6.111 Final Project: Internet Controlled Robot

By: Babuabel Wanyeki and Brandon Perez

2020

1 Introduction **3**

2 Overview/Setup **4**

2.1 Robot 4

2.1.1 Parts 4

2.1.2 Setup Overview 5

2.2 Controller 5

2.2.1 Parts 6

2.2.2 Setup Overview 6

3 Data Structures **7**

4 Networking **8**

5 Ethernet Interface(Babu) **8**

5.1 Ethernet Reception 10

5.2 Ethernet Transmission 11

6 Motor Control(Babu) 13

7 Distance Sensor(Babu) 14

8 Camera Sensor(Babu) 15

9 Controller Top Level - (Brandon) 16

10 VGA Display - Numbers(Brandon) 18

11 VGA Display - Camera(Brandon) 19

12 Development Process 21

12.1 Robot 21

12.2 Controller 22

13 Ideas/Conclusion 23

14 Verilog Source Code 24

14 Verilog Simulation Code 83

15 Python Server Code 87

16 Python Application Code 90

16.1 Robot 90

16.2 Controller 95

1 Introduction

COVID-19 has forced us to be apart during these times, but we figured we could take advantage of this opportunity for our project. My partner and I live in different states right now, so we thought that it would be interesting to be able to have one person control a robot in another person's room across the United States. Additionally, the robot will also be able to send sensor data through the transmitter which the controller would be able to receive and display the data using the VGA interface. Another reason that we chose to do this project is because it allowed us to have a lot of learning opportunities throughout the semester, one of which being a deep understanding of how Ethernet UDP protocol works.

The central idea of the two projects is that the two FPGAs, the controller and the robot, will be able to communicate through a shared server on the internet. How this will work is that the respective FPGA is connected to a PC via Ethernet(to enable high speed ~100 Mbps reception and transmission), this means that we will be able to send video, sensor, and control data appropriately. Second the corresponding PC will have to run a Python program, which we created, that enables data to be sent on the server. Similarly, the other FPGA will also need to be connected to a PC via Ethernet and the PC needs to be running a similar Python program. Once this is done a reception and transmission connection should be established between the two FPGAs, allowing video, sensor, and control data to be transmitted and received at the same time.

The controller will use the up(btnu), down(btnd) ,left(btnl), and right(btnr) buttons on the FPGA in order to send control data packets to the robot. Additionally the controller will also process all of the video and sensor data from the robot and display it on the VGA. The robot on the other hand receives the controller data and translates that to forward, backward, left, or right commands on the FPGA. This then gets sent to the motor driver for directional movement. Additionally the robot takes in distance sensor data and camera pixel data and transmits that to the controller FPGA for processing and display. In the end we have a working system that enables the controller to control where the robot is moving while also using camera data and proximity data to see where the robot is going.

2 Overview/Setup

2.1 Robot

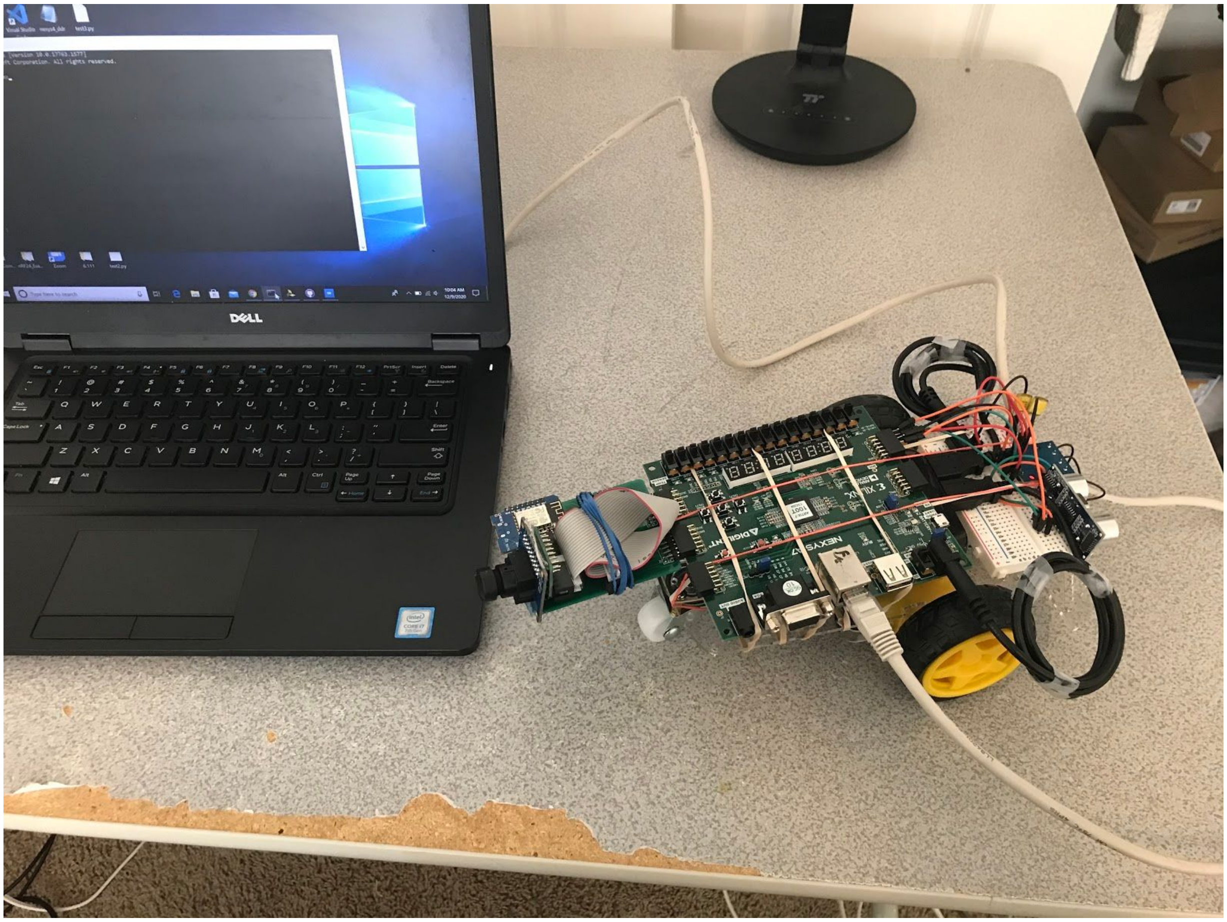


Figure 1: Full robot setup with an ethernet connection to the PC

2.1.1 Parts

- Nexys4 DDR FPGA
  - SMSC 10/100 Ethernet PHY (SMSC part number LAN8720A)
- Windows PC
- Portable Battery
- Solderless Breadboard with a DC power jack
- Robot Car Chassis with 2 motors
- OV7670 Camera

- ESP8266 Microcontroller
- HC-SR04 Distance Sensor
- L9110 Motor Driver

2.1.2 Setup Overview

The FPGA is mounted on top of a portable battery character which is then placed on a robotic car chassis. The robot chassis contains 2 motors in the back as well as a third wheel in the front for balance. Additionally, on the back of the robot chassis is also a small solderless breadboard which contains the connections to and from the FPGA PMOD pins and the HC-SR04 distance sensor, the L9110 motor driver, and the DC power jack.

