- Net_data_in: 1-byte GPIO data received during transmission from another DE1-SoC board.
- Start_scan_receive: 1-bit GPIO signal allowing our scanner to begin scanning. Received from another DE1-SoC board acting as the station.

- Ready_transfer_receive: 1-bit GPIO signal indicating to our station that the network scanner is ready to transfer. Received from another DE1-SoC board acting as the scanner.
- Transfer_receive: 1-bit GPIO signal indicating to our scanner to transfer its data to the station. Received from another DE1-SoC board acting as the station.

Outputs

- Transmit_en: 1-bit signal indicating to the ART that the microprocessor is transmitting data.
- Load: 1-bit signal indicating to the ART that it should load parallel data in.
- Net_data_out: 1-byte GPIO data transferred during transmission to another DE1-SoC board.
- Start_scan_send: 1-bit GPIO signal commanding the scanner system to begin scanning. Sent to another DE1-SoC board acting as the scanner.
- Ready_transfer_send: 1-bit GPIO signal indicating to the station that the scanner is ready to transfer. Sent to another DE1-SoC board acting as the station.
- Transfer_send: 1-bit GPIO signal indicating to the scanner to transfer its data to our station. Sent to another DE1-SoC board acting as the scanner.

SOFTWARE IMPLEMENTATION

In order to develop the C program to interact with the scanner subsystem and the network system, we used the Eclipse for NIOS II IDE to write and compile our C code. It also assist us to generate a Board Support Package (BSP) to have our C program to adapt on the NIOS II processor.

Because that the PIOs of the NIOS II processor are memory mapped IOs, we can address the PIO lines with hex memory address pointers. And all the following softwares utilize this characteristics of the NIOS II processor.

Phase 1: scanner_integration.c

This program begins by asking the controller if they would like to start the scanning subsystem via a prompt in the Eclipse NIOS II console. The controller can input a ‘y’ character for yes or a ‘n’ character for no. If the controller sends a start scan signal, the program will begin sending the scanner subsystem data from 0 to 9 and this data is stored in the scanner’s buffer system. Once this data has reached 80%, the controller will be prompted with the string “scanner 1 ready to transfer” and “start scanner 1 transfer? (y/n)”. It will then await user input. If the controller sends the scanner the transfer signal, the string “transferring...” will appear in the console and the program will start reading data from the scanner’s buffer. It will display this data in the console

as it is being read. The user will also be prompted if they would like to start the second scanner. A similar sequence of events will follow once the second scanner has begun.

Phase 2: receive.c and transmit.c

In the case of receiving data from another DE1-SoC board, our program will act as the station and send a *start_scan_send* signal to the scanner board via console i/o. While the scanner is scanning, our system's console will display a "waiting..." string, until we have received a *ready_transfer_receive* signal, indicating that the scanner board is ready to transfer. Our program will then prompt the user if the station should send a *transfer_send* signal to the scanner board. If the *transfer_send* signal is sent, our station will start receiving data from the scanner board via *net_data_in,* register the data consecutively into an array and display the received data in the console.

In the case of transmitting data to another DE1-SoC board, our program will act as the scanner system and wait for a *start_scan_received* signal. Once it has received this, the program will send a *start_scanning* signal to the scanner system and begin writing data to the scanner buffer. Once the buffer has reached 80% capacity, the program will send a *ready_transfer_send* signal to the station board. If our scanner system receives a *transfer_receive* signal, it will begin reading the scanner buffer, register all the data available into an array in the processor and transmitting the data out via *net_data_out* to the station board. The program will then instruct the CPU to signal the transmit part of the Network system to load the data on net_data_out, signal transmit enable and wait for the character to be sent. When a char_sent signal is received by the CPU, the CPU will load another byte of data to the transmitter. The process is repeated until all the data in the array are exhausted. If, however, the scanner system does not receive a *transfer_receive* signal, it will instead dump scanner 1's data and prompt the user to begin scanner 2's data collection and a similar sequence of events will follow.

HARDWARE IMPLEMENTATION

NIOS II CPU

The NIOS II processor is implemented with the assistance of the Qsys software. We first pick the NIOS II/e core as the centerpiece of the processor. Then, in order to work with Eclipse and the library it provides, we selected 20KB of on-chip memory to the RAM of the system. We then added the input, output, relative wires in the scanner system and network system to the Parallel I/O (PIO) in the windows of *System Contents*. Finally, we assigned the base address for all the memory space and the PIOs and used *Generate HDL* function in the toolbox to get the stub code. In this way, we can route PIOs to the NIOS II CPU. For instance,

```
nios_system u0 (
        .char_received_external_connection_export  (char_received),
        .char_received_external_connection.export    (char_sent),
        ……
);
```

In the fraction code above, we can see that *char_received* signal in the scanner system is connect its corresponding PIO in the NIOS II CPU.

