Remote IO for an FPGA

A Design Project Report
Presented to the School of Electrical and Computer Engineering of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering, Electrical and Computer Engineering

Submitted by
Anthony Viego
MEng Field Advisor: Hunter Adams and Bruce Land
Degree Date: May, 2021

Abstract

Master of Engineering Program
School of Electrical and Computer Engineering
Cornell University
Design Report

Project Title: Remote IO for an FPGA
Author: Anthony Viego
Abstract: With the COVID-19 pandemic, education has changed greatly. A majority of students now learn remotely, unable to gather in a lab and physically interact with devices. This has resulted in limitations on the kinds of projects, labs, and learning opportunities that students have available to them. In order to grant more freedom to students, specifically those in ECE 5760, this project has been developed to allow them to make use of peripherals and devices remotely through a GUI. Combined with custom logic implemented on the FPGA, students are able to fully utilize peripherals in a manner similar to before. The logic and GUI have been designed with ease of use in mind, ensuring that students will only need a few minutes to integrate it into their future projects. While the pandemic is coming to an end, we hope this project will be used to augment in-person labs and projects in the coming semesters.

Executive Summary

With the COVID-19 pandemic, education has changed greatly. A majority of students now learn remotely, unable to gather in a lab and physically interact with devices. This has resulted in limitations on the kinds of projects, labs, and learning opportunities that students have available to them. In order to grant more freedom to students, specifically those in ECE 5760, this project has been developed to allow them to make use of peripherals and devices they would normally need to be in person to interact with. It has been designed with ease of use in mind, ensuring that students will only need a few minutes to integrate it into their future projects. While the pandemic is coming to an end, we hope this project will be used to augment in-person labs and projects in the coming semesters.

The design aspect of this project involved the development of a GUI for use on a host PC, a "middle man" device capable of handling communication, and hardware on an FPGA. The GUI on the host PC was developed using the PySimpleGUI library. This allows the GUI to be easily modified by students for use in labs and projects. The GUI is modeled after the layout of the DE1-SoC FPGA board to replicate the way students normally interact with the device. To allow for communication between this GUI and the FPGA, a custom serial protocol was developed. This protocol frames each message sent and defines a series of characters used for indicating unique events.

For the FPGA, custom logic was designed to allow for a flexible but easy to use system. All of the logic on the FPGA was developed as a Qsys module (a module used in the Intel Quartus software), allowing students to simply add or remove the entire system with a single click. The system features a custom serial transmitter and receiver, which handles sending and receiving serial messages using the custom protocol. For the sake of flexibility, the transmitter receives data from a round robin arbitration system, consisting of a number of FIFOs. This entire system, including the FIFOs and arbiter, was designed for this project without the use of any existing IP. This has allowed the system to be completely configurable in terms of the number of FIFOs, the depth of each FIFO, the width of each FIFO, how often data is transmitted, etc. This flexibility allows students to not only make use of the existing functionality provided by the GUI, but also allows them to easily expand the system to support new functionality for use in labs and projects.

At the present moment, the project itself has entered a "completed" state. The completed project currently features a GUI consisting of 6 hexadecimal displays, 10 LED indicators, 10 toggle switches, 4 push buttons, and a synchronization button. Communication between the GUI and FPGA appears to be working well, with almost no packet loss having occurred during testing. However, there is still room for improvement. The serial protocol currently implemented doesn't have any form of reliability or error checking, there is no method of flow control implemented between the devices, and the currently supported baud rate is only 38400 baud. Improvements in these areas would mean large improvements in both performance of the transmitter as well as reliability of the transmissions.

Contents

1 Introduction 3

2 Design Alternatives 3

3 System Design 5
- 3.1 FPGA - Hardware 5
  - 3.1.1 FPGA - Logic Design 5
- 3.2 FPGA - Qsys 7
  - 3.2.1 FPGA - Challenges 9
  - 3.2.2 GUI - Design 10
  - 3.2.3 GUI - Challenges 11
  - 3.2.4 Arduino 12

4 Results and Testing 12

5 Future Work 13

6 Conclusion 14

Appendices 14

Appendix A User Manual 14
- A.1 What to Read 15
- A.2 Adding Components to Qsys 15
- A.3 Configuring the Top-Level 20
  - A.3.1 Instantiating the Qsys system 20
  - A.3.2 Pin Assignments and Project Files 22
- A.4 Running and using the GUI 23
  - A.4.1 Functionality of the GUI 24
- A.5 Quartus Interface - Details 24
  - A.5.1 Serial Protocol 26
  - A.5.2 System Infrastructure 27
  - A.5.3 Data Transmission 27
  - A.5.4 Data Receiving 28
  - A.5.5 Modifying the System 28
- A.6 Modifying the GUI 29

Appendix B References and Documents 29
- B.1 Project Setup Guide 30
- B.2 DE1-SoC User Manual 30
- B.3 Quartus Handbook 30

Appendix C User-Manual Appendix **32**
C.1 Serial Protocol Types 32
C.2 Serial Protocol Functions 32
C.3 FPGA System Overview 33
C.4 References 33
C.4.1 DE1-SoC Project Setup Guide 33
C.4.2 DE1-SoC User Manual 33
C.4.3 Quartus 15.1 Handbook 33
C.4.4 Avalon Interface Specification 33
C.4.5 Platform Designer (Qsys) Property Specification 34

1 Introduction

With the COVID-19 pandemic, education has changed greatly. A majority of students now learn remotely, unable to gather in a lab and physically interact with devices. This has resulted in limitations on the kinds of projects, labs, and learning opportunities that students have available to them. In order to grant more freedom to students, specifically those in ECE 5760, this project has been developed to allow them to make use of peripherals and devices they would normally need to be in person to interact with.
The requirements and constraints of this project were determined by analyzing the needs of the students. In a typical semester, students would make use of various peripherals attached to the physical FPGA board, such as LED's, HEX displays, toggle switches, etc. As such, supporting each of these became the first requirement. Additionally, accessing these peripherals in verilog is usually extremely simple, and so the second requirement became to ensure that using the "remote peripherals" would be just as easy. These two requirements formed the original goals of the project, and helped to drive initial approaches to the problem. Over time, additional goals such as ease of setup and expand-ability were added as we hoped that this project would serve as as a backbone for students to use in their own final projects. This resulted in the final set of requirements for this project seen here:

- Create a GUI which support all peripherals on the DE1-SoC board used in ECE 5760
- Create a verilog interface which allows for simple access to these peripherals
- Ensure setup of the GUI and verilog components are simple and can be done relatively quickly
- Create a way for students to expand on the GUI and data being transmitted to support use in their own final projects

2 Design Alternatives

Throughout the course of this project many different designs were considered for the proposed system. The original idea presented was to make use of three components in the design, the first being a host PC running a python based GUI, the second being an Arduino to act as a "middle man", and the third being the DE1-SoC board. The idea here was that the host PC would send messages over serial to the Arduino. These messages would then be converted by the Arduino into individual I/O lines, forming a parallel interface of sorts. This would result in a one to one relation between peripheral and GPIO. Figure 1 shows the initial system overview. As can be seen, each peripheral supported by the GUI would receive its own set of I/O lines between the Arduino and FPGA. This results in a relatively simple system in which the FPGA simply needs to read the state of each GPIO pin in order to determine the state of a peripheral. However, the major downside of this approach is that the Arduino has a fairly limited amount of digital I/O pins (around 13 in

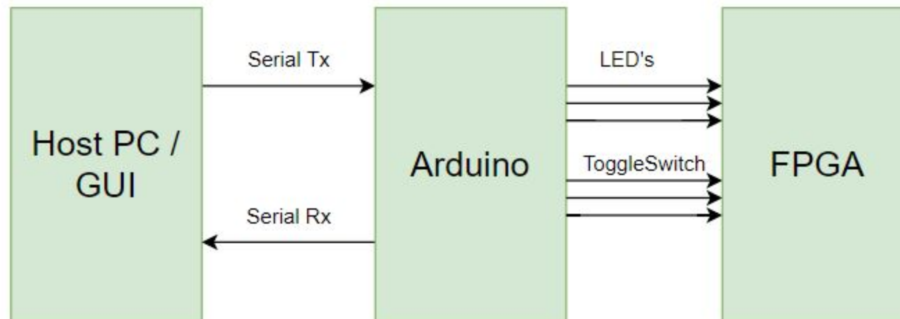


Figure 1: Initial System Overview

total). As such, this severely limits the peripherals that can be supported without some sort of additional hardware. For example, one could create a series of lookup tables which encode the possible state of each peripheral, and then use the GPIO pins to index into those tables. However, this approach would still shy away from the idea of simple expand ability as eventually every I/O pin on the Arduino would be consumed.

Attempting to improve upon the previous idea, it was decided to make use of a serial communication method instead in the hopes that this would allow for a more expandable system in the end. As shown in figure 2, the hardware on the DE1-SoC board is broken up into 2 halves, with one half being natively connected to the FPGA, and the other half being connected to the hard processor. Originally, this posed a problem as there was no simple way to connect the host PC directly to the FPGA, as there is no USB to UART converter connected to the FPGA. However, looking deeper into the Qsys module for the HPS (hard processor) revealed that there is a ”peripheral pin multiplexing” system in place. This allows certain pins on the HPS to be ”loaned” to the FPGA, including the pins connecting to the UART to USB converter. Along with a standard UART, this meant that serial communication could be achieved between the host PC and the FPGA using a single USB cable. Unfortunately, a downside of this approach is that the HPS typically uses this connection for outputting the serial console. As the Ethernet connection to the HPS was reported to be unstable at times, having this serial console was important for students to be able to reliably interface with the HPS. Due to this, it was decided to reintroduce the Arduino as a ”middle man” device.