In order to set up the robot, first connect the portable battery to the DC power jack on the solderless breadboard. Next, also connect the portable battery to the FPGA, making sure that the power is derived from the DC jack rather than a USB connection. Once finished, program the FPGA with the compiled bitstream, and make sure that the FPGA is connected via an Ethernet cord to the WIndows PC. At this point no LEDs should light up.

Next, make sure that either the robotic user side or the controller user side turns on the Internet server and runs the communications.sh bash file, which essentially starts the two server Python files(RobotRX_ControlTX.py and RobotTX_ControlRX.py) concurrently. Lastly, run the reception.py file and transmission.py file in order to receive and transmit the UDP data packets to and from the servers. At this point 2 LEDs on the board should light up blue meaning a connection to the reception and transmission servers. Now the robot is ready to communicate with the server. Once the controller side programs their FPGA and runs the reception.py and transmission.py python programs, then you can exchange data with the other FPGA anywhere in the world.

2.2 Controller

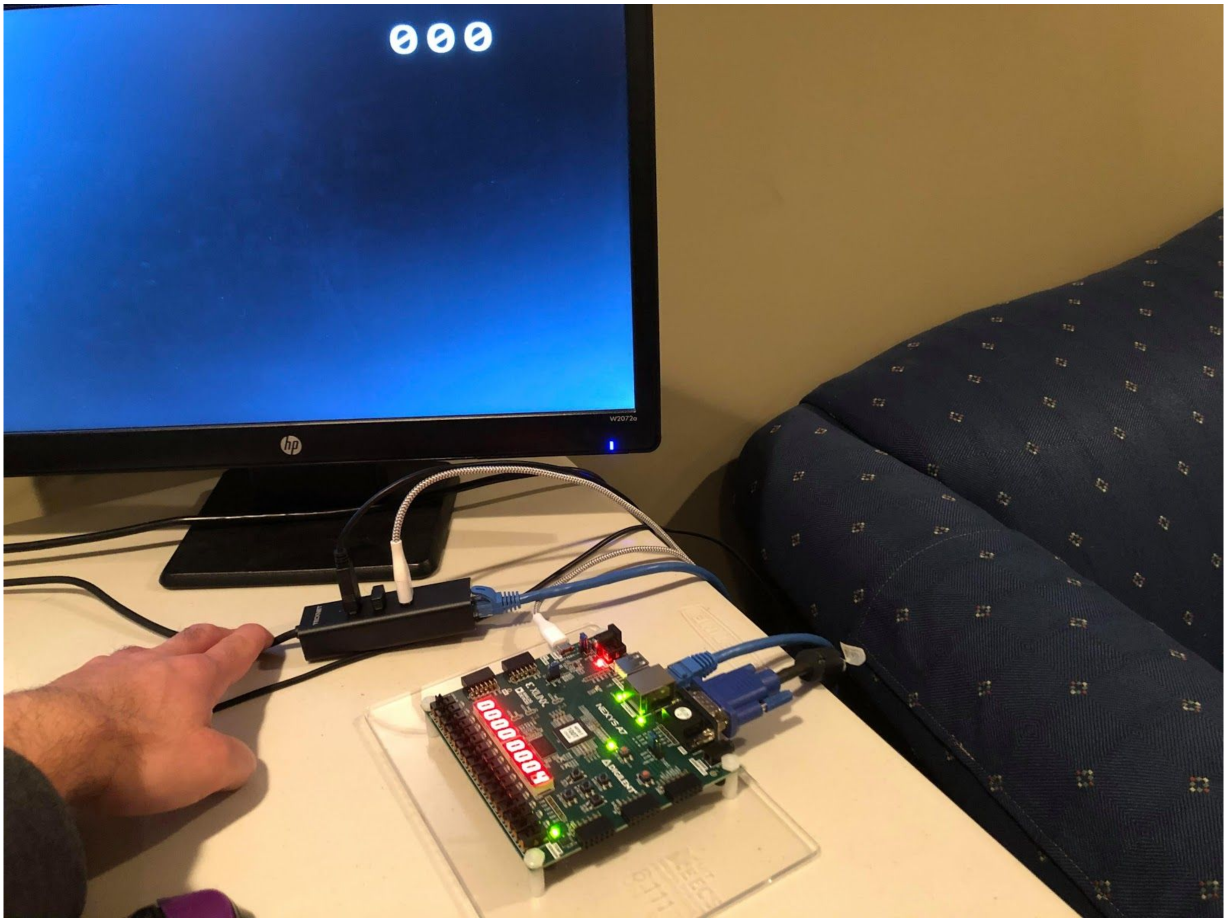


Figure 2: Full Controller Setup with a VGA monitor

2.2.1 Parts

- Nexys 4 DDR
  - 4 directional buttons(btnu, btnd, btnl, btnr, and btnc for resetting connection)
- Windows PC
- VGA Monitor
- Ethernet/USB adapter
- Ethernet cable

2.2.2 Setup Overview

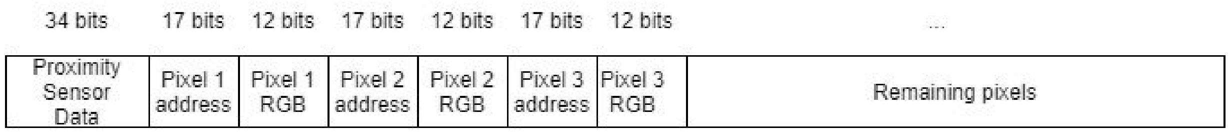
I used a USB hub with a built in ethernet adapter to allow for a connection between my PC and the FPGA board. I was concerned that a 3rd party adapter would lead to problems (like dropping UDP packets), but it turned out to be quite capable for our purposes. Though the robot side ethernet cable should be long enough to support mobility of the robot, the controller side ethernet can be 3ft or less as long as the FPGA board can be placed close to both the computer and monitor. The rest is a basic VGA display setup similar to lab 3.