Scanner subsystem with buffer modification

In order to realize true memory storage and data IO of the scanner system. We modified the buffer modules integrated with the scanners. To meet the specification, a 8-bit wide, 10 entry array of register/latch was declared inside of the buffer module. We have also declared two address counter for read and write purpose.

The output of data from the buffer is determined by the read address counter. The following verilog code illustrate the mechanism of it:

```
assign data_out = m[read_address]
```

The storage of data is determined by the data_in input and write address counter as well as the write enable input signal. On posedge of the write enable, the buffer will store the data on the data input bus to the address indicated by the counter, and on the negedge of the write enable signal, address counter will increment. This mechanism ensure consecutive writing to the buffer with minimal control effort. The following code illustrate the mechanism:

```
if (wr_en)
        m[address] <= data_in;
else if (~wr_en)
        address <= address + 1'd1;
```

However, due to the nature of processor signaling, the processor will not be able to give clock perfect signal to the buffer hardware. Therefore, an edge detector is implemented to ensure the integrity of the writing process. And the code above would only be activated on the edge of write enable signal. The edge detector is as follow:

```
always @(posedge clk) begin
        wr_en_delayed <= wr_en;
        wr_en_edge <= wr_en ^ wr_en_delayed;
end
```

Similar mechanism is implement for the read address counter. The read address will always increment on the posedge of the read increment signal.

To detect the scanner buffer fullness, the read and write address is used to indicate the status of the buffer. When the address is incremented to 7, it is indicated that the buffer is 80% full, and when it is 8, it is indicated that the buffer is 90% full, etc. The control logic of the two scanners are based upon it.

Network system

Transmit side has 3 modules, that are, *serial_out_cntrl, serial_buffer, serial_out.*

Serial_out_ctrl

Serial_out_ctrl contains two counters, BSC and BIC. When the trans_en signal is set to high, BSC increases by 1.

```
always@(posedge tBase[7]) begin
        if(~trans_en || ~rst) begin
                BSC <= 4'b0;
        end else if (trans_en) begin
                BSC <= BSC + 1'b1;
        end
end
```

When BSC reaches to 15, set SR_clk to high. If active low reset is set to 0, set SR_clk back to 0.

```
always@(posedge clk) begin
        if(~rst)
                SR_CLK <= 1'b0;
        else if (BSC == 4'b1111)
                SR_CLK <= 1'b1;
        else
                SR_CLK <= 1'b0;
end
```

When BSC reaches to d'15, trans_en signal is enabled, reset is set to inactive and BIC is less d'11, BIC increases by 1. When BIC reaches to 11 , set char_sent signal to high.

```
always@(posedge tBase[7]) begin
        if(~trans_en || ~rst)begin
                BIC <= 4'd0;
                char_sent <= 1'b0;
        end else if (BSC == 4'b1111 && BIC < 4'd11)
                BIC <= BIC + 4'd1;
        else if (BIC >= 4'd11) begin
                BIC <= BIC;
                char_sent <= 1'b1;
        end
end
```

serial_buffer

When the load signal is received by the serial buffer, output reg *bitshift* is filled with the input parallel data that is ready to be sent out serially. When the active low reset is set to high, output reg is filled with default data. On each sr_clk which is received from the serial_out_ctrl, the least significant bit is concatenated and a bit 1 is added to the most significant bit.

```
always@(posedge sr_clk or posedge load or negedge rst) begin
        if (~rst)
                bitShift <= 11'b11111111111;
        else if (load)
                bitShift <= parallel_in;
        else if (sr_clk)
                bitShift <= {1'b1, bitShift[10:1]};
end
```

When the data is clocked and active low reset is set to high, the least significant bit is shifted out. If active low reset is set to high, *1'b1* is shifted out.

```
always@(posedge sr_clk or negedge rst) begin
        if (~rst)
                bit_out <= 1'b1;
        else
                bit_out <= bitShift[0];
end
```

Serial_out

Format the input parallel data *data_in_full* by setting most significant bit to 1 , least significant bit to 0 and get the parity bit. *Serial_out* also works as the top module of transmit side.

```
assign parity =    (data_in[0] ^ data_in[1]) ^
                     (data_in[2] ^ data_in[3]) ^
                   (data_in[4] ^ data_in[5]) ^
                   (data_in[6] ^ data_in[7]);

assign data_in_full = {1'b1, parity, data_in, 1'b0};
```

Receive side has 3 modules, that are, *serial_in_ctrl, parallel buffer, serial_in.*

Serial_in_ctrl

If there is available bit input, CPU does not receive any bit, active low clk is set to high, set *en* (start_bit_ctrl) to true.

```
always@(posedge clk or negedge rst or posedge char_received) begin
        if(~rst)
                en <= 1'b0;
        else if (char_received)
                en <= 1'b0;
        else if (~ser_data_in)
                en <= 1'b1;
end
```