The final design settled on is similar to what was presented in figure 1, but makes use of a serial connection between the Arduino and the FPGA, rather than a parallel one. The use of a serial connection is intended to allow generic data to be transmitted between the host PC and the FPGA, meaning that students could add their own data to be transmitted later on. For example, each serial message would contain some sort of update about the status of a peripheral. One message might contain 8-bits, with each bit describing whether a different LED is on or off. The FPGA would send this message to the host PC, and

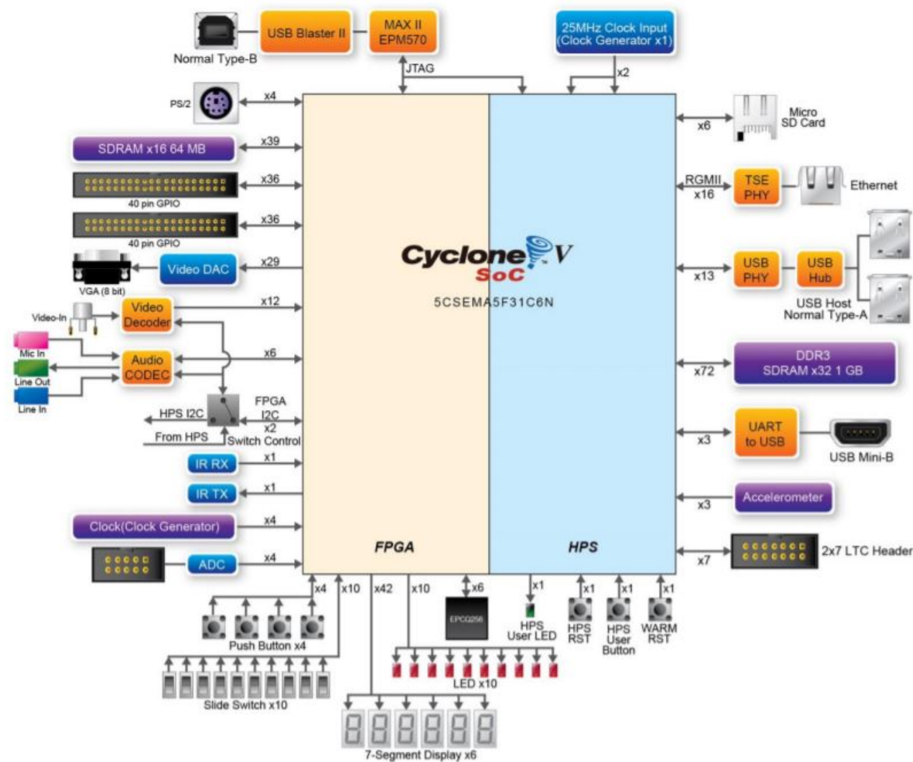


Figure 2: Block Diagram of DE1-SoC

the GUI would then be updated accordingly. As each message follows a defined protocol, students could easily format and send their own messages if they later wanted to expand upon the GUI.

3 System Design

3.1 FPGA - Hardware

As mentioned, the components of this project can be broken up into three pieces, the logic implemented on the FPGA, the Arduino which acts as a pass-through device, and the host PC running a GUI. Here we discuss the implementation of the logic on the FPGA.

3.1.1 FPGA - Logic Design

One of the goals of this project was to create an expandable logic design on the FPGA which would serve to not only interact with the peripherals added to the GUI for this project, but to also allow for interaction with peripherals added to the GUI later on by students in their own final projects. The resulting logic design can be found in appendix C.3. As

can be seen, the FPGA logic consists of several pieces, a serial transmitter, serial receiver, round robin arbiter, several FIFOs, and several data ports. The key point to mention in this design is that the number of FIFOs, size of the selecting mux, and the arbiter can all be scaled with a single parameter. By default, 7 FIFOs are needed to store the data being sent to the FPGA (6 FIFOs for the 6 seven segment displays and 1 for LED data) with an eighth being included for extra data. However, by changing a single parameter in Qsys (explained in a later section) students can adjust the system to include additional FIFOs. This allows the FPGA to support the transmission of as much data as is desired by the students, provided the FPGA has enough space to handle the associated hardware.

In terms of data transmission, data is directly stored in each FIFO through the top-level interface provided to each student, along with the valid signal. The empty signal of each FIFO is then used to signal to the arbiter that a FIFO has data available to be sent to the serial transmitter. From the FIFOs that are ready, the arbiter chooses one in a round-robin fashion, and the resulting grant is used as the selection bit for the mux. When the serial transmitter has received the data, it will send a signal to the FIFO allowing it to "pop" the data. Through this system, data from each FIFO can be transmitted to the GUI in a fair manner. Further details on the interface provided for data transmission can be found in appendix A.5.

The serial transmitter block itself is a finite state machine used to take in data from a FIFO and convert it into the required serial protocol format. The fsm can be seen in figure 3. In the IDLE state the transmitter is waiting to receive valid data from a FIFO. Once received, the transmitter signals that the FIFO can pop the data, and transitions into the WRITE state. In the WRITE state exists another FSM used for communicating with the UART. The two devices communicate through a val/rdy interface, in which the transmitter signals it has valid data, and the UART signals it is ready to receive. When both signals are high, communication between the two devices occurs. Once data transmission is completed, the transmitter returns to the IDLE state.

A simple custom serial protocol was created for the purposes of this project, which allows for data to be easily identified with an associated peripheral. For example, each serial message contains a "type" byte indicating the desired peripheral. Messages carrying updates for the LEDs are represented by the type byte "0x86". The serial protocol also defines starting and ending characters to make it simpler for serial transmitters and receivers to know when a message has been fully communicated. Further details on the serial protocol and the currently defined types and functions can be found in appendix A.5.1 and appendix C.1/C.2 respectively.

The serial receiver functions in a similar manner to the serial transmitter. Data is received from the RS232-UART block and stored in a local FIFO. The data is then parsed for identifying information such as the type byte, and payload of the message is then stored in the registers corresponding to that peripheral. For example, the GUI may send an update indicating that a toggle button has been pressed, in which case the corresponding bit in the key_out registers would be set to 1. Outside of these registers, there are no additional storage mechanisms currently implemented for storing received data. It is currently assumed that

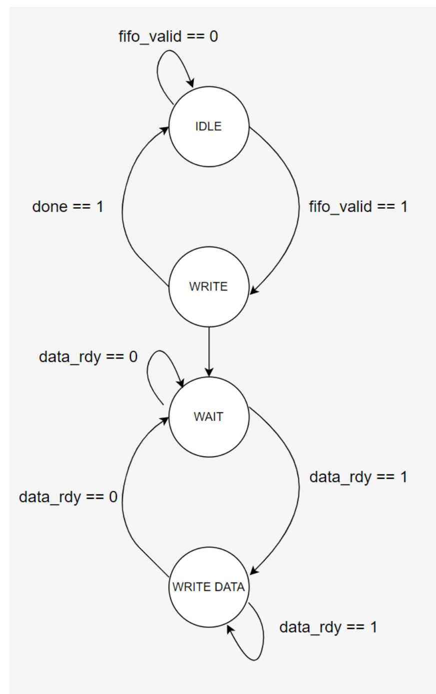


Figure 3: Serial Transmitter FSM

data being received by the FPGA would be polled when needed. If students later desired each piece of data received from a specific peripheral to be stored, the FIFO module created for data transmission could be added at the top level for this purpose.
One final aspect to note about the system implemented on the FPGA is that we make use of the Avalon streaming interface for communication between the serial transmitter/receiver, rather than the memory-mapped Avalon bus. The main reason for this decision is that the streaming interface is much simpler to implement, and doesn't require the additional space of a bus controller. Instead, simple val/ready signals can be used to communicate when data is ready to be received or sent from one device to another. It should be noted that the streaming interface only allows for direct communication between two devices, which happens to work perfectly in this scenario. Further details on the memory-mapped and streaming interfaces can be found in appendix B.3.1.

3.2 FPGA - Qsys

As described previously, one of the major goals for this project was to ensure that any logic implemented on the FPGA would be simple for students to add to their own projects. Additionally, the interfaces created should be simple to interact with, as well as expandable to allow for inclusion in final projects. Given these constraints, it was decided to make use of the Qsys tool (now known as Intel platform designer) within the Intel Quartus Software

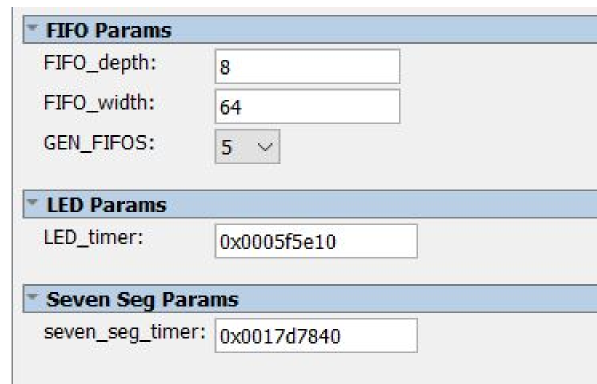


Figure 4: Qsys Module Parameters for io_avalon_interace

to encapsulate the logic. Qsys allows for systems to be built in a "drag and drop" sort of manner. Pre-defined modules can be easily added or removed from Qsys, and once a system has been arranged, Qsys will handle generating the necessary code to instantiate it in a top-level verilog file. As such, encapsulating any logic added for this project into a single Qsys module allows for students to easily add it into their own projects later on.

