EE 371 Lab 5

Extending a Simple Microprocessor – Building an Application….

University of Washington - Department of Electrical Engineering

James K. Peckol

Introduction:

In this fifth and final project, we will continue work with the Qsys tools and concepts as we extend a microprocessor based scanner system on our FPGA. Now that we have the core processor with basic I/O implemented and integrated and have put our knowledge of C to work with a couple of simple programs we will utilize those skills to complete the system.

The first phase of the project will entail developing and testing an asynchronous serial network. The second phase will utilize the network to transfer the collected data to a tracking station next to the canal.

Again, of note, the processor that we are building and working with is very similar to that which we have in our home or lab PCs.

Prerequisites:

Familiarity with the Quartus II, Qsys development environment, and the Signal Tap (nope still ain't *Spinal Tap*) logic analyzer will definitely help (note, this is not *Q* from Star Trek nor the Game of Thrones…nor the Greek god of the sky). A continued willingness to learn and to explore is good. No birra or Barbera del Monferrato until the project is completed and you can turn on a several LEDs using switches on the processor, print the requisite *hello world* as you run several C programs on our new microprocessor, talk with the outside world, create a new form of indie rock, and have fun. Hey, these are all still cool.

Cautions and Warnings:

Once again, try to keep your DE1-SoC board lower than your PC thus making it easier for the electrons to get to your board and making the data transfer much faster. However, this will not affect the speed at which your program runs because by that point, the program is already at its destination. Is it true that if your program is a few bits short or your logic has a few bad bits that you can go to engineering stores and can buy some new ones using bit coins? Will WD40 help you loosen up any stuck bits?

Background:

Have some knowledge of the C language and ability to develop simple C programs. Have a working understanding of the Verilog HDL modeling and synthesis, intra and inter system timing and timing constraints, and basic methods of system I/O. Have a basic understanding of network fundamentals.

When debugging digital circuits, it is helpful to understand what each part of the circuit is doing and exactly when certain events are occurring. The amount of information can be huge, and often we would like to filter out as much of the unnecessary data as possible. Also, sometimes we are interested in timing measurements, other times we are more concerned

with comparing the different states of various parts of the circuit. As we've learned, a logic analyzer can be a great help with all of this, and much more.

Microcomputer communications is a rapidly growing field with an ever-increasing number of applications, ranging from intra and inter exchanges on SOCs (system on a chip) to local PC networks and on to large-scale communication systems between cities and countries. Familiar schemes include SPI, I^2C, Bluetooth (developed in Professor Sahr's lab at the UW) CAN, MAP, USB….

Central to any communications between electronic devices is a *protocol* for transmitting and receiving information. A key point to understand is that a protocol is *not* simply a collection of binary 0's and 1's expressed by different voltage levels or the presence or absence of light or radio waves. A protocol is the *meaning* placed on those signals and that different protocols can apply at different levels of a network hierarchy.

Within a microprocessor, data is usually transferred in parallel form. While parallel communication is far more efficient than serial, it is not always as practical, and thus most communications between computers and other electronic devices not in the immediate vicinity of each other usually occurs in serial form. Of course having two different formats for communicating data can generate many occasions when data have to be converted from one form to the other.

Another important aspect of communication is ensuring that the data given to the user, following reception, contains no errors arising from such a transmission. Note, we do not guarantee no transmission errors, these happen; rather, at the end of the day, we guarantee the data to be correct. There are a variety of schemes by which this is accomplished: all begin with recognizing that a transmission error has occurred. We'll examine one, simple parity checking.

Serial Communication: Asynchronous vs. Synchronous

Asynchronous communication means and is used where there is irregular intervals between the sending of data. In this project, we'll develop one such scheme.

Suppose that a serial communication line is set up to transmit ASCII characters as typed by a person at a keyboard. The spacing between the transmission of each character will vary widely, and there may be long periods when no characters are typed (coffee breaks, tea breaks, bio breaks, checking Facebook, texting, naps, etc.). In such situations, the receiving device needs to know when a character is being sent so that it can receive the character then sort out which part is data, which part is the error-checking field, and so on.