If the active low clk is set to high and *en* is set to high, increase the BSC by 1. Since the bit is sampled at the halfway point, when BSC reaches *4b'0111*, set SR_CLK to high.

```
always@(posedge tBase[7]) begin // set tBase[7] for 9600(12207)
baudrate
        if(~en || ~rst) begin
                BSC <= 4'b0;
        end else if (en) begin
                BSC <= BSC + 1'b1;
        end
end
```

```
always@(posedge clk) begin
        if(~rst)
                SR_CLK <= 1'b0;
        else if (BSC == 4'b0111)
                SR_CLK <= 1'b1;
        else
                SR_CLK <= 1'b0;
end
```

If BSC reaches to *4'b1111* and BIC is not up to *4'd11,* increases the BIC. If BIC reaches to *4'd11,* set *char_received* to high.

```
always@(posedge tBase[7]) begin
        if(~en || ~rst)begin
                BIC <= 4'd0;
                char_received <= 1'b0;
        end else if (BSC == 4'b1111 && BIC < 4'd11)
                BIC <= BIC + 4'd1;
        else if (BIC >= 4'd11) begin
                BIC <= BIC;
                char_received <= 1'b1;
        end
end
```

Parallel_buffer

When active low reset is set to high, set parallel data to default value of *11'b11111111111,* otherwise when the data is clocked, add *1'b1* to the most significant bit and concatenate the least significant bit off the parallel data.

```
always@(posedge sr_clk or negedge rst) begin
        if (~rst) begin
                bitShift <= 11'b11111111111;
        end else begin
                bitShift <= {ser_data_in, bitShift[10:1]};
        end
end
```

Serial_in

Concatenate the the least significant bit, most significant bit and parity bit off the received parallel data to pass 8 bit data to the CPU. *serial_in* also works as the top module of receive side.

```
assign data_out = parallel_out[8:1];
parallel_buffer SR(parallel_out, ser_data_in, sr_clk, rst);
serial_in_ctrl SR_ctrl(sr_clk, char_received, ser_data_in, clk, rst);
```

Top Level Integration

The top level hardware routes all the PIOs from the scanner/buffer subsystem as well as the Serial-Parallel-Serial Network Interface to the NIOS II CPU, effectively handling all communication from the CPU to the Serial-Parallel-Serial to the scanner/buffer subsystem and back. The top level interface is also responsible for calling the BCD module (Binary-coded decimal) to display both scanned data and transferred data of each scanner on the board's hex displays.

TEST PLAN

Scanner subsystem with buffer modification

With the buffer modification, we need to retest the functionality of the scanner module. If the single scanner can behave correctly, we can say, with confidence, that the scanner subsystem will behave correctly, as it is built based on the same top level logic with the legacy scanner. The following are the functionalities to be tested of the scanner:

1. Active Low reset pull the scanner to Low Power stage.
2. Go_to_Standby signal pull the scanner to Standby stage.
3. Start_Scan signal pull the scanner to Active stage.
4. Data will be correctly registered with command of write enable line
5. Idle when buffer is full.
6. Transfer signal pull the scanner to Transfer stage and start transfer data.
7. Reading address will increment according to read_inc input command
8. All data should be stored in the correct address, and output when called.
9. Flush signal will pull scanner to Flush stage and flush all data
10. Ready to Transfer signal high when buffer is 80% full
11. Signal the other scanner to go to Standby and start scan when 80% and 90% full.

Network system

Network system is comprised of two parts, transmit and receive. We test two parts separately. We test if bits propagate correctly when the data is clocked on both sides. Data need to have start bit, end bit and parity bit so that we test if these are added or stripped off correctly.

Software and Top Level Integration

Phase 1:

1. Test if commands sent to the CPU through the Eclipse console have an effect on the scanner/station subsystems which are displayed by hexes and LEDs on the board.
2. Test if the CPU can successfully send data to the scanner buffer for it to hold.
3. Test if the CPU can successfully receive data that is held by the scanner buffer.

Phase 2:

Our system as the scanner

1. Test if the CPU receives a *start_scan_receive* signal from the GPIO (connected to another DE1-SoC board).
2. Test that the CPU can forward this signal to the scanner subsystem and that data is written to the scanner buffer system once it receives this signal.
3. Test that the CPU can issue a *ready_transfer_send* signal through the GPIO once the scanner buffer is 80% full.
4. Test that the program waits for a *transfer_receive* signal through the GPIO, otherwise moves on to the second scanner.
5. Test that once the *transfer_receive* signal is received, the CPU begins reading data from the scanner's data buffer and transmits it through the *net_data_out* bus GPIO.
6. Test that the second scanner also is able to start scanning and sends and receives the appropriate signals.