For this project, we created a single Qsys module named "io_avalon_interface", the parameters for which can be found in figure 4. As described in the logic design section, the number of FIFOs can be configured using the dropdown parameter "GEN_FIFOS". This dropdown allows students to select the number of additional FIFOs outside of the seven required for the peripherals currently implemented in the GUI. Due to limitations in how the generate statements for certain blocks work, the minimum number of FIFOs allowed at any given time is 8 (the 7 required plus one extra data FIFO). However, some students may not want to waste cycles transmitting data that they don't intend to use (such as updates from the seven segment displays). As such, timers for how often data is written to the seven segment FIFOs and LED FIFOs are provided. Setting these timer values to 0 allows students to disable updates to these FIFOs entirely. In addition to supporting an arbitrary number of FIFOs, the Qsys module also contains parameters for adjusting the depth and width of each FIFO. The idea behind adjustable depth is that students may encounter a scenario in which they want to send a burst of updates quickly (within a small number of cycles). If the number of data points exceeds 8, this would exceed the default capacity of the FIFO, and result in lost data. As such, the depth parameter allows for students to customize the depth of the FIFOs to allow for more bursty messaging.

The io_avalon_interface module then connects to the Qsys bus as shown in figure 5. Once again the idea here is that minimal work should need to be done by the students in order to properly setup the Qsys module. As such, two scripts were created to assist in this endeavor. The first is the tcl script which defines the various ports and parameters of the io_avalon_interface module. Here we make use of two interfaces, the first being the avalon streaming interface which we use to communicate between the UART and our serial trans-

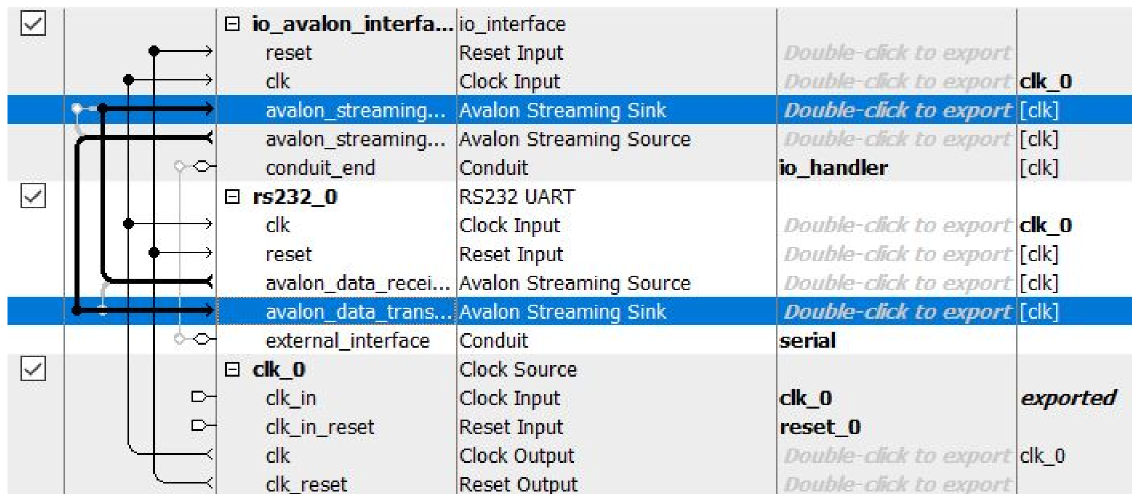


Figure 5: Qsys Bus Connections for io_avalon_interace

mitter/receiver. The second interface we use is a conduit. Conduits in Qsys are essentially ways of exporting wires to the top-level module. As such, the conduit is used in this case to expose the FIFO data lines and outputs from the serial receiver, as described in appendix A.5. The second tcl script handles external pin assignments, specifically for the pins required for the UART as well as the LED and clock. The script deals with assigning wires to their associated pins and setting the required pin voltage. This prevents students from needing to deal with referencing 400+ page documents in their search for the correct pin associations. Armed with these scripts and the Qsys module, getting setup with this project is fairly simple, and the exact steps required to do so can be found in appendix A.2 and A.3.

3.2.1 FPGA - Challenges

The design of data transmission system was the probably the largest challenge encountered during this project. Every component used in this project was designed from scratch, including the FIFO and arbiter. In particular, the arbiter was designed to be quick, making use of parallel prefix OR computation (ppc) to efficiently find the next valid requestor. While this did not prove difficult to implement for a fixed size, determining how to properly create a series of generate statements which could correctly construct an arbitrary size ppc proved difficult. While a solution was eventually found using a combination of generate statements and functions, it does require that the number of FIFOs (the requestors) be divisible by four, creating a small limitation on the configurability of the system.
Another challenge faced while implementing this system was determining how to properly setup parameters for a Qsys module. By default, Qsys provides a module creation tool which will automatically generate a tcl script to describe a HW interface. However, this tool has a number of limitations such as a lack of support for drop down based parameters.

Additionally, using the tool to modify an existing module (such as adding more signals to a conduit) often results in groupings being erased and signals being assigned to the wrong type of interface. As such, it was eventually decided to create a custom script based off of a template created from the tool. This allowed for the customization of parameters as desired.
Furthermore, the use of a customized tcl script also allowed us to navigate around a known bug in version 15.1 of Quartus, in which the module creation tool outputs no useful information when an error in RTL compilation is encountered, resulting in an almost impossible to debug scenario. Useful errors can actually be located by attempting to compile the project outside of the tool provided that Qsys has already generated RTL for top-level instantiation. The issue here arises in the fact that Qsys will not create a tcl script used for generation if the RTL it is defining contains certain errors. As such, having Qsys generate files using a custom script allows for one to get around this issue.

3.2.2 GUI - Design

The final GUI design can be found in figure 10, located in appendix A. Continuing with the theme of ease of use, the GUI was designed to mirror the layout of the physical peripherals on the DE1-SoC board. The idea here is that by mirroring the physical design, students can make a direct correlation between how the peripherals functioned physically, and how they should function now (both on the GUI and the FPGA). As can be seen, by default the GUI includes 6 hexadecimal displays, 10 LEDs, 10 toggle switches, 4 push buttons, a potentiometer, and two system control buttons.
The GUI itself was created using python, making heavy use of the PySimpleGUI library. This library was primarily chosen for two reasons. The first is that the library has decent documentation which was extremely helpful throughout the project and will be useful for students attempting to add their own additions to the GUI. The second is that the library uses an event based system with timeouts. This is desired as it means the GUI will not waste CPU cycles if it is not currently being used, lowering the amount of resources the GUI uses overall. In general, the GUI functions by first waiting for an event to occur (this means a button being pressed, or some sort of interaction with the GUI). While it is waiting, the GUI yields processing time to another task on the CPU. If an interaction occurs within the timeout period, an event is created with an associated "key". This key can be used to identify which element generated the event. For example, in the case of a push button, we would set a key of "pushbut1".

As the goal of this project is to allow students to make use of peripherals remotely, we needed some way of sending GUI data to the FPGA. To do this we made use of the serial library in python along with the event system of PySimpleGUI. Each time an event is generated, a serial message is constructed using the serial protocol defined in appendix A.5.1, and information regarding the event is stored in the message. For example, if a toggle switch is pressed, the number of the toggle switch along with the "toggle" function (as defined in appendix C.2), are encoded and sent in the serial message to the FPGA. A similar process is used for receiving updates from the FPGA (such as updates to the seven

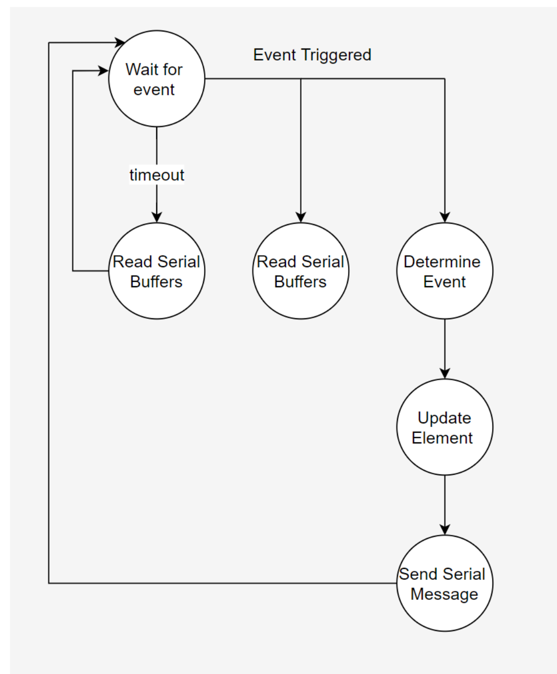


Figure 6: GUI Event Loop

segment display), although it differs slightly. Due to the fact that an event is not generated when a serial message is received, we needed a different way of triggering serial reads. As such, we made use of the timeout mechanism. By setting the timeout to around 50ms, we could ensure that the serial buffer is read every 50ms at the most. If an event is generated before this, the serial buffer will also be read. This ensures that the GUI is being updated with information from the FPGA, even if it is not being interacted with. An overview of the GUI event loop process can be found in figure 6. For more information on running and using the GUI, see appendix A.4.