In our design, this is accomplished by a procedure known as *framing* (no, this is not a telecoms selfie). To signify the start of a character, an initial state transition from the specified quiescent state occurs on the transmission line before the first bit of the data character. After all of the bits of the character are received, the state of the transmission line returns to the original quiescent state. These two specific states that *frame* the character are designated the *start bit* and the *stop bit*. The *start* bit enables the receiving device to temporarily synchronize with the transmitting device, while the *stop* bit allows the receiving device time to get ready for the next frame.

In contrast, when blocks (usually large ones) of regularly spaced data are transferred over a serial line, the transmitter and receiver can be synchronized, transferring characters at a much

higher rate. In this format, known as *synchronous transmission*, start and stop bits are no longer needed, since the receiving device knows exactly where each character begins and ends.

Framing

In asynchronous serial communications, data is grouped into frames, which have already been partially described above. As noted above, the start of a frame is signaled by the start bit, the end by one or more stop bits. Data and error checking (parity bit) codes are contained within each frame.

Inside of our microprocessor, we want to work with data expressed in parallel rather than serial. In order to perform serial to parallel conversion, our system must know what part of the frame is being looked at. For example, we don't want to be looking for a start bit in the middle of the data, nor do we want to be checking parity when the start bit is being received. One way to keep track of the serial input is to use a counting mechanism activated by the start bit of each frame. The output of the counter can be used as input to control logic that determines when to clock in a data bit, when to check parity, and so forth.

Timing

With an asynchronous exchange, the transmitting and receiving devices are operating on independent clocks that are generated locally at the source and destination. Although the clocks are ideally operating at the same rate, clearly, there is no guarantee that the edges are coincident. However, if managed properly, a difference of a percent or two in relative clock transitions should not result in transmission errors.

Clocks are usually designed to operate at 16x, 32x, or 64x the data rate - why do we do this? We will originally be using a data transfer rate of 9600 bits/sec (bps), and a 16x clock. That is, for the converter to be built in the first part of the project, the data link will be sampled for the start bit at 16x the data transmission rate. We'll see why shortly.

Once the start bit is detected, the *bit count* logic will begin counting the number of clocks. In the circuit given in Figure 1, two 4-bit counters are used in conjunction with data transfer in each direction. One is identified as the *bit sample count*, (*bsc*) and the other is the *bit identification count* (*bic*). The value of the *bsc* will indicate whether we are looking at the start or end of a bit, or somewhere in between, whereas the value of the *bic* will tell us what bit is present at the input.

Objectives:

The major objectives of this project include:

- Design and develop a hardware system on the Cyclone V FPGA on the DE1-SoC board that comprises a NIOS II processor and a serial-parallel-serial network that will support asynchronous message exchange with a similar system.
- Design and develop the control software and display driver for such a system.
- Utilize our system to support a basic I/O driver. The driver will be written in the C language.
- Incorporate the ability to transfer data from the scanner system to a canal side collection station.

Designing and Building the Hardware Subsystem:

For the first part of the project, we will continue to work with the Quartus II and the Qsys tool to design and implement a simple microprocessor system that utilizes the Altera NIOS II core that will incorporate an interface to support a serial-parallel-serial network.

Implementing the Network Interface

Figure 1 below gives a high-level block diagram for the serial-parallel-serial network interface to be developed in this project. The network supports asynchronous full duplex data transmission and reception.