Our system as the station

1. Test that the CPU can trigger a *start_scan_send* signal through the GPIO.
2. Test that the CPU waits until it receives a *ready_transfer_receive* signal from the GPIO.
3. Test that once the CPU receives a *ready_transfer_receive*, it prompts the user if they would like to send a *transfer_send* signal to the GPIO.
4. Test that a transfer signal can successfully be sent through the GPIO.
5. Test that data can be received through the *net_data_in* bus GPIO and is displayed in the Eclipse console.

TEST SPECIFICATIONS

Scanner subsystem with buffer modification

To test the functionality of the scanner system, we created a testbench and used iverilog and GTKwave to generate a waveform of the modelled system. The following is the sequence used to test the scanner module logic.

1. Initialize all input signal to inactive.
2. Reset the scanner.
3. Pull the scanner to stand by.
4. Pull the scanner to active stage.
5. Write 5 to 14 consecutively to the scanner buffer, addressed from 0 to 9.
6. Signal the scanner to transfer.
7. Signal the scanner to increment read address 10 times to read the data in all addresses.
8. Write to the scanner to full again
9. Signal the scanner to flush all data.

This sequence will thoroughly test all functionality of the scanner module, have the scanner to enter all possible stages and should activate any flags from the scanner at least once.

Network system

Module *serial_buffer*

- Test if reg *bitshift* set to default value when load signal is set to high
- Test if *bitshift* concatenate the least significant bit and add *1'b1* to the most significant bit when sr_clk is set to high.
- Test if value of *bit_out* (1 bit of data) is the same with second least bit in the *bit shift* since the least significant bit is about to concatenated in the next sr_clk.

Module *serial_out*

- Since data need to include 1 bit of parity data, start bit is set to 0 and end bit is set to 1, test if data will be formatted correctly.

Module *parallel_buffer*

- Test reg *bitShift* concatenate the lsb and add 1 bit input (ser_data_in) to the msb

Module *serial_in*

- Test if msb, lsb and parity are striped off when CPU receives the data.

Software and Top Level Integration

Phase 1

1. Once the user sends the start scan signal via the Eclipse console, the console should reply with the string “scanning...” and the hex display counter representing scanner 1 should begin incrementing from 0 to 9. The board will also display LEDs representing that the scanner is currently in the active state.
2. Once the buffer reaches 80% capacity, the Eclipse console will display the string “scanner 1 ready to transfer” and will prompt the user if they want to “start scanner 1 transfer? (y/n)”.
3. If the user commands the scanner to transfer with a ‘y’ character input to the Eclipse console, the scanner data should show up in the Eclipse console, and the hex display on the board should show the data that is being transferred. Once the transfer is complete, the user will be prompted if they want to start the second scanner.
4. If the user commands the scanner to not transfer with a ‘n’ character input to the Eclipse console, the first scanner should dump its contents and the program will prompt the user if they would like to start the second scanner.
5. The second scanner’s expected behaviour should follow similarly to that of the first scanner.

Phase 2

Our system as the scanner

1. If the program has initialized correctly on the CPU, the Eclipse console will display the string “Initialized”. Once the scanners have initialized, the console will display “scanners initialized”.
2. The console will then wait for the GPIO signal *start_scan_receive*. If this signal is true, scanner 1 will begin scanning, otherwise the system will wait.
3. Once *start_scan_receive* is true, the CPU will forward this command to the scanner subsystem and scanner buffer will begin collecting data and displaying it in the Eclipse console and the hex display.
4. Once the buffer reaches 80% capacity, the CPU will send a *ready_transfer_send* GPIO signal, as well as display the string “scanner 1 ready to transfer”.

5. Once the buffer is 100% full, the program will wait to receive a *transfer_receive* GPIO signal. If this is not received, the first scanner's data will be dumped and second scanner will be activated. It will follow a similar sequence of events.
6. If the *transfer_receive* signal is activated, the console will display "transferring..." and then show the data that has been received through the *net_data_in* bus, followed by the string "transfer complete".
7. The second scanner will operate in a similar fashion.

Our system as the station

1. If the program has initialized correctly on the CPU, the Eclipse console will display the string "Initialized".
2. The console will display a prompt "start scanning? (y/n)" and await user input.
3. If the user starts the scan, the *start_scan_send* GPIO signal will be true and the console will display "waiting..." as it waits for the scanner to collect data.
4. If the user does not start the scan, the system will continue asking the user if they would like to start the scan.
5. When the CPU receives a *ready_transfer_receive* signal from the GPIO, it will alert the user through the console with "ready to transfer" and then follow up with the prompt "transfer? (y/n)" and await user input.
6. If the user send the transfer signal, the *transfer_send* signal will be active and our system should receive data via the *net_data_in* bus and display them in the Eclipse console.
7. If the user does not send the transfer signal, no action will be taken and the program will loop back to the beginning and prompt the user if they would like to begin scanning.