3.2.3 GUI - Challenges

One challenge faced while designing the GUI was desynch issues between the FPGA and the GUI. For example, the toggle switches send updates to the FPGA when pressed or released. However, the state of the toggle switch on the GUI is lost if the GUI is closed or crashes. As such, if one presses a toggle switch and then closes the GUI, the FPGA will believe the value of that toggle switch is one, while the GUI will believe the value is 0. This wouldn't be a problem under normal conditions; however, when working virtually, there isn't a way to reset the FPGA easily. Thus, we implemented a "sync" button. The sync button sends the current state of each toggle switch to the FPGA along with the "sync" function rather than the typical toggle switch function. This tells the FPGA to put the value of each toggle

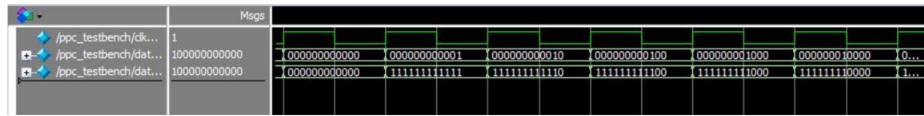


Figure 7: RTL Simulation for the PPC Unit

switch state into the data registers rather than simply flipping the value currently there. This allows for the two devices to synchronize without the need for an FPGA reset.

3.2.4 Arduino

As mentioned, the Arduino is used as a pass-through device for serial communication. Due to the need for serial console access to the HPS connected to the FPGA, a device was needed that was capable of passing serial messages to GPIO wires. Wanting to avoid having to make 20+ USB to pin header adapters, we chose to use an Arduino, which could accept serial messages from the host PC through USB, and pass them to the FPGA through pin headers. As such, the code on the Arduino consists solely of passing the serial messages read from one serial port to the other. As the Arduino used only has a single hardware UART, we make use of the software serial library for implementing the second serial connection.

4 Results and Testing

Testing this project was mainly split into two separate components. The first component involved individually testing the various logic designs created for this project through the use of verilog test benches. As we weren't creating something as complex as a full processor, we decided to mainly make use of unit testing. An example of a unit test for the ppc unit can be found in figure 7. This specific section of the unit test was designed to ensure that a single 1 would be correctly propagated down the chain of nodes in the ppc unit. This was done as previous testing had revealed that the initial generated hardware would sometimes result in incorrect propagation, leading to random combination of 1s and 0s in the output (the output should always be a series of 1s followed by a series of 0s, unless there is no 1 in the input). Similar unit tests were performed for the arbiter as a whole as well as the FIFO design.

The second component in testing this project mainly involved ensuring functionality through observation. As a major component of the design involved a GUI, a large portion of the testing involved confirming the correct output occurred based on what appeared on the GUI. For example, for the purposes of ensuring correct functionality, the top-level logic on the FPGA was designed to tie the output from the toggleswitches on the GUI, to the LED indicators on the FPGA. As the status of each LED is sent back to the GUI periodically, this creates a simple way to ensure that information is being transmitted correctly. Figures 8 and 9 show the GUI and fpga respectively during these tests. As can be seen, a series of toggle switches (9,7,6,4,2) have been activated on the GUI, and their corresponding

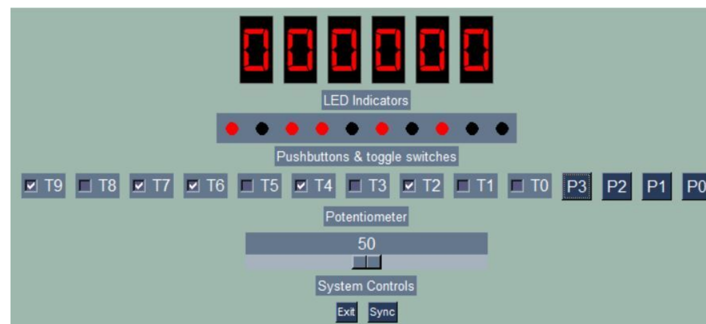


Figure 8: GUI Showing Toggleswitches and the Corresponding LEDs

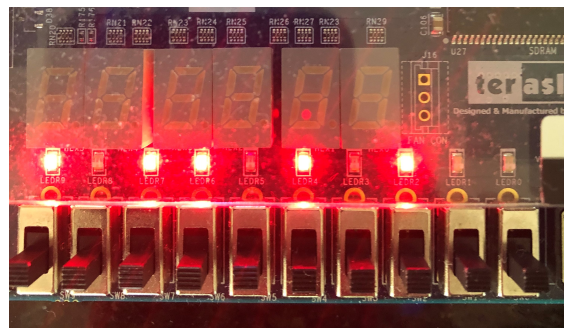


Figure 9: LEDs Mapped to Toggleswitches on the FPGA

LEDs are enabled on both the GUI and FPGA. This helps ensure that the system is working as intended. A similar process was used for testing the pushbuttons, seven seg displays, and potentiometer. Through both methods of testing, we were able to confirm that this system works as intended, and thus it was eventually distributed for student use in ECE 5760.

5 Future Work

While we were able to achieve the goals specified at the beginning of the project, there is still more that could be done to create a more robust product. To begin with, while almost no packet loss was observed during testing (probably due to the relatively low baud rate of 34000 being used), creating some sort of packet acknowledgement and resend system would be beneficial. This would essentially eliminate the chance of losing information while also potentially allowing for the use of higher baud rates.
Additionally, there is currently no method implemented for ensuring data correctness. This poses a slight problem as while the GUI can simply spit out an error and move on, there is no real way to tell exactly how the hardware on the FPGA will behave. We can get around this for now by hard coding safe values (if the action isn't supported, do nothing for example), but having a way to ensure the data is safe means that we can eliminate these

hard-coded checks, improving code cleanliness and maintainability.
Finally, there is nothing currently preventing either device from overflowing the serial receive buffers on both sides. To prevent this, it would be beneficial to add some sort of flow control (such as credits) to ensure that the receiving buffers have enough space to store incoming data. These three improvements would drastically improve the reliability of communication between the FPGA and GUI; however, their implementation could potentially hurt performance. Further work would need to be done to determine the performance impact of these changes and their necessity when attempting to run at higher baud rates.

6 Conclusion

The overall goal of this project was to create a system which would allow remote students to make use of peripherals that are normally only accessible on the physical DE1-SoC board. Along with this, we wanted to ensure that the system was simple for students to add to their own projects, while also allowing for them to add their own peripherals to the GUI and FPGA.
By designing the logic implemented on the FPGA to be able to automatically expand to meet the requirements of the students project, while also making use of a simple Qsys module to provide the hardware interface, we were able to achieve our goals. Adding this project to another takes only a few minutes, and the parameters provided in the Qsys module allow for students to increase the amount of data that can be transmitted with a single click. The GUI is also easy for students to use and mirrors the layout of its physical counter parts on the DE1-SoC board. Future expansion of the GUI should also be relatively easy thanks to the fact that the PySimpleGUI library provides a large amount of documentation and examples. While many students will be returning to campus for Fall 2021, we believe that this project is still potentially useful as a backbone for potential final projects, and are excited to see its use in the future.

Appendices

Appendix A User Manual

This document is intended to provide you with all the information necessary to setup and use the remote IO system. This system consists of three main components, the virtual desktop running the GUI, the arduino acting as a serial pass-through, and the fpga. The steps necessary for setting up the system on each of these devices will be described in separate sections. A detailed description of how the system works is also provided in later sections.
In general, the remote-IO system is meant to provide you with access to IO which you would normally need to be in person to make use of. This includes seven-segment displays, LEDs, push-buttons, toggle switches, and potentiometers. A version of the GUI for this system

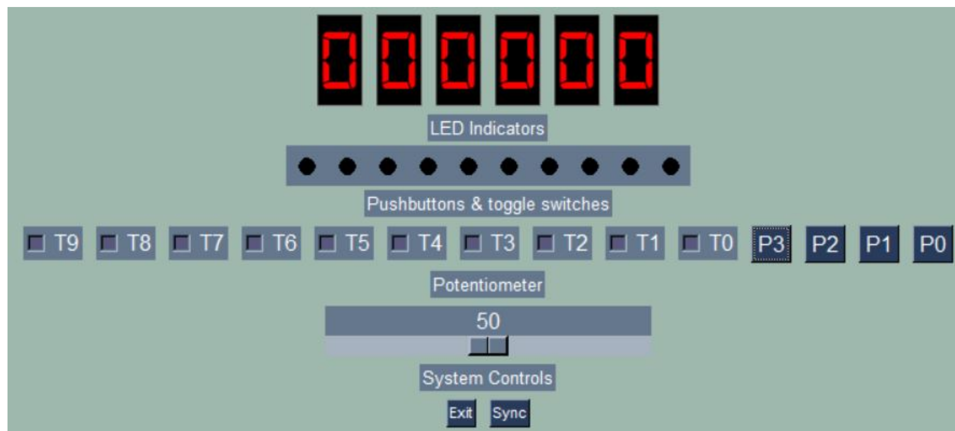


Figure 10: Remote IO GUI

can be seen in figure 10.

A.1 What to Read

This user manual consists of several sections, not all of which you will need to read depending on your situation. There are two primary audiences this document is intended for. The first is those who are using the PIO example project, and the second is for those looking to setup the system from scratch.