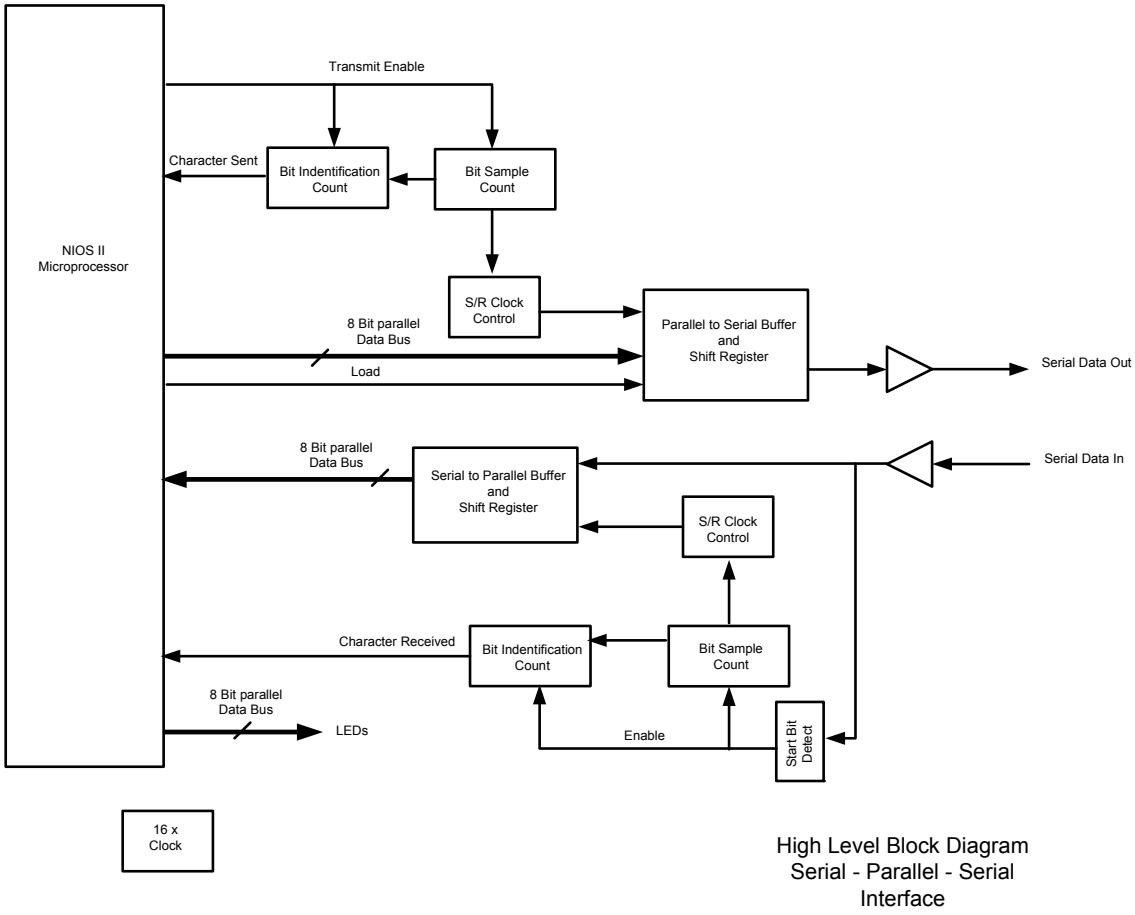

Figure 1

Major Functional Modules

Note: the module descriptions below and the circuit diagram are intended as high-level guides for your design. You are free to work with them or to make as many modifications as you feel are necessary, while still meeting the specifications, to achieve a well-organized, working design.

Counting Modules - BSC and BIC

As noted above, two counters are used to keep track of the incoming or outgoing serial data and are identified as a *bit identifier count* (BIC) and a *bit sampling count* (BSC). The BIC should be at the initial state of 0000 for the *start bit*. For the BSC, state 0000 indicates the start of a bit, 0111 indicates the middle, and 1111 indicates the end of the bit.

Any action that should be executed at certain times in the transmission frame will be directly controlled by the bit count.

Start Bit Detect – Transmit Enable

This logic block is used on the receive side to identify the start of an input data frame and thence signal the control logic to determine when to start and stop counting input data bits. The Transmit Enable serves the same function on the transmit side.