Concerning the servers, I was unable to use the bash script even with a windows version of bash installed on my pc. Beware of mac/pc compatibility. So, instead I would have to start the above-mentioned transmission.py and reception.py scripts separately. We had the least trouble when I performed the connection setup in the following manner. Have one of us start the servers. I would then run the reception.py script. Connect the ethernet between the adapter and FPGA. The blue reception LED would consequently light up. Then I would run the transmission.py script and both reception and transmission LEDs would turn on.

3 Data Structures

The two most prominent data structures that must be addressed are those of the two payloads (both transmitted and received). On the controller side, the transmitted payload contains the robot command data, and the received payload contains the camera/sensor data. On the robot side, the transmitted payload contains the camera/sensor data, and the received payload contains the command data. Let us name these 2 different structures the sensor payload and the command payload. Both payloads are 548 bytes (4384 bits) for consistency, and in order to maximize the pixel data transmitted from the camera as the maximum payload allowed for UDP packets is 548 bytes.

The sensor payload is formatted in the following way. The first 34 bits contain the sensor data. Though the maximum reading from the sensor was approximately 400, only requiring 9 bits, allotting 34 bits allowed for the camera data to evenly fit into the remaining bits. The camera portion of the payload allowed for data for 150 pixels. Each pixel had a 17 bit address (calculated from the row and column position to be used for writing to a RAM), and a 12 bit rgb value. In total, each pixel took up 29 bits. 29*150 = 4350, which is also equal to the total number of bits minus the sensor data bits (4384 - 34). See the diagram below.



34 bits	17 bits	12 bits	17 bits	12 bits	17 bits	12 bits	...
Proximity Sensor Data	Pixel 1 address	Pixel 1 RGB	Pixel 2 address	Pixel 2 RGB	Pixel 3 address	Pixel 3 RGB	Remaining pixels

Figure 3: Data structure of how the Distance Sensor and Pixel Data are being sent.

The command payload is much simpler. The commands from the controller side are at most a 3 bit number, so one only has to assign the first 3 bits of the payload to the output of the top level controller module. For brevity's sake, a diagram is not included for the command payload.

4 Networking

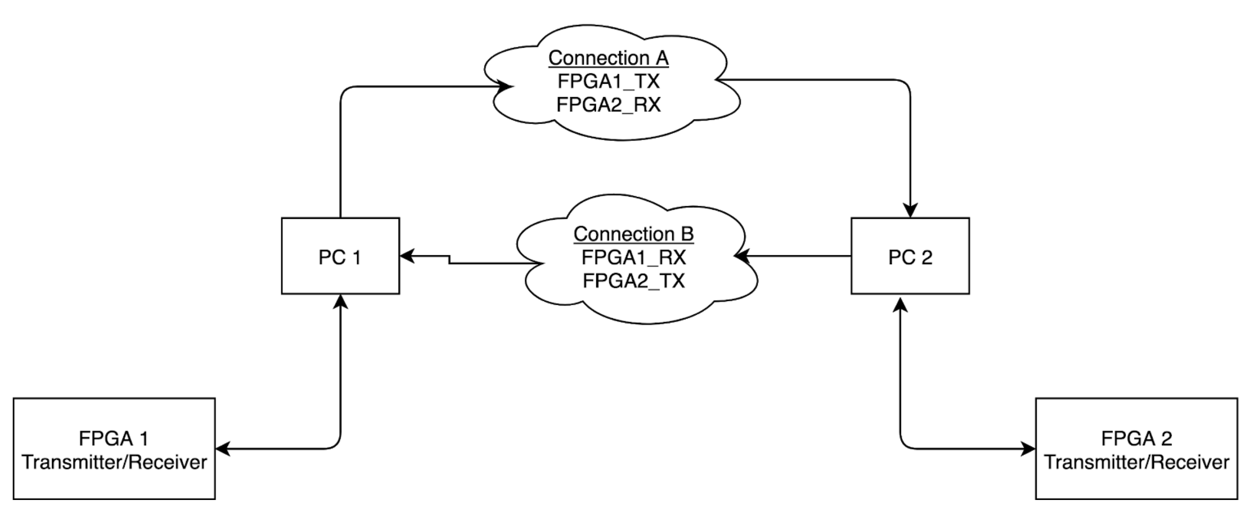
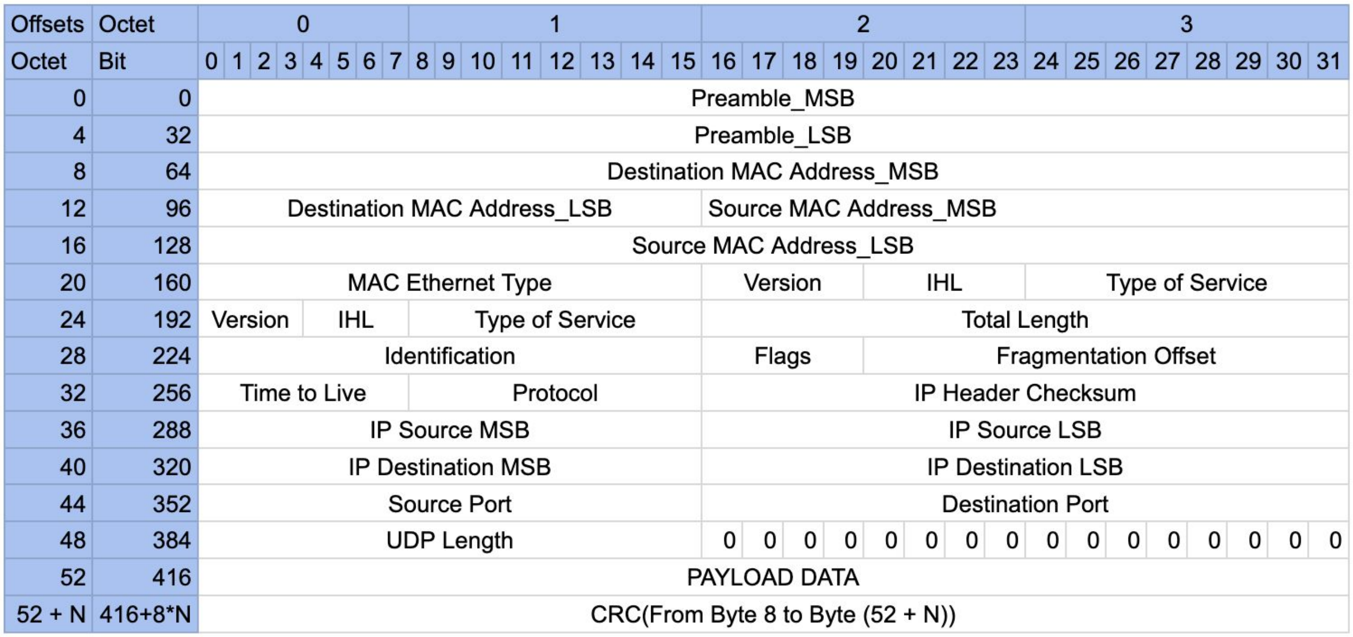


Figure 4: Networking block diagram. Illustrates the connection from the FPGA to the PC then to the server and back down again on the other side.