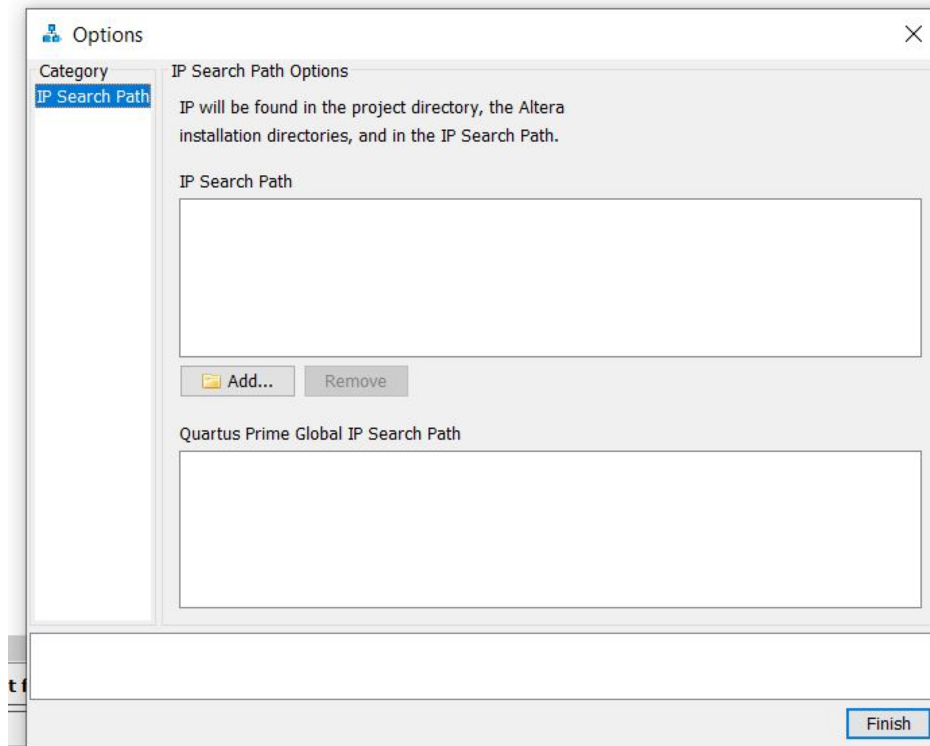
1. **Using example Project**: With the example project, everything should already be setup so there is no need to read through the sections discussing this. However, please do follow the first 4 points in section 2, to ensure that Qsys has found the IP path for the new files. Following this, refer to section 4 for an overview of how to run the GUI and what functionality it currently contains. Section 5 discusses the underlying infrastructure in place if you are looking to eventually add on to this system.

2. **Starting from scratch**: If you are starting from scratch with a brand new project, or simply adding the system to an existing project, then you will want to read sections (2-3) at the very least. Section 2 describes how to setup the necessary components in Qsys, and section 3 describes how to properly configure the top-level of quartus to handle the new Qsys modules. The remaining sections discuss the functionality of the GUI and the underlying infrastructure, so they are not strictly necessary for getting things setup and running.

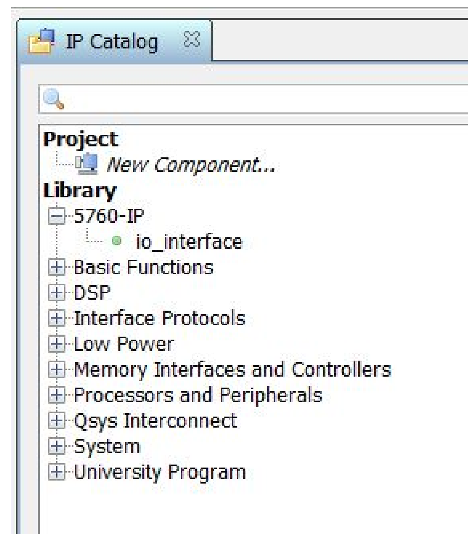
A.2 Adding Components to Qsys

The first step in setting up this system will be adding two new components to QSys. Mainly the io_avalon_interface, and the RS232-UART. If you have not done so already, download the stand-alone zip file from the ECE 5760 website.

1. Within the zip file there should be three folders (GUI, arduino, FPGA). Place the FPGA folder into your existing quartus project folder.
2. Open your current project and go to qsys
3. In qsys go to tools− >options. This should open up a GUI that looks like the following:



4. Click on add and select the FPGA folder that you added to your project folder earlier. Once done click finish. You should now see Qsys scan the folders and if successful, the IP catalog on the left-hand side of Qsys will now show a 5760-IP category as shown in the image below:

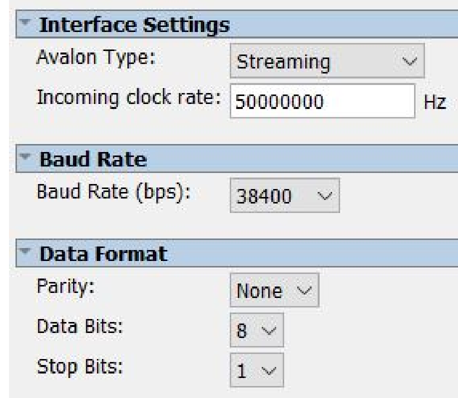


5. The io_interface will be the first component we want to add to Qsys. Go ahead and click on the component and click add. This will open up a tab showing you the block symbol and the parameters. Ensure that the block symbol and parameters match those in the images below

Block Diagram
Show signals
io_avalon_interface_0
reset
clk
avalon_streaming_sink
conduit_end
reset
clock
avalon_streaming
conduit
avalon_streaming
avalon_streaming_source
io_avalon_interface

FIFO Params
FIFO_depth: 8
FIFO_width: 64
GEN_FIFOS: 5
LED Params
LED_timer: 0x0005f5e10
Seven Seg Params
seven_seg_timer: 0x0017d7840

6. There will be a number of errors and warnings but we will fix these later. For now in the IP-Catalog go to University Program → Communications → RS232 UART. In the parameters that appear, change the avalon type to "streaming" and the Baud Rate to "38400". The parameters should now match the ones in the image below.



7. These are the only two new components that will be needed for setting up the system on the FPGA side. For the purposes of saving PLLs we are going to make use of the external CLOCK_50 the FPGA provides. For this step, insert a clock source from basic functions → clocks and plls. Make sure the clock frequency is set to 50MHz. Connect the clock output and reset output of this clock source to the clock and reset inputs of the io_interface and RS232 UART.

8. With the clock and resets setup we now want to export our conduits. A conduit in Qsys is an interface meant for data that isn't part of avalons predfined signal types (memory-mapped signals, val/rdy, etc). For the purpose of this setup, export the io_interface conduit as io_handler and the UART conduit as serial.

9. Your two blocks should now look like those shown in the image below. Note the exported names don't necessarily matter.

Name	Description	Export	Clock
⊟ **io_avalon_interfa...**	io_interface		
reset	Reset Input	Double-click to export	
clk	Clock Input	Double-click to export	**clk_0**
avalon_streaming...	Avalon Streaming Sink	Double-click to export	[clk]
avalon_streaming...	Avalon Streaming Source	Double-click to export	[clk]
conduit_end	Conduit	**io_handler**	[clk]
⊟ **rs232_0**	RS232 UART		
clk	Clock Input	Double-click to export	**clk_0**
reset	Reset Input	Double-click to export	[clk]
avalon_data_recei...	Avalon Streaming Source	Double-click to export	[clk]
avalon_data_trans...	Avalon Streaming Sink	Double-click to export	[clk]
external_interface	Conduit	**serial**	
⊟ **clk_0**	Clock Source		
clk_in	Clock Input	**clk_0**	*exported*
clk_in_reset	Reset Input	**reset_0**	
clk	Clock Output	Double-click to export	clk_0
clk_reset	Reset Output	Double-click to export	

10. At this point we now need to configure the connections between the two blocks. The goal here is to have the streaming source of one block connect to the streaming sink of the other. Make sure that you do not configure the block to feed the data back into itself. The connections should look like the image below

Name	Description	Export	Clock
⊟ **io_avalon_interfa...**	io_interface		
reset	Reset Input	Double-click to export	
clk	Clock Input	Double-click to export	**clk_0**
avalon_streaming...	Avalon Streaming Sink	Double-click to export	[clk]
avalon_streaming...	Avalon Streaming Source	Double-click to export	[clk]
conduit_end	Conduit	**io_handler**	[clk]
⊟ **rs232_0**	RS232 UART		
clk	Clock Input	Double-click to export	**clk_0**
reset	Reset Input	Double-click to export	[clk]
avalon_data_recei...	Avalon Streaming Source	Double-click to export	[clk]
avalon_data_trans...	Avalon Streaming Sink	Double-click to export	[clk]
external_interface	Conduit	**serial**	
⊟ **clk_0**	Clock Source		
clk_in	Clock Input	**clk_0**	*exported*
clk_in_reset	Reset Input	**reset_0**	
clk	Clock Output	Double-click to export	clk_0
clk_reset	Reset Output	Double-click to export	

11. With this complete. The last step is to simply generate the HDL using generate → generate HDL. You can then copy the instantiation template using generate → show instantiation templates. It should look similar to the image below if you are using a new project

```
computer_system u0 (
    .serial_RXD                 (<connected-to-serial_RXD>),
    .serial_TXD                 (<connected-to-serial_TXD>),
    .io_handler_key_out         (<connected-to-io_handler_key_out>),
    .io_handler_psh_out         (<connected-to-io_handler_psh_out>),
    .io_handler_reset           (<connected-to-io_handler_reset>),
    .io_handler_fifo_full       (<connected-to-io_handler_fifo_full>),
    .io_handler_fifo_data_valid (<connected-to-io_handler_fifo_data_valid>),
    .io_handler_seven_seg_1     (<connected-to-io_handler_seven_seg_1>),
    .io_handler_seven_seg_2     (<connected-to-io_handler_seven_seg_2>),
    .io_handler_seven_seg_3     (<connected-to-io_handler_seven_seg_3>),
    .io_handler_seven_seg_4     (<connected-to-io_handler_seven_seg_4>),
    .io_handler_seven_seg_5     (<connected-to-io_handler_seven_seg_5>),
    .io_handler_seven_seg_6     (<connected-to-io_handler_seven_seg_6>),
    .io_handler_led_in          (<connected-to-io_handler_led_in>),
    .io_handler_fifo_data_in    (<connected-to-io_handler_fifo_data_in>),
    .io_handler_pot_out         (<connected-to-io_handler_pot_out>),
    .clk_0_clk                  (<connected-to-clk_0_clk>),
    .reset_0_reset_n            (<connected-to-reset_0_reset_n>)
);
```

A.3 Configuring the Top-Level

This section focuses on instantiating the system defined in the previous step, and configuring quartus to define pins for our external connections.