Shift Registers and Clock Control

On the receive side, the serial to parallel shift register will propagate the value present on its serial input to the left on each transition of its clock. In theory, the bit will be sampled at the halfway point.

On the transmit side, the parallel to serial shift register will take the entire frame that has been stored and shift it out serially, to the right, on each transition of its clock.

The clock control determines when the data is to be clocked.

Both registers are supported by buffers. Why are these useful or necessary?

Monitoring LED's

This allows us to easily see exactly what data is present on the parallel bus. It should simply be a series of LED's displaying the status of the bus.

Building a Microprocessor

For this first part of the project, we will focus on the core microprocessor system. Repeat all of the steps you used in the 3rd project to create the basic system. The *lights and switches* project is a good template to work from.

Things to think about…

a. Which NIOS II core would be the best choice?
b. How much on chip memory should you use?
c. Which system clock should you use?

Assign all your FPGA I/O pins, then build and download you new system onto the DE1-SOC board. You're halfway there.

Developing an Application

Once you have successfully built the hardware portion of the NIOS II processor based system and downloaded it to the DE1-SoC board, it's time to move on to the software part.

Let's first do a quick smoke test (no, not that kind !!!! these boards are expensive) to make certain that we truly do have a properly functioning system before moving on. Following exactly the same steps that you did for the 4th project, create a new C/C++ application. Stay with the *Hello World Small* template.

Use Signal Tap and the Eclipse console to verify the operation of each of the individual subsystems in your design

Include the header file altera_avalon_pio_regs.h. Then you'll be able to use the functions

IOWR_ALTERA_AVALON_PIO_DATA(*targetAddress, aValue*);

aValue = IOWR_ALTERA_AVALON_PIO_DATA(*sourceAddress*);

to write to or read from a memory address.

Once you have confirmed the individual subsystems, configure your system such that the transmit and receive outputs are connected together. Write a simple program to transmit a character out and then read that same character back in.

Developing the Final Application

When you have the first application successfully running – you've completed your of smoke test, move on to building the final application.

High Level Requirements

Overview

As we work towards our final satellite borne climate data collection system, we will use the NIOS II microprocessor to control the subsystem and to manage the communication with and transmission of data to a canal side tracking station. We will then connect to a second DE1-SoC system. In such a configuration, one system will model the satellite and the second the tracking station then reverse the roles.

Requirements

- The microprocessor shall generate the *StartScanning* command to begin the data collection process.
- When the *ReadyToTransfer* is received from the active scanner subsystem, the microprocessor shall send a request to the tracking station on shore.
- When the tracking station returns permission, the microprocessor shall send the *Transfer* command to the scanner subsystem.
- When the active scanner subsystem enters the transfer state, the microprocessor shall retrieve the contents of the scanner data buffer, format the data, and transmit one byte at a time to the tracking station.
- To format the data, the microprocessor must convert each data byte retrieved from the data buffer into an ASCII character. This can be done as a simple C language statement:

  ```
  dataByte = dataByte + '0'
  ```

Testing

Phase 1

- Using the testing strategies developed and utilized in the previous project and the first phase of this one, configure the system so that the microprocessor acts both as the scanner subsystem controller and the canal side tracking station.

  Specifically, configure your system such that the transmit and receive outputs of the comms portion of the system are connected together. Use the Eclipse

console to act as the tracking station to display messages received from the satellite and to respond by sending the appropriate commands to the satellite. Confirm the operation of the system.

Phase 2

- Connect your system to another DE1-SoC board. Let the second board serve as the canal side tracking station. Confirm the operation of the system.
- Let your system serve as the canal side tracking station and the second board as the canal scanner. Confirm the operation of the system.

Deliverables:

1. A full lab report as described on the class web page under *Workload and Grading*.
2. The annotated Verilog and C source code for the final applications both on the DE1-SoC board and on the PC.