TEST CASES

Scanner subsystem with buffer modification

- Setup: Using a testbench, assign registers to inputs and wires to outputs respectively. Simulate using the testing procedure indicated in the testing specification, synthesize the testbench file using iVerilog and produce waveform on GTKWave.
- Inputs:
    - rst: Active low reset line. Initiate the scanner and pull it into Low Power.
    - go_to_standby: Pull scanner to standby stage when in Low Power
    - start_scan: activate the scanner to start taking in data from data_in.
    - transfer: start the transfer process of the scanner, read address can only be incremented in the transfer stage.
    - flush_signal: prompt the scanner to flush all data and reset the write address to zero, scanner go to Low Power.

    - read_inc: read address increment signal. Read address should increment on the posedge of this signal in Transfer stage. Should be strobed for 10 times during transfer test sequence.
    - wr_en: write enable line. The buffer will write the data on the data input bus to the current address indicated by the write address counter on posedge of this signal. Counter will increment on negedge of the signal. Should be strobed for 10 times during scanning test sequence.
    - data_in: 8-bit data input bus. Scanner buffer should store the data appeared on this bus. Should increment from 5 to 14 during scanning test sequence.
- Expected Outputs:
    - state: 3-bit register that store and indicate the current stage of the scanner. Should change to the appropriate value at different stage of the scanner.
        - low power = 3'b000
        - active = 3'b001
        - standby = 3'b010
        - idle = 3'b011
        - flush = 3'b100
        - transfer stage = 3'b101
    - data_out: 8-bit data output bus. Should display 5 to 14 in the transfer test sequence.
    - address: the write address of the scanner buffer. Indicate the current memory address to write into, as well as the fullness of the buffer. Should be set to zero during Low Power stage and increment on negedge of the wr_en signal.
    - ready_second_buffer: signal that pull the second buffer to stand by. Should be high when the buffer is 80% full.
    - start_second_buffer: signal that pull the second buffer to active stage. Should be high when the buffer is 90% full.
    - ready_to_transfer: signal that notify the station that the scanner is ready to transfer the data in buffer. Should be high when buffer is 80% full or above.
- Pass/Fail Criteria
    - All output should display the expected result during the correct time period. The test should be repeatable.
    - If any signal falls out of the expectation, the system has design flaws and should be corrected.

Network system

Module *serial_buffer*

- Setup: Using a testbench, assign registers to inputs and wires to outputs respectively. Simulate using the testing procedure indicated in the testing specification, synthesize the testbench file using iVerilog and produce waveform on GTKWave.
- Input:
    - rst: 1 bit signal to rest the each bit of data in the parallel_in reg to be 1 when 1 bit load signal is received.
    - sr_clk: 1 bit active high clock control signal.
    - parallel_in: 8 bit parallel data
    - load: 1 bit signal that fill the reg which contains 8 bit parallel data that is ready to sent out serially.
- Outputs:
    - bit_out: 1 bit data is sent out when the parallel data is clocked.
- Expected results:
    - bits in reg *bitshift* set to default value when load signal is set to high
    - *bitshift* concatenate the least significant bit and add *1'b1* to the most significant bit when sr_clk is set to 1
    - *bit_out* is the same with second least bit in the *bitshift*
- Pass/Fail Criteria:
    - The above expected results based on the input values given are considered a pass in any case.
    - Otherwise the system has design flaws.

Module *serial_out*

- Setup: Using a testbench, assign registers to inputs and wires to outputs respectively. Simulate using the testing procedure indicated in the testing specification, synthesize the testbench file using iVerilog and simulate on GTKWave.
- Inputs:
    - rst: reset the tBase counter
    - clk: standard clock signal coming from the top level module.
    - trans_en: 1 bit signal to enable the BIC and BSC counter
    - load: 1 bit signal that fill the reg which contains 8 bit parallel data that is ready to sent out serially.
    - data_in: 8 bit parallel data that is ready to sent out serially.
- Outputs:
    - bit_out: 1 bit data is sent out when the parallel data is clocked.
    - char_sent: 1 bit signal to indicates that 1 bit of parallel data is sent out.

- Expected results:
    - The correct format add msb as *1'b1*, followed by a parity bit. It also adds lsb as *1'b0*
- Pass/Fail Criteria:
    - The above expected results based on the input values given are considered a pass in any case.
    - Otherwise the system has design flaws.

Module *parallel_buffer*

- Setup: Using a testbench, assign registers to inputs and wires to outputs respectively. Simulate using the testing procedure indicated in the testing specification, synthesize the testbench file using iVerilog and simulate on GTKWave.
- Inputs:
    - rst: 1 bit signal to rest the each bit of data in reg to be 1
    - sr_clk: 1 bit active high clock control signal.
    - ser_data_in: 1 bit data that is received
- Outputs:
    - parallel_out: 11 bit data
- Expected results:
    - reg *bitShift* concatenate the lsb and add 1 bit input (ser_data_in) to the msb
- Pass/Fail Criteria:
    - The above expected results based on the input values given are considered a pass in any case.
    - Otherwise the system has design flaws.