A.3.1 Instantiating the Qsys system

If you have not done so already, copy the instantiation template from Qsys. This is an incredibly helpful tool that will ensure we do not miss any ports when instantiating our new system. Once copied, paste this template into your top-level design file. We will now begin to setup the connections necessary to get the design compiling.

1. To start with, assign a 1 bit 1 to both of the reset ports. As we won't have anyway of resetting the system manually, there is no need for us to connect an actual button to either reset signal.

2. Connect a 50MHz clock signal to clk_0_clk. If you are using a new project, define this clock signal as an input CLK_50. Make sure this is included in the port list as well.

3. With the clocks and resets taken care of, we now need to connect the serial ports. Define a 1-bit output port TX, and a 1-bit input port RX. NOTE: Don't forget to add these to the port list as well. Connect these two ports to the serial_RXD and serial_TXD connections in our instantiated module. Make sure to connect RX to RXD and TX to TXD. At this point your top-level should look something like the image below. Note: if you are using a new project, define a 10-bit output port LEDR as well. This will be useful for debugging.

```
module DE1_SoC_Computer (
    LEDR,
    CLK_50,
    RX,
    TX
    );
    input         CLK_50;
    input         RX;
    output        TX;
    output [9:0]  LEDR; |

     computer_system u0 (
         .clk_0_clk                     (CLK_50),
         .io_handler_key_out            (),          /
         .io_handler_psh_out            (),          /
         .io_handler_reset              (1'b1),
         .io_handler_fifo_full          (),        //
         .io_handler_fifo_data_valid    (), //
         .io_handler_seven_seg_1        (),      //
         .io_handler_seven_seg_2        (),      //
         .io_handler_seven_seg_3        (),      //
         .io_handler_seven_seg_4        (),      //
         .io_handler_seven_seg_5        (),      //
         .io_handler_seven_seg_6        (),      //
         .io_handler_led_in             (),
         .io_handler_fifo_data_in       (),    //
         .io_handler_pot_out            (),          /
         .reset_0_reset_n               (1'b1),
         .serial_RXD                    (RX),
         .serial_TXD                    (TX)
     );
```

4. For testing purposes we want some way of knowing that data is being passed from the GUI to the fpga, and vice versa. To accomplish this we will make use of the potentiometer and LEDs on the GUI. To start, create an 8-bit wire call POT and connect it to the pot_out connection on our module.

5. From here, assign the first eight bits of the LEDR port to our new pot connection, and then assign LEDR to the led_in connection on our new module. This will not only send the LED information back to the GUI, but also enable the LEDs on the fpga itself. Your top-level should now look like the following:

```
module DE1_SoC_Computer (
    LEDR,
    CLK_50,
    RX,
    TX
    );
    input         CLK_50;
    input         RX;
    output        TX;
    output [9:0] LEDR;

    wire [7:0] POT;

    assign LEDR[7:0] = POT[7:0]; |

     computer_system u0 (
         .clk_0_clk                   (CLK_50),
         .io_handler_key_out          (),
         .io_handler_psh_out          (),
         .io_handler_reset            (1'b1),
         .io_handler_fifo_full        (),
         .io_handler_fifo_data_valid (), //
         .io_handler_seven_seg_1      (),      //
         .io_handler_seven_seg_2      (),      //
         .io_handler_seven_seg_3      (),      //
         .io_handler_seven_seg_4      (),      //
         .io_handler_seven_seg_5      (),      //
         .io_handler_seven_seg_6      (),      //
         .io_handler_led_in           (LEDR),
         .io_handler_fifo_data_in     (),     //
         .io_handler_pot_out          (POT),
         .reset_0_reset_n             (1'b1),
         .serial_RXD                  (RX),
         .serial_TXD                  (TX)
```

A.3.2 Pin Assignments and Project Files

1. If you were to try compiling this design now. You will notice that quartus will complain that there is no module "computer_system". This is because we need to add a file generated by Qsys to our project. To do this goto Project →Add/Remove Files. Once in this gui click the small .. icon and browse to the following file: computer_system/synthesis/computer_system.qip . Once selected make sure to click the add button near the top of the GUI and then click ok. You should now see the qip file listed in the project files. The qip file is a "quartus IP file" and specifies to quartus the file paths for all the verilog files associated with the IP it was generated for. This is useful as we do not need to manually add each file being used in qsys to the project.

2. From here, our next step is to run analysis and elaboration. We need to do this in-order to allow quartus to see what ports the top-level module has declared. This will be important for our next step. To run analysis and elaboration go to processing → start → analysis and elaboration.

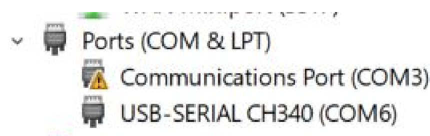
3. Now that we have run analysis and elaboration, we want to make use of the pin planner to assign pins to our RX, TX, CLK_50, and LEDR ports. To save the time and trouble, there is a tcl script prepared already which will do these pin assignments. Navigate to tools → tcl scripts. Once here find the FPGA folder and select pin_assign.tcl With the script open click run and it will automatically assign the pins and the necessary voltage levels. If you now go to assignments → pin planner, you should see that every pin has a location and 3.3V I/O standard NOTE: if you are not working from a new project but instead are using the existing DE1_SoC_Computer you should only need to go to assignments → pin planner and assign pins for RX and TX like in the following image. It should be noted that these pins may already be taken by GPIO_0[0] and [1]. Remove those assignments.

RX	Input	PIN Y17	4A	B4A N0	3.3-V LVTTL
TX	Output	PIN AC18	4A	B4A N0	3.3-V LVTTL

4. At this point you can now compile your project and there should no longer be any errors. Once you have finished compiling, program the FPGA.

A.4 Running and using the GUI

At this point everything should be ready to go on the FPGA side. The only remaining steps are to check which COM port the arduino is connected to, and ensure the GUI program is attempting communication with this port. To begin check the COM port the arduino is on by going to device manager and scrolling down until you see the COM ports section. Here you should see a device listed as USB-SERIAL CH340 like in the image shown below. Make sure that the COM port that appears here matches the COM port specified on line 12 of gui.py and gui-fpga.py found in the GUI folder. It is recommended at this point to open the arduino code found in the arduino folder of the zip file, and flash the code to the board (remember to select the proper COM port in the arduino IDE).



Open a terminal of your choice and navigate to where you unzipped the GUI folder. You should now be able to run gui-fpga.py using the command (python gui-fpga.py). This should open a GUI similar to the one seen in Figure 1 on page 2. NOTE: if for some reason you get a python error talking about serial not having a write object, this means that the program failed to connect to the arduino. Once again, make sure that the COM port specified is the one for the arduino. Note, if you use the arduino IDE to open the ”serial console”, this will prevent the GUI from being able to use the COM port. You cannot have both the GUI and ”serial console” open simultaneously.
With the GUI now running you should be able to make use of the various buttons and switches available.

A.4.1 Functionality of the GUI

Here we will briefly talk about the functionality current available on the GUI. The image in Figure 1: shows the current GUI that opens when using gui-fpga.py.

1. **Seven-Segs**: The seven segment displays can be used to display information sent from the FPGA. Simply sending a binary encoded digit (0-15) to any of the seven_seg inputs will result in the data eventually being displayed on the corresponding seven segment display here on the GUI.
   These displays are actually just text elements which use a seven segment font, so adding additional displays for other uses should be simple. The font used also includes letters if one wanted to expand the functionality to allow for hex displays.

2. **LEDs**: The LEDs on the GUI are tied to the led_in port of the qsys module on the FPGA. When a bit on the input port is set high, the corresponding LED on the GUI will change from black (off) to red (on).

3. **Toggle Switches**: The toggle switches can be used just like the normal toggle switches on the fpga. Pressing a toggle switch will place a check-mark in the box and will update the key_out port to have a 1 for the corresponding switch. When the switch is pressed again, the bit will be set back to 0.

4. **Push Buttons**: the push buttons work similarly to the toggle switches. When the button is pressed and held down, the corresponding bit on the psh_out port is set to 1. When the button on the GUI is then released, the bit is set back to 0.

5. **Potentiometer**: The potentiometer sends an 8-bit value to the FPGA. This allows the user to act as if they have the output of a potentiometer hooked-up to an ADC. As such, by adjusting the slider, the port on the FPGA side will change accordingly. The example program created earlier in this document ties the LEDs on the GUI to this potentiometer so that we can see the values changing.

6. **Sync Button**: The sync button is a way of ensuring that the current state of the GUI is correctly represented on the FPGA. For example, if the GUI is closed or crashes when a toggle switch is currently high, this high state will remain on the FPGA, but the switch will appear to be low when the GUI is restarted. Pressing the sync button will send the status of the potentiometer and all toggle switches to the FPGA, so it can update its outputs accordingly.

A.5 Quartus Interface - Details

This section is meant to give an in-depth explanation of the system on the FPGA. It will mainly focus on explaining the currently implemented blocks and features, and describe the various ports currently defined in the top-level. This section is intended to allow the reader to gain an understanding of how to use the interface currently provided.