The networking state machine that we developed for the FPGA is integral in order to get the two FPGAs to communicate with each other. The parts of the network are illustrated above and are FPGA1(Robot FPGA), PC1, Connection A, Connection B, PC2, and FPGA2(Control FPGA). In general, FPGA1 and FPGA2 receive and transmit data to and from PC1 and PC2, respectively, via the Ethernet UDP/IP protocol. Looking at FPGA1, we then have PC1 establishing a connection to the Connection A server and Connection B server via a Python program with a simple state machine. Once a connection is made for Connection A a UDP verification packet is then sent to the FPGA lighting up the 1 LEDs to blue and similarly for Connection B which lights up another LED to blue. The same is done for FPGA2 and PC2 once they establish a connection to the Connection A server and Connection B server. We chose the UDP/IP protocol rather than the more popular TCP/IP protocol because it was simple, connectionless, and faster for high speed streaming such as video streaming. UDP/IP however has a drawback for its simplicity which is that sometimes packets get dropped and can even arrive at the destination out of order from when you sent it. This isn't a problem with streaming data because as long as you don't rely on only a few packets to make all of the decisions, but instead on collections of control or video data, the overall message that you wanted to send should be received.

5 Ethernet Interface(Babu)



Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Preamble_MSB																															
4	32	Preamble_LSB																															
8	64	Destination MAC Address_MSB																															
12	96	Destination MAC Address_LSB																Source MAC Address_MSB															
16	128	Source MAC Address_LSB																															
20	160	MAC Ethernet Type																Version				IHL				Type of Service							
24	192	Version				IHL				Type of Service								Total Length															
28	224	Identification																Flags				Fragmentation Offset											
32	256	Time to Live								Protocol								IP Header Checksum															
36	288	IP Source MSB																IP Source LSB															
40	320	IP Destination MSB																IP Destination LSB															
44	352	Source Port																Destination Port															
48	384	UDP Length																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
52	416	PAYLOAD DATA																															
52 + N	416+8*N	CRC(From Byte 8 to Byte (52 + N))																															

Figure 5: Ethernet UDP/IP packet with a IPv4 header and UDP header.(technically the packet begins after the preamble, bit 64, which indicates the start of the packet).

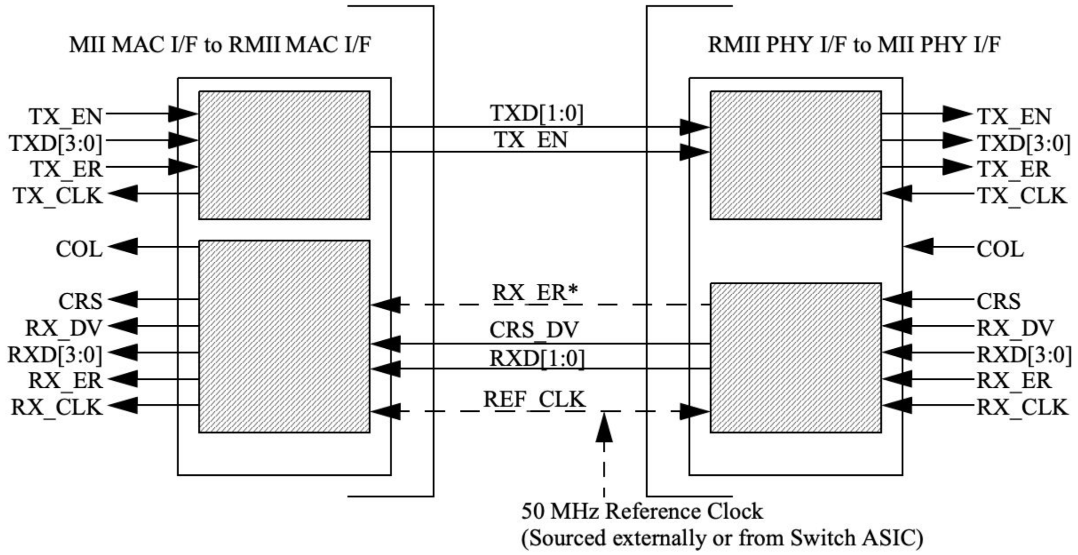


Figure 6: External ethernet transmission and reception interface. The pins primarily used for the project are the 2 bit transmit TXD, 2 bit RXD, the REF_CLK(50Mhz), the TX_EN, and the CRC_DV(data valid).

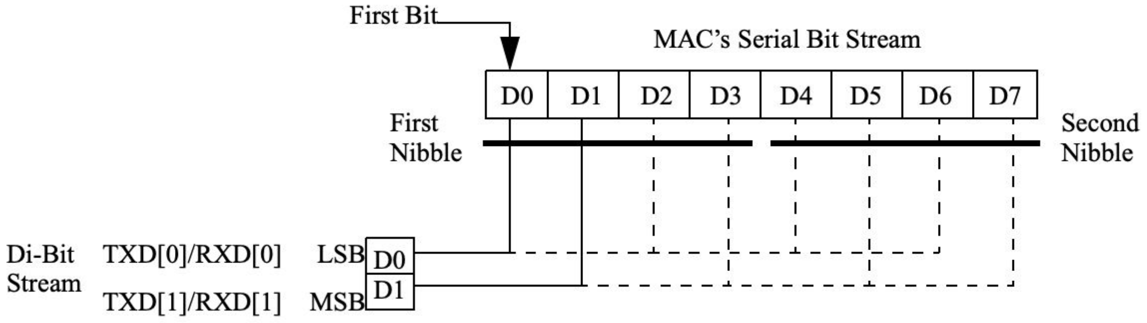


FIGURE 5. Bit Ordering

Figure 7: Bit ordering for transmission and reception of the Ethernet bits. You must swap the nibbles as well as the semi-nibbles(or dibits). This means that if the received byte is ab_cd_ef_gh(where received using big endian) then the actual data byte is gh_ef_cd_ab. Similarly if you want to send the data byte ab_cd_ef_gh(send using big endian) you must transmit gh_ef_cd_ab.