Module *serial_in*

- Setup: Using a testbench, assign registers to inputs and wires to outputs respectively. Simulate using the testing procedure indicated in the testing specification, synthesize the testbench file using iVerilog and simulate on GTKWave.
- Input:
    - rst: reset the tBase counter
    - clk: standard clock signal coming from the top level module.
    - ser_data_in: 1 bit signal that indicates if there is data available to be received.

- Outputs
    - data_out: 8 bit data is received by CPU when the serial data is clocked.
    - char_received: 1 bit signal to indicates that data is received.
- Expected results:
    - msb, lsb and parity are striped off when CPU receives the data.

- Pass/Fail Criteria:
    - The above expected results based on the input values given are considered a pass in any case.
    - Otherwise the system has design flaws.

Software and Top Level Integration

Phase 1

- Setup: Run the scanner_integration.c program on the NIOS II CPU. Ensure that the scanner/station subsystem has been deployed on the FPGA.
- Input:
    - Data_in_0: If scanner 1 has received a transfer signal, this will contain scanner 1's data buffer. It will be displayed in the Eclipse console and the hex display of the board.
    - Data_in_1: If scanner 2 has received a transfer signal, this will contain scanner 2's data buffer. It will be displayed in the Eclipse console and the hex display of the board.
    - Ready_to_transfer_0: Once scanner 1 has reached 80% data capacity, it will send this signal to the station to indicate that it is ready to transfer data. This signal will be displayed in the Eclipse console.
    - Ready_to_transfer_1: Once scanner 2 has reached 80% data capacity, it will send this signal to the station to indicate that it is ready to transfer data. This signal will be displayed in the Eclipse console.
- Outputs
    - Data_out_0: Data to be written to scanner 1's buffer during its active state. Displayed in the Eclipse console and the hex display of the board.
    - Data_out_1: Data to be written to scanner 2's buffer during its active state. Displayed in the Eclipse console and the hex display of the board.
    - Start_scanning: Signal that starts scanner 1. Received through Eclipse console and sent to scanner subsystem.
    - Start_transfer: Signal that permits either scanner to begin transferring data to the station. Received through Eclipse console and sent to scanner subsystem.
    - Wr_en_1: Signal required to be pulsed to allow scanner 1 to write data to its buffer. This must be pulsed each time we write to the scanner.
    - Wr_en_2: Signal required to be pulsed to allow scanner 2 to write data to its buffer. This must be pulsed each time we write to the scanner.
    - Read_inc_1: Signal required to be pulsed to allow scanner 1 to transfer data from its buffer to the station. This must be pulsed each time we read from the scanner.

    - Read_inc_2: Signal required to be pulsed to allow scanner 2 to transfer data from its buffer to the station. This must be pulsed each time we read from the scanner.
    - Scanner_rst: Resets scanners and dump collected data. System will initialize back and prompt user to start scan via the Eclipse console.
- Pass/Fail Criteria:
    - The above expected results based on the input values given are considered a pass in any case.
    - Otherwise the system has design flaws.

Phase 2

- Setup: Run the receive.c or the transmit.c program on the NIOS II CPU. Ensure that the scanner/station subsystem has been deployed on the FPGA and that you have connected the GPIO pins to another system.
- Note: The inputs/output test cases for this phase are identical to the ones in phase 1, with the following additions:
- Inputs
    - Char_sent: Signal indicating that the CPU has sent data to the ART. Should be high when data is being sent out.
    - Char_received: Signal indicating that the CPU has received data from the ART. Should be high when data is being received.
    - Net_data_in: Contains data received during transmission from another DE1-SoC board. This will only be active once our system has sent a *transfer_send* signal to the scanner system via GPIO.
    - Start_scan_receive: Signal allowing our scanner system to begin scanning. Received from another DE1-SoC board acting as the station.
    - Ready_transfer_receive: Signal indicating to our station that the network scanner is ready to transfer. Received from another DE1-SoC board acting as the scanner. Once this is received as a high input, our CPU will display a string in the console stating that the scanner is ready to transfer. Following this, the console should prompt the user if they would like to send a *transfer_send* signal.
    - Transfer_receive: Signal indicating to our scanner to transfer its data to the station. Received from another DE1-SoC board acting as the station. Once this is received, our scanner will begin transmitting data via the GPIO pin *net_data_out.*
- Outputs
    - Transmit_en: Signal indicating to the ART that the microprocessor is transmitting data. Should be high while our scanner is transferring data to the station.
    - Load: Signal indicating to the ART that it should load parallel data in. Should be active while our scanner system transmits data out.