Port	Length	Description
clk	1-bit	Provides a clock signal for the system to use
reset	1-bit	reset signal for everything in io_interface
reset_n	1-bit	reset signal for clock generator
key_out	10-bits	contains the state of each toggle switch
psh_out	4-bits	contains the state of each push button
pot_out	8-bits	contains the number displayed on the potentiometer of the GUI
fifo_full	Variable	contains the full flag for each FIFO
fifo_data_valid	Variable	used to indicate data on the input of a FIFO is valid
fifo_data_in	Variable	used to insert data into each FIFO
seven_seg(1-6)	4-bits	used to insert data into a seven seg FIFO
led_in	10-bits	used to insert the state of each LED into the led FIFO
RXD	1-bit	UART serial input
TXD	1-bit	UART serial output

The table above gives a list and short description of the various ports on the io interface. Here, we will discuss a few of the ports needed to interact with the remote io system.

1. **seven_seg**: The seven_seg ports are meant for sending data to the GUI. Each port is used for sending data specific to that seven_seg display. (0 being the rightmost display on the GUI). It is important to note that GUI expects decimal numbers encoded as binary, NOT seven seg controls. Thus if you want to display a 2, simply put an 4'b0010 onto the seven_seg port. There is no need for a seven segment controller.
   The rate at which the data on these ports is sampled is based on the seven seg timer parameter that appears in Qsys. As the clock we are using is a 50MHz clock, the default seven seg timer is set to 25,000,000. This means the data on the port is sampled and sent to the GUI every half second.

2. **led_in**: The led_in port is similar to the seven_seg port and is meant for sending the state of each LED to the GUI. Unlike the seven_seg ports however, there is only one LED port. The led_in port is 10 bits wide and as such, each bit is used to encode the status of one LED. If you want to say that LEDs 0 and 1 are high and the rest are low, simply send a 10'b0000000011.
   Once again, the rate at whch the data on this port is sampled is based on the led timer parameter that appears in Qsys. The default value is set so that the status of the LEDs is sent every eighth of a second.

3. **pot_out, psh_out, key_out**: These three ports are all fairly similar, and allow you to use the push buttons, toggle switches, and potentiometer as if you were in person.

pot_out is an 8-bit value which contains the binary equivalent of the value displayed on the GUI. key_out and psh_out represent the values of each push button and toggle switch on the GUI (high or low). This allows you to read these values just as easily as you would normally read the value of the buttons and switches.

4. **fifo_data_in, fifo_data_valid, fifo_full** the FIFO ports are the more complex part of the interface and are meant as a way for you to extend the functionality of the GUI. Lets say you wanted to add a new feature to the GUI (graph for example) and you want to send data to that graphing element, these FIFO ports are the way for you to do that.
The first thing to note about the FIFO ports is that they are variable in size. The size of each port is determined by the GEN_FIFO parameter from Qsys. This parameter allows you to easily change the number of general FIFOs without having to modify the underlying infrastructure. It should also be noted that you can change both the amount of data a FIFO holds (FIFO_DEPTH) and the width of each piece of data (FIFO_WIDTH). However, the system will not allow you to have a depth less than 1 or a width less than 32. **NOTE: DO NOT CHANGE THE FIFO WIDTH AT THIS TIME**
The width of the fifo_data_in port is $(64 \times GEN_FIFO)$. Every 64 bits will be the data in for a separate fifo. Thus if you want to send data into two fifos, one assignment would be from [63:0] and the second would be from [127:64]. The size of fifo_data_valid and fifo_full simply scale with GEN_FIFO. To push data into a FIFO you must set the corresponding fifo_data_valid bit on the clock cycle you would like to do so.
It is NOT recommended to leave the valid high, as you will quickly fill the FIFO and begin to lose data. The fifo_full bit corresponding to each FIFO can be used to determine when a fifo is full and is a helpful way of providing flow control. It should be noted however that sending data whenever the FIFO is not full may also be problematic since we are relying on the arduino to pass messages, and it may not be able to handle sending and receive data at extremely high rates. As such it is recommended to use a timer system to send messages every so often.

Hopefully from the explanation of the various ports you should now have an idea of what data is available on the fpga side and how you can send data to the GUI. But what about handling new data sent from the GUI to the fpga? The following section describes the serial protocol currently in place and how you can add to it to support new functionality.

A.5.1 Serial Protocol

The serial protocol currently implemented is fairly basic and mostly exists to provide a way to frame the data being sent back and forth. The diagram below shows how data is broken up in the existing protocol.

```
#packet layout
#-----------------------------------------------------------
# start_char | type | payload_length | payload | end_char
#-----------------------------------------------------------
```

1. **0x80 : start_char** The start_char is used to indicate to serial receivers that a message is beginning. This allows us to properly begin receiving a message without needing some sort of additional reliability mechanisms or handshakes
2. **type**: The type byte is used to indicate to a receiver what device/function a message is meant for. Receivers will then handle the payload accordingly based on the type of the message. A table of the currently supported message types can be seen in appendix C.1
3. **payload_length** The payload length byte is used to indicate to the serial transmitter how long of a message is to be sent. The transmitter will then send that many bytes followed by an end_char. This is also used by receivers to check whether an error in transmission occurred (the payload length field doesn't match with how many bytes were received before end_char was detected).
4. **payload**: The payload is dependent upon the type of message. Typically for a push button, the payload will contain which push button it is and what event is occurring (either a press or release). A table of events/functions can be seen in appendix C.2
5. **0x81 : end_char** The end_char is used to allow serial receivers to know that the current message has finished being received. Alongside the payload length this is one way of ensuring some sort of reliability since there is currently no re-transmission or check sums implemented.

A.5.2 System Infrastructure

This section will describe underlying infrastructure on the FPGA side. The goal here is to provide an understanding of how everything is working together for those who wish to expand upon its functionality or correct any errors.

A.5.3 Data Transmission

Appendix C.3 Shows an overview of the io_interface block. As can be seen the general infrastructure revolves around a series of FIFOs. These FIFOs take in data from fifo_data_in as well as the seven segment and LED data. The empty bits of the FIFOs are then combined to form the request vector of the round robin arbiter. This is a relatively fast arbiter which allows us to chose which FIFO to pull data from in a single cycle. The arbiter is also work conserving so there are no additional cycles when looping back around the last FIFO to the first one. The grant of this round-robin arbiter feeds into a

multiplexer which is then used to send data to the serial transmitter. It should be noted that data is only popped from a FIFO when the serial transmitter is ready to receive new data.
For those looking to expand on the existing data the FPGA sends, it is as simple as adding more general FIFOs via the parameters described in previous sections. The point of describing the transmission process is to present its weaknesses so that they can be avoided. Currently it takes roughly 0.2 milliseconds (based on 38400 baud) to send a byte of data (or roughly 1.6 milliseconds to send a full 64-bit packet). As such, data can only be taken from a FIFO roughly once every 1.6 milliseconds. If every FIFO in the system has data, it would take $(1.6 \times N)$milliseconds to make one round trip of all the FIFOs. If the rate at which data is entering the FIFOs increases beyond this point, the FIFOs will begin to fill up and eventually data will be lost. To avoid this, it is recommended to make the rate at which you enter data into the FIFOs much slower, or make use of the fifo_full port.

A.5.4 Data Receiving

The serial receiver is where data being sent to the FPGA is handled. Data handling begins when the receiver reads in a start_char. Data is then continually stored in internal registers until an end_char is found. Once this occurs the serial receiver looks at the type of message, payload length and function and determines what to do with the data. In the case of toggle switches for example, it will change the state of key_out depending on which switch number was in the payload and if the function byte in the payload is a switch event. If the function byte was instead a sync event, a different action occurs. For those looking on how to modify the system to allow the FPGA to receive new events, this is where you will need to begin.

A.5.5 Modifying the System

Appendix C.4 shows the FSM for the serial receiver. The "PROCESS" state is where the serial receiver processes a received packet. In order to handle new data, you will need to add a new statement to this state in (serial_receiver.sv). Use the other statements as a guide on how to process your own data, as a note, the start character and end characters are not a part of the data being stored for processing. You will likely need to add additional ports to both (serial_receiver.sv and io_avalon_interface.sv) in order to get new data to the top-level module. When making theses change it is important to note that you will need to modify the io_interface Qsys component in-order to get the changes to take effect. There are two ways you can accomplish this.

1. **Qsys Component Editor**: The component editor is a great way of editing Qsys components; however, I would not recommend using it for simply adding new ports to the conduit interface. This is due to the fact that when adding new ports, it tends to mess up the interfaces and reset parameter groupings. Additionally in Quartus 15.1 the component editor is bugged, and any syntax or verilog issues (even a missing semicolon) will result in a "null module error" with no other provided

information. If you would still like to use it, you can simply run analyze synthesis files in the files tab, and that should find your new ports.

2. **TCL file editing**: This is probably the simpler way to add a new port to a conduit. In the qsys folder which lives in the FPGA folder, you will find (io_avalon_interface_hw.tcl). This is the tcl script that tells Qsys which ports exist, what size they are, what parameters are there, etc. Lets say I wanted to add a new port called "NEW_IN" all I would need to do is add the following line to the bottom of the file.
   `add_interface_port conduit_end NEW_IN new_in` Output 8
   The above line tells Qsys that we want our internal port declaration of NEW_IN to become new_in when it gets exported via the conduit. Additionally, it tells Qsys that this port should be 8 bits in width. When you then generate the HDL in Qsys, it will take care of creating this port and you can add it to your top level instantiation.