5.1 Ethernet Reception

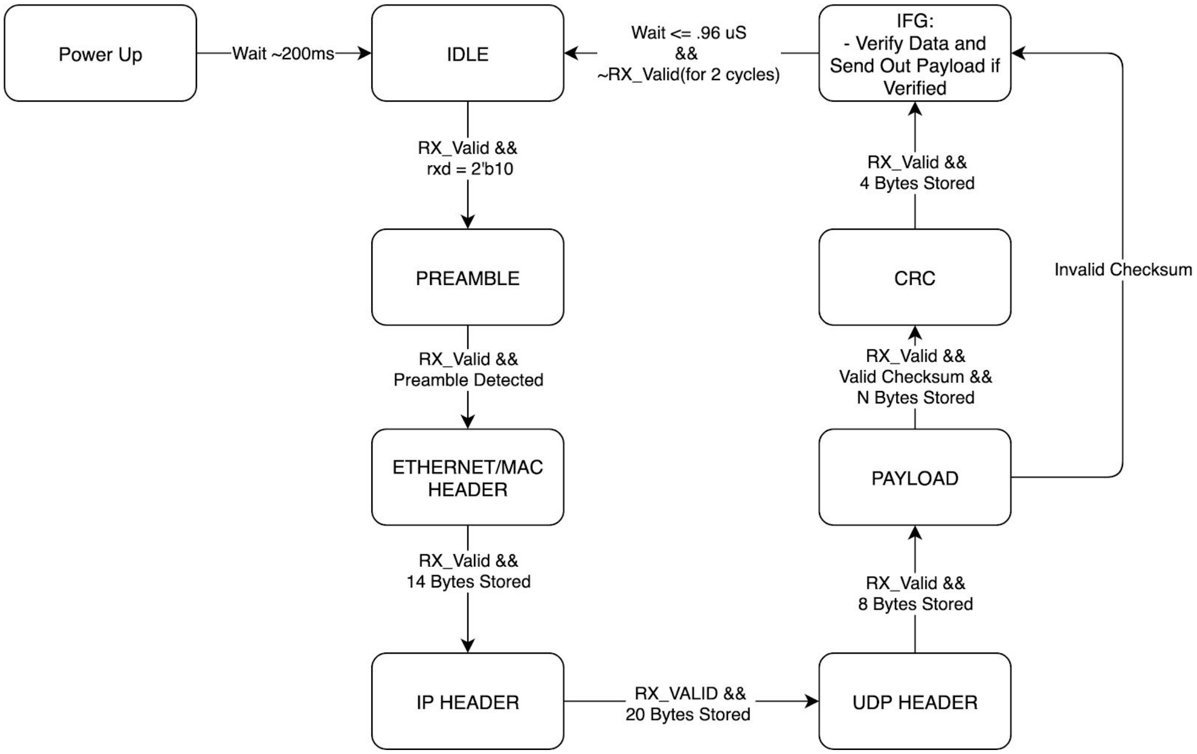


Figure 8: Finite State Machine for the UDP/IP packet reception module

The Ethernet Reception module is how high speed sensor and control data is received by the FPGA from the PC. This data comes into the module 2 bits at a time. How this module functions, at a high level, is to decode the UDP Ethernet Packet as well as verify that the information is valid. This verification is done through an IP address check, Port number check, IP header checksum check, and lastly a Cyclic Redundancy Check for any errors during transmission. Two important things to note are that overall the data is sent to you from the PC in big endian in bytes and bits. The only exception is the CRC which is sent in little endian in bytes and big endian in bits. Additionally, because of the external interface of the Ethernet PHY(physical layer) you must swap the nibbles and the di-bits in a byte when you receive the information in order to get the actual data.

Next I'll do an overview of the state machine. To begin, the first state is Power Up. As the name suggests, this state has a parameter that allows for a proper power up time. After power up, the reception module goes to IDLE which means that it will wait for a signal of 2'b01 which indicates the start of the preamble. The next stage is the preamble, so when the rolling received information is a 64 bit hexadecimal of 64'h55_55_55_55_55_55_55_D5(remember that it's received big endian and that to swap the nibbles and semi-nibble(or di-bits) in every received byte) then you go to the next stage. The next stages are for the MAC header, the IP Header, and the UDP header. In all of these stages we know their appropriate sizes so we can transition states when appropriate. Additionally the headers give us the length of the payload for the next stage. One note is that if the header checksum is wrong(corrupted) we want to skip the payload and go to the end stage cause it could have produced a wrong length and stay in the payload stage for a long time. Another thing is that we must be calculating the CRC from the MAC header all the way up to the end of the payload in order to verify the packet at the end. So, once the length of the payload has been transversed and the payload is stored, stop calculating the CRC and go to the reading CRC stage. Once done, we go to the end stage, or Inter-Frame Gap as it is called, where we can stay for at most 48 clock cycles until we need to go back to IDLE and wait for another packet. In the end stage, we can use the CRC calculated and the CRC received in order to check if the packet is valid, additionally we can also check on the IP header checksums(the UDP checksum is optional so we don't check it) and IP addresses and ports.