    - Net_data_out: GPIO data transferred during transmission to another DE1-SoC board. Should only be used if our scanner system has received a *transfer_receive* signal from the station.
    - Start_scan_send: GPIO signal commanding the scanner system to begin scanning. Sent to another DE1-SoC board acting as the scanner. Received from user input in the Eclipse console.
    - Ready_transfer_send: GPIO signal indicating to the station that the scanner is ready to transfer. Sent to another DE1-SoC board acting as the station. This signal must only be sent once the scanner data buffer has reached 80%.
    - Transfer_send: GPIO signal indicating to the scanner to transfer its data to our station. Sent to another DE1-SoC board acting as the scanner. This signal is received from user input in the Eclipse console.
- Pass/Fail Criteria:
    - The above expected results based on the input values given are considered a pass in any case.
    - Otherwise the system has design flaws.

PRESENTATION OF RESULTS

Scanner subsystem with buffer modification

The test on the scanner has proven to be successful. All functionality of the scanner has been tested and it has provided the expected results. The following are GTKwave captures that shows some of the resulting waveform.

As can be observed in the waveform below, the scanner was first initialized and got to stand by stage on command. After a start scan signal is issued, it went to the active stage. As write enable line strobe and data input increment, the scanner is putting the data into consecutive address. And during later transfer stage, as read increment signal strobe. Data that are previously stored reappear on the data output bus in the correct order. After all data is displayed/transferred, the scanner goes back to Low Power.

Figure 1. Scanning and Transfer sequence of the Scanner module Test

In the second half of the test, after writing the buffer full, a flush signal is issued. As can be observed below, the scanner enters the flush stage briefly, reinitialize the data in the buffers and put write address counter back to zero.

Figure 2. Flush test for the scanner module

Network system

Module *serial_buffer*

The testbench provide the signal requirement specified in the Test Cases. And the module has provided the desired signal outputs. The following GTKwave capture shows some of the resulting waveform:

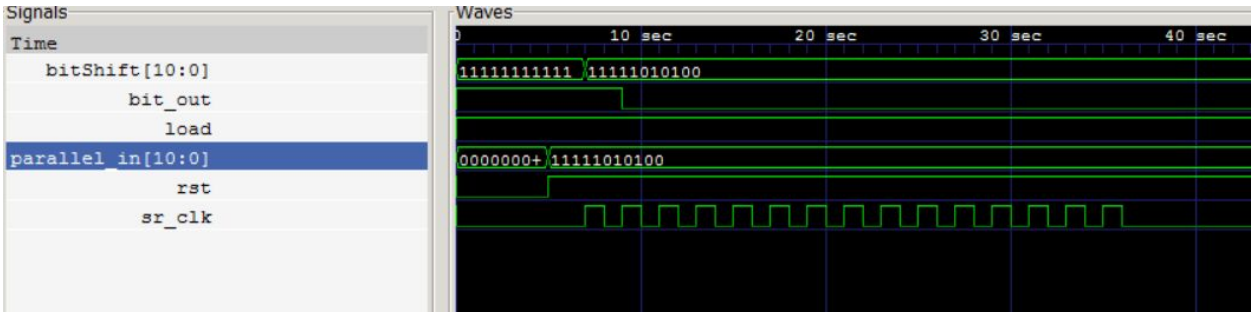

Figure 3. serial_buffer testbench

As can be observed, when signal load is set 1, *bitShift* is set to default value (*11'b11111010100*)

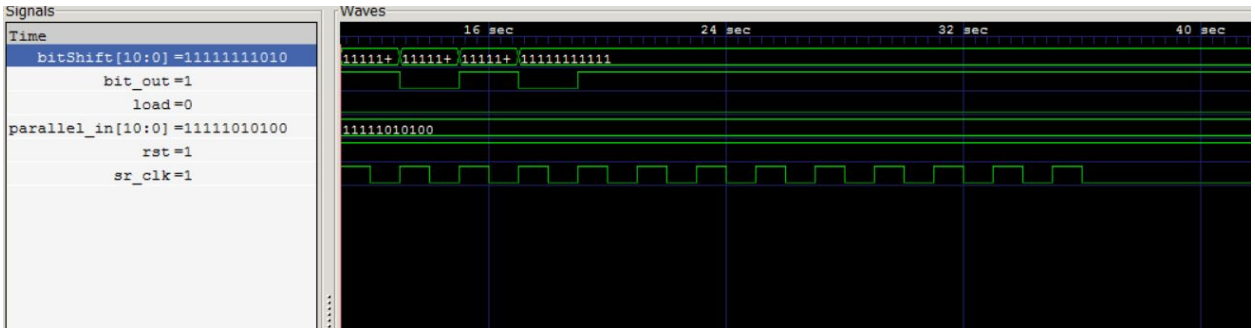

Figure 4. serial_buffer testbench

As can be observed, *bitShift* concatenate the least significant bit and add *1'b1* to the most significant bit to the *parallel_in*. In the sr_clk, the lsb of *bitShift* will be concatenated, so the lsb will become *1'b1*, which is the same with the *bit_out*.