A.6 Modifying the GUI

With an understanding of how data is sent and received on the FPGA side, lets now take a look at how we can modify the GUI to add new functionality. The GUI uses the python GUI library known as PySimpleGUI. It is recommended that one take a brief look at the examples and documentation provided by PySimpleGUI before attempting to modify the GUI. From reading the documentation you will discover that PySimpleGUI handles updates to the GUI through an event system. When a button is pressed for example, an event will be triggered, with the event name being equal to the "key" defined in the creation of the button. We use these events to then take an action. In the case of a button, we send a serial message corresponding to either the press or release of a button using the python serial library. If you would like to add addition functionality to the GUI, you will likely need to follow a similar process. Currently, the serial protocol byte definitions occur in protocol.py
In terms of handling new data being sent to the GUI, one should only need to add a new case to the handle_serial function. Previous functions should have already taken care of reading in a serial message and storing it in the serial_storage array. Once again the start and end bytes are not stored in this array, so keep this in mind when indexing.

Appendix B References and Documents

This section is intended to go through various references/documents that were used in this project, and that may be helpful to anyone looking to create a similar project. A list of references can be seen in Appendix C.4.

B.1 Project Setup Guide

Appendix C.4.1 includes a link to a DE1-SoC project setup guide. This guide goes through what is required to setup a project from scratch in Quartus, including creating the empty project, doing pin assignments, configuring the top-level, and more. This guide was made for more recent versions of Quartus; however, most of what it describes is still relevant to Quartus 15.1

B.2 DE1-SoC User Manual

Appendix C.4.2 includes a link to a DE1-SoC user manual. This manual includes a lot of information on the usage of the board, from the various peripherals it has, to several example projects showing how to use the FGPA and HPS together. Chapters 2 and 3 were particularly useful for this system.

1. Chapter 2 shows the overall diagram of the board along with a diagram which shows which peripherals are connected to the FPGA, and which are connected to the HPS. This is useful as it allows one to see what devices the FPGA has access to without needing any sort of pin muxing in Qsys.
2. Chapter 3 goes into detail on each peripheral connected to both the FPGA and HPS. In cases where applicable, the manual also provides a circuit diagram of the peripheral, which is helpful in seeing what the default state may be (pull-ups to 3.3V in the case of the push buttons). Along with descriptions of each peripheral, the manual also provides the list of associated pins for use in the pin planner. This saves one the trouble of trying to figure out the pins arrangements themselves.

B.3 Quartus Handbook

Appendix C.4.3 includes a link to the intel Quartus II Handbook for version 15.1. This is a rather massive 1600 page guide which covers pretty much every detail about Quartus that you would want to know. Here we will mention a couple of sections that were particularly useful.

1. **Section 5: Creating Qsys Components - Page 265** This section discusses how to create a Qsys component in extreme detail, so what in here is useful to take a look at? For starters take a look at table 5-1. This table provides a description of each interface type available in this version of Qsys. If you are looking to create a new Qsys component, understanding the various interfaces and their use cases should help guide you in determining what you need to make your design function as intended. Here we will provide a brief description of each interface to help clear up any confusion:
   (a) **Memory-Mapped**: The Avalon memory mapped interface is mainly meant to be used between a ”host” and a series of ”slaves”. Each slave device will

consists of some sort of write-able or readable RAM/memory which is given an associated address range in Qsys. When the host wants to read from or write to these slave devices, it will specify an address on the Avalon bus, along with a serie of other signals. You can think of this interface as being similar to an I2C protocol, so if you are looking to setup communications between a master, and multiple other devices, this would be a way of doing that.

(b) **Streaming (ST)**: The Avalon Streaming interface is used for unidirectional transfer of data from one device to another. What this means is that data flows from the data source port of one device to the data sink port of another. While the interface is "unidirectional", a device can have both a data source and sink port, allowing for it to simultaneously read data in, and write data out. In terms of flow control, the interface is different from the memory mapped interface in that is uses a simple val/rdy protocol to allow device to know when data can be sent or received. If you only need to communicate from one device to another, this is the way to go.

(c) **Conduit**: The conduit interface is different from the other two interface mentioned so far, in that it is more of a way to provide data sharing, rather than data communications. What is meant by this is that data put on a conduit is just like any other wire you might create in a file. The conduit interface has no val/rdy signls, or write enable signals, so it is up to the hardware on both sides to know when to sample data. As such, the conduit interface is meant to be used as a simple input/output port rather than a communications port.

The remainder of this section goes into great detail on the various types of Qsys components, and the ways to create them. For the purposes of simplification, there are really two main ways of creating and editing components in Qsys. Both of these methods were mentioned earlier in this document in section A 5.2.3, but we will describe them in a little more detail here.

(a) **Qsys Component Editor**: The component editor is a really useful tool for getting started on creating a new Qsys component. It is particularly useful in that it will allow you to generate a TCL script for a component simply by analyzing the HDL files you specify for your component. By then making use of the parameters and signals tabs, one can configure and create an entire Qsys module without the need for editing an TCL scripts. Additionally, page 278 of the Qsys handbook describes the various prefixes that can be used for naming ports in a HDL file. Using a prefix in the format of ¡interface type prefix¿_¡signal type¿ will automatically tell the component editor to assign that port to the interface of that type. This will make ones life much easier when using the component editor. Unfortunately however, the component editor does have its limitations. The parameter system that Qsys has is very cool, but its abilities are limited to assigning parameters to groups if you are using the

component editor. For anything outside of this, you will need to use a TCL script.

(b) **TCL Scripts**: TCL scripts are used in Qsys to define the various interfaces, parameter configurations, display items, etc, that a module contains. As mentioned before, the component editor actually generates a TCL script when it is used, so it is recommended that one uses the component editor when first creating a component. However, once that initial TCL script has been generated, we can modify the TCL script to allow us to have more flexibility. As an example of what we can do, let say that you want to limit the range of a parameter, so that it can't take just any random value. We can accomplish this using the parameter property ALLOWED_RANGES, which is shown on handbook page 286. Furthermore, lets say you wanted to make a "optional port or interface", you can also accomplish this through modifications to the TCL script, like the ones towards the bottom of page 288 of the handbook. For those interested, appendix C.4.5 links to the section of the Qsys/Platform Designer documentation which discusses all of the available properties that can be used in scripting.

Outside of the ways to make a component and the types of interfaces, the section also goes into a lot of other details (including how to create composed modules / subsystems, dynamically generated parameters, and more), but those are topics that were not necessary for this system, and so will not be discussed here.

Appendix C User-Manual Appendix

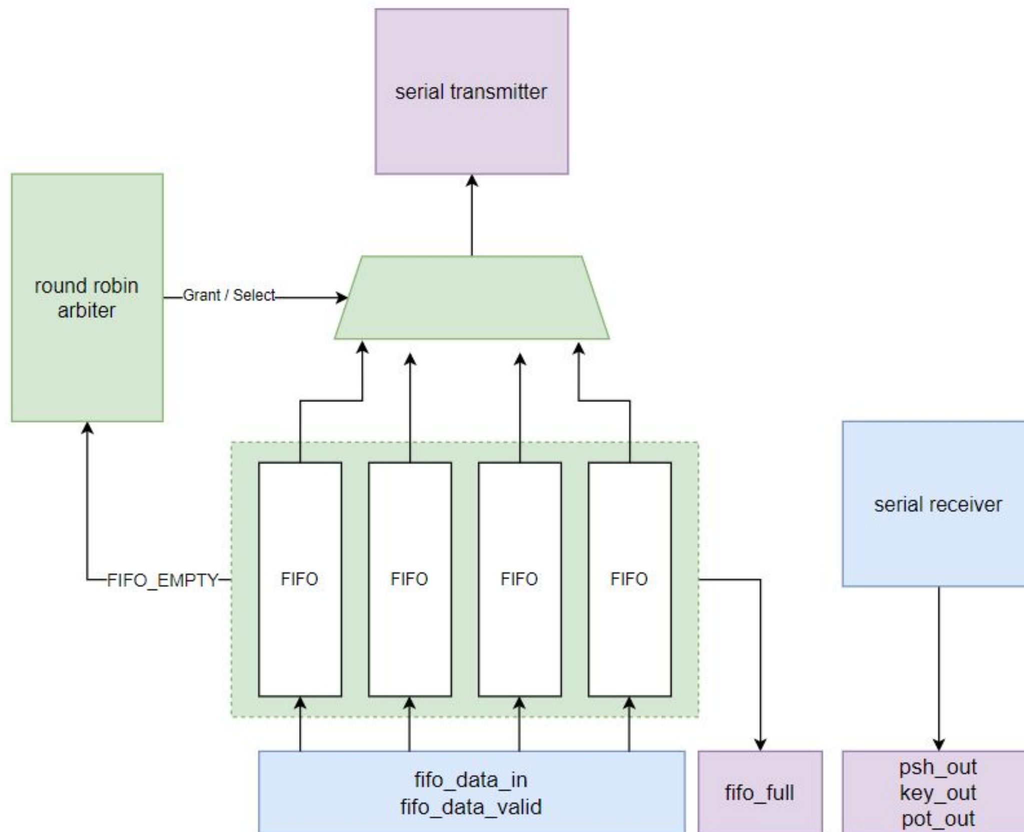
C.1 Serial Protocol Types

Char	Type
0x82	push button
0x83	toggle switch
0x84	key pad
0x85	seven segment
0x86	LED
0x87	potentiometer

C.2 Serial Protocol Functions

Char	Function
0x70	button press
0x71	button release
0x72	toggle (toggle switch)
0x73	sync

C.3 FPGA System Overview



C.4 References

C.4.1 DE1-SoC Project Setup Guide

PDF Project Setup Guide

C.4.2 DE1-SoC User Manual

PDF DE1-SoC User Manual Revf

C.4.3 Quartus 15.1 Handbook

PDF Quartus Handbook

C.4.4 Avalon Interface Specification

WEB Avalon Specification

C.4.5 Platform Designer (Qsys) Property Specification

WEB Property Specification