5.2 Ethernet Transmission

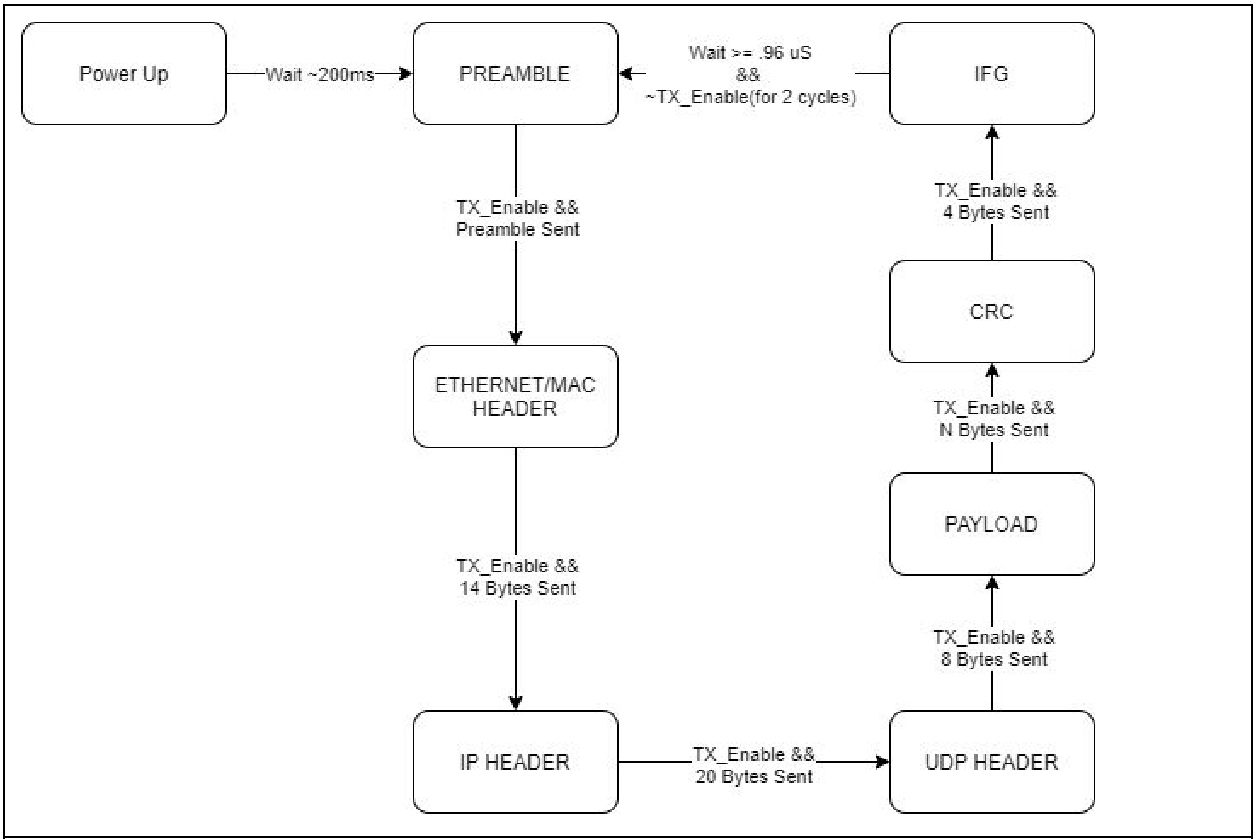


Figure 9: Finite State Machine for the Ethernet UDP/IP transmission module

The Ethernet Transmission module works in a lot of the same ways as the reception module, but instead this time it's sending data instead of taking data in. Transmission is also sent to the destination 2 bits at a time. At a high level it works by encapsulating variable payload data of a fixed length within multiple relatively constant headers. In this implementation no header or payload length changes, but the only data that changes after uploading to the FPGA is the payload information, the destination port, and the CRC. Right now the Payload size is the maximum of 548 bytes in order to fit in as much camera and sensor data as possible but it can go to as low as 18 bytes. To reiterate, two important things to note are that overall the data you are sending is in big endian in bytes and bits. Except for sending the CRC which is little endian in bytes and big endian in bits. Additionally, because of the external interface of the Ethernet PHY(physical layer) you must swap the nibbles and the di-bits in a byte when you transmit the information in order to the PC to receive the original intended information.

Next, I'll do an overview of the state machine. Similar to the reception, the first stage is the preamble. Once you do the appropriate swapping of nibbles and di-bits, send the 64 bit hexadecimal of 64'h55_55_55_55_55_55_55_D5, which indicates a start of a packet. After this you send the MAC Header, IP Header, and UDP Header, then you transition to the payload. Once you send your payload, making sure that you're calculating the CRC using everything but the preamble, then you transition to the CRC stage. Here is the state where you send the calculated CRC making sure that you send it little endian in bytes. Once finished, the module is transitioned to the end state or Inter-Frame Gap where the module waits for at least 48 clock cycles, equivalent to 12 sent bytes, until it can send another packet. Note that some computers have flow control that drops packets that come in too fast. One solution that I found is to increase the amount of cycles in the Inter-Frame Gap in order to space out the packets. On my Windows PC computer, I found that 250 bytes or 1000 clock cycles in-between packets worked well, and could still transmit data fast enough to stream video data.

6 Motor Control(Babu)

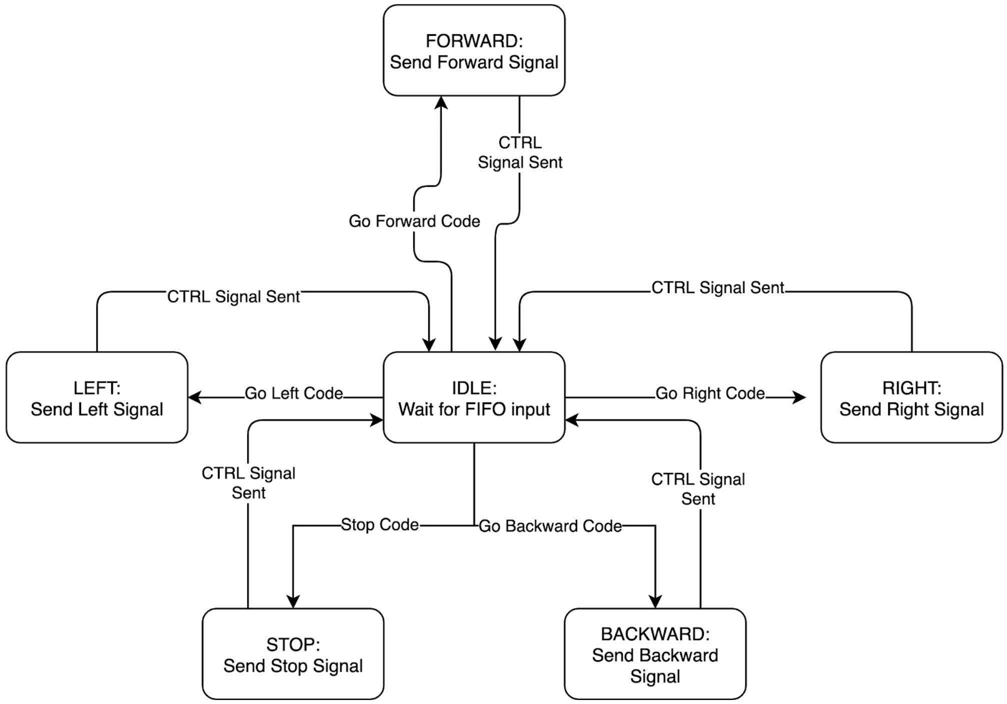


Figure 10: Finite State Machine for the Motor Control module

A_I(in)	B_I(in)	A_O(out)	B_O(out)	motion
0	0	0	0	off
1	0	1	0	forward/backward
0	1	0	1	backward/forward
1	1	1	1	off

Table 1: 2 input control and 2 output control for one motor. The motor direction of forward or backward depends on the wiring.