Module *serial_output*

The testbench provide the signal requirement specified in the Test Cases. And the module has provided the desired signal outputs. The following GTKwave capture shows some of the resulting waveform:

Figure 5. serial_out testbench

As can be observed, data_in_full adds add msb as *1'b1*, followed by a parity bit *1'b0* to the data_in. data_in_full adds lsb as *1'b0* to the data_in

Module *parallel_buffer*

The testbench provide the signal requirement specified in the Test Cases. And the module has provided the desired signal outputs. The following GTKwave capture shows some of the resulting waveform:

Figure 6. parallel_buffer testbench

As can be observed, *ser_data_in* is added at the msb in the *bitShift,* and lsb is concatenated.

Module *serial_in*

The testbench provide the signal requirement specified in the Test Cases. And the module has provided the desired signal outputs. The following GTKwave capture shows some of the resulting waveform:

Figure 7. serial_in testbench

As can be observed, msb, lsb and parity are striped off from *parallel_out.* data_out shows the correct format.

Software and Top Level Integration

Phase 1
The following figure is an example of Eclipse console I/O.

```
scanners initialized
start scanning?(y/n)

y
scanning...

scanner 1 ready to transfer
start scanner 1 transfer?(y/n)

n

scanner 2 ready to transfer
start scanner 2 transfer?(y/n)

y
transferring...
0
1
2
3
4
5
6
7
8
9
transfer complete
start scanning?(y/n)
```

Figure 8 NIOS II Eclipse console output while running Phase 1

This console shows user interaction with the CPU system. If the user chooses not to permit scanner 1 to transfer its data, the system moves on to scanner 2. Once the user prompts to transfer scanner 2's data, the received data is displayed in the console.

Phase 2 followed a similar console output, depending on if our system acted as the scanner or the station. If our system acted as a scanner, the console output would show "Initialized" on system reset and "scanners initialized" once the scanners are ready. Once a *start_scan_receive* signal is received, the console shows the data being written to the scanner (similar to how it's shown in Figure xxxx). Once the scanner has reached 80% capacity, the console outputs "scanner 1 ready to transfer" successfully and then outputs "waiting to transfer" until it receives a transfer signal from the station. Once it receives a transfer signal, the console outputs "transferring..." and then

displays the data being transferred to the station (similar to Figure xxxx), followed by the output "transfer complete". If it does not receive a transfer command, the system moves onto scanner 2 and performs a similar console I/O.

If our system acted as the station, the console output shows "Initialized" on a system reset, followed by a prompt to the user "Start scanning? (y/n)". If the user inputs 'y' for yes, the station will send out a *start_scanning_send* signal and output "waiting..." to the console, signifying that it is waiting for a *ready_transfer_receive* signal from the scanner. Once it receives this, the console will output "ready to transfer" and then prompt the user "transfer? (y/n)". If the user inputs 'y' for yes, the system sends out a transfer signal to the scanner and begins displaying the received scanner data on the console. If the user does not permit a transfer, the program simple loops back to the beginning and asks if the user wants to start a new scan.

Our system behaves in the expected way without any flaws, so we have a successful system.

ANALYSIS OF ERROR

No apparent error remained in the final design of the system. The system meet all specifications and behave with stability.

SUMMARY

In this lab, we start by building small projects to get us to know how to use Qsys to design, implement and test microprocessor on Altera NIOS II core and apply our C language to develop applications on the microprocessor using Eclipse. We then apply our C knowledge to integrate the scanner system into the microprocessor by adding extra PIOs to it. We also build and test an asynchronous serial network into our project so that it allows scanner and CPU to transmit and receive to each other.

CONCLUSION

In this lab we were able to successfully use new tools, such as Qsys and Eclipse, to build a NIOS II microprocessor and run C programs on it that allowed us to communicate with our scanner/buffer subsystem hardware. Initially, we designed several basic programs such as the "lights" and "hello world" program to learn how to use the CPU to communicate with existing hardware. We then configured the CPU and added additional PIOs so that it would be able to communicate with our scanner subsystem hardware. Once this was successful and we were able to use the microprocessor to act as both the station and the scanner control, we designed ART hardware by constructing a Serial-Parallel-Serial interface and using GPIO pins on the DE1-SoC board. The purpose of this was so that we could communicate with another DE1-SoC board. This way, we were able to act as the scanner controller system, while using another DE1-SoC board as the station system, and vice-versa.

CHART OF CONTRIBUTIONS

Task	Team Member
Count Binary	Boren Li, Sailesh Suri, Wenbing Zhang
Lights and switches	Boren Li, Sailesh Suri, Wenbing Zhang
Hello world small	Sailesh Suri, Wenbing Zhang
Scanner system modification	Sailesh Suri Wenbing Zhang
NIOS II processor	Wenbing Zhang, Boren Li
Network interface	Boren Li, Sailesh Suri, Wenbing Zhang
C code for NIOS Eclipse	Sailesh Suri, Wenbing Zhang

Appendices

1. Top level RTL view of the designed system