The motor controller module is used to convert the commands STOP = 0, FORWARD = 1, BACKWARD = 2, RIGHT = 3, LEFT = 4 into the robot's motor movements. STOP indicates that both motors are not moving. FORWARD indicates that both motors are moving forward. BACKWARD indicates that both motors are moving backward. RIGHT means that the left motor is moving forward and the right motor is not moving. LEFT means that the right motor is moving forward and the left motor is not moving.

After making sure that the motor driver is connected to the portable battery via a DC jack, then all that is needed in order to send input signals A_I and B_I to each individual motor are 4 PMOD pins from the FPGA(2 for each motor). Then make sure that you wire the motors to the output A_O and B_O corresponding to each input. The module then just acts as a simple case statement that takes the STOP, FORWARD, BACKWARD, RIGHT, and LEFT commands and sets the inputs A_I and B_I for both motors.

7 Distance Sensor(Babu)

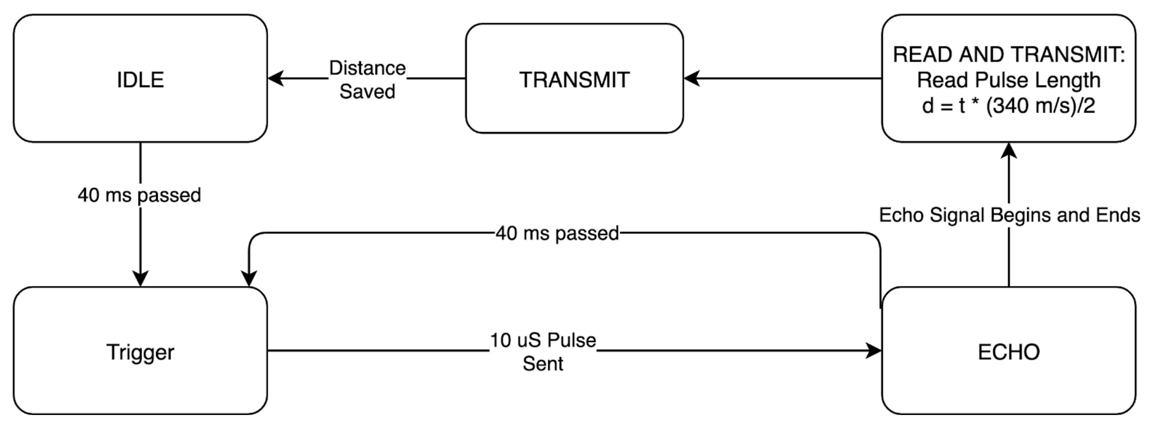

IDLE
Distance Saved
TRANSMIT
READ AND TRANSMIT: Read Pulse Length d = t * (340 m/s)/2
40 ms passed
Echo Signal Begins and Ends
40 ms passed
Trigger
10 uS Pulse Sent
ECHO

Figure 11: Overview of the FInite State Machine for the distance sensor module. Note that the distance equation above is for conversion to meters.

The HC-SR04 distance sensor inputs the echo pin(along with 5V and GND) and outputs the trigger pin in order to calculate the distance of an object with a range of 2cm - 400cm.

Above, I created a rough finite state diagram of how the distance sensor is supposed to work as a finite state machine. To begin, the module first starts out at the IDLE stage. After 40ms it sends out a trigger signal to the distance sensor. After 10us you disable the trigger signal. You then wait in the echo stage and if you see an echo signal, count how many microseconds until the signal ends. If you don’t see a signal for 40 ms then go back to the trigger. If you do see a signal convert it to either cm or inches, then transmit/save it then go back to idle.

8 Camera Sensor(Babu)

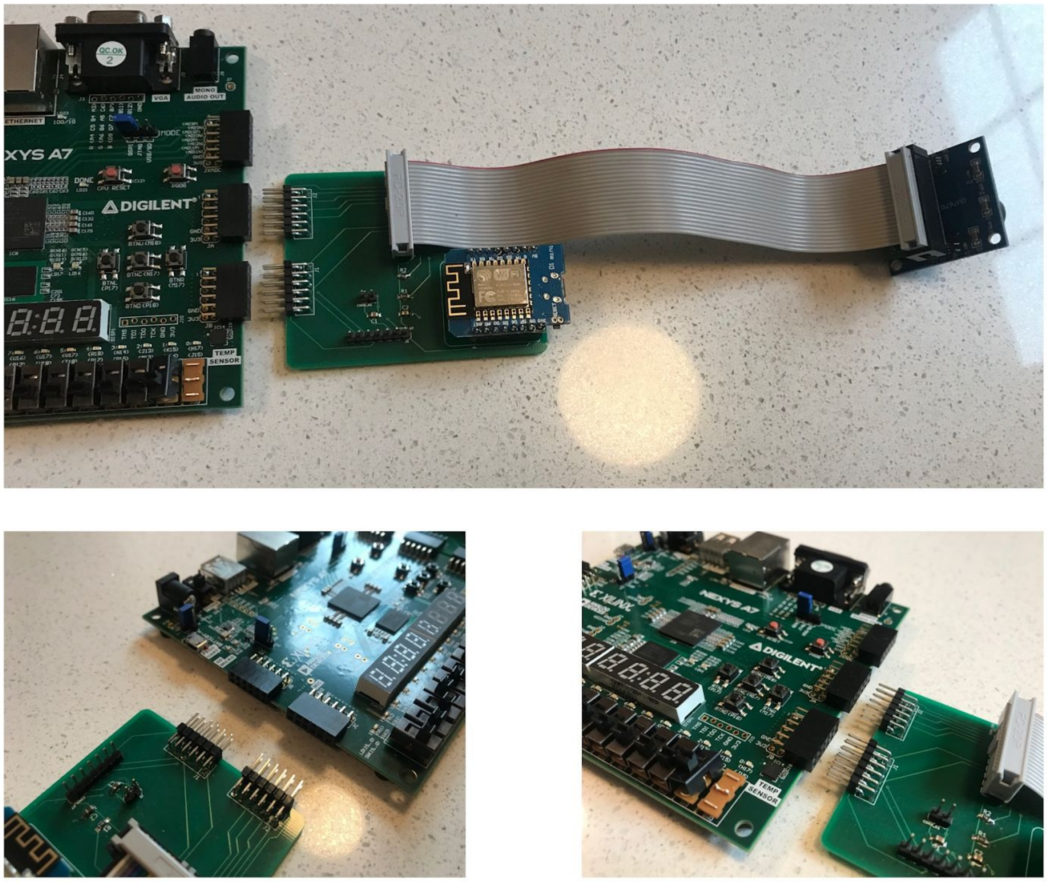


Figure 12: Camera OV7670 sensor connection to FPGA. In addition to the camera sensor we also had an ESP8266 to configure the FPGA.

The OV7670 camera sensor provided the video data that the robot transmitted to the controller for the display. We had an ESP8266 microcontroller that used an I2C in order to configure the OV7670 so all the FPGA had to do was process the camera data.

To begin, using the camera_read module we were able to acquire the 16 bit pixel RGB565 pixel data by using VSYNC_in and HREF_in pins in order to do row capture. Once we could read frames pixel by pixel we then put the data into an asynchronous input and output bram. This allowed us to save pixel data in one OV7670 pixel clock speed and output data into the Ethernet Transmission 50 MHZ clock speed. Additionally, the 16 bit RBG565 was converted to RGB444 by taking the highest 4 bits from the red, blue, and green bits. From here data could be read at 50 MHZ and transmitted through the Ethernet to be displayed on the controller side.

9 Controller Top Level - (Brandon)

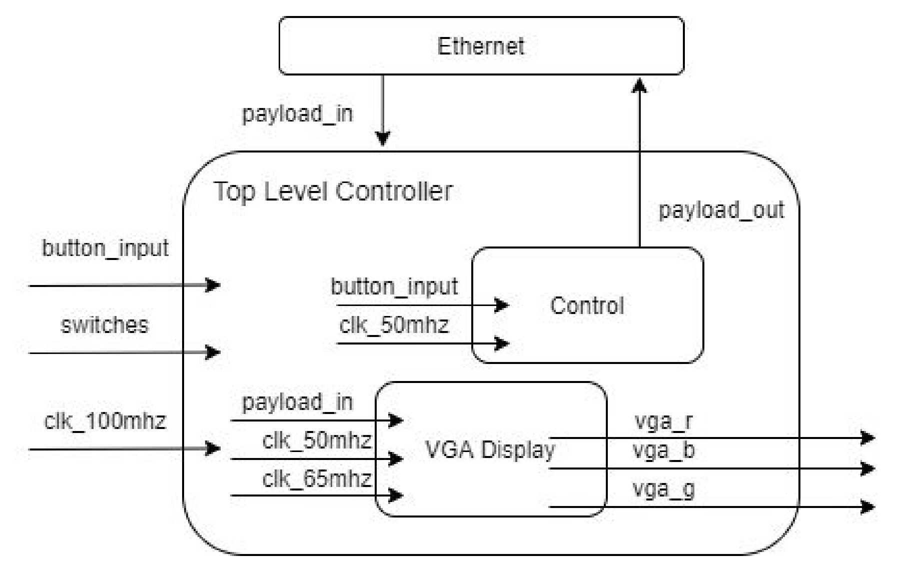


Figure 13: Controller Top Level Block Diagram

The side operating the controller must perform two tasks. One is to prepare commands for the robot from the button inputs to use as a transmitted payload. The other is to interpret the received packets that contain the camera and sensor data in order to display them both on a VGA monitor. For consistency, we will read and write the payloads according to a 50mhz clock (created by the clock wizard IP). The VGA display module will need an additional 65mhz pixel clock. The switches will allow for different settings of the display (camera on/off and 2x resolution).

Controller Interface

The controller itself was straightforward to implement. Our idea was to use the buttons on the FPGA as a sort of d-pad (directional pad) for the robot. Pressing btnu would make the car go forward, pressing btnl would make the car go left, etc. We reserved btnc for resetting the ethernet connection. We then made sure to debounce btnl, btnu, btnr, and btnd at the system clock rate (100 mhz). Whenever any of these buttons was pressed the corresponding value was passed to the transmitted payload register. These were determined by the following mapping.

forward	1
backwards	2
right	3
left	4
stop	0

Table 2: Command mapping

When no buttons were being pressed, the payload would simply be 0 and force the robot to stop. There is a delay when sending any of these commands, so during testing we made sure to only press the buttons for short periods of time or in pulses so as to avoid having the robot move further than expected and possibly crash.

VGA Overview

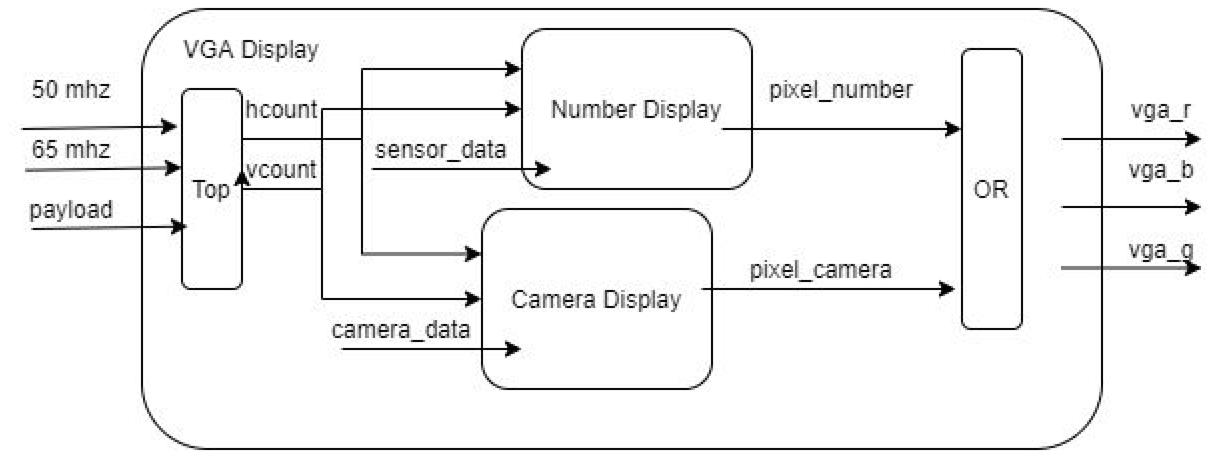


Figure 14: VGA Overview Block Diagram

In essence, the vga module takes in the 548 byte payload and writes the proper rgb pixel values to the vga pins in order to display both the video from the camera feed and the numbers that represent the reading on the sensor. Much of the raster logic comes from lab 3, where hcount and vcount represent the current position of the pixel on the screen that is being overwritten. This is done with a pixel clock of 65mhz. However, it is important to note that the payload is being updated at the top level clock, 50 mhz. The first 34 bits contain the sensor data, which will be used as the input for the number display logic. The remaining bits contain the RAM addresses (17 bits), and rgb values (12 bits) for 150 pixels. We can only write one pixel value to the RAM at a time, so we used a counter to be able to iterate over all of the 150 pixel representations in the payload. From there, we simultaneously read and write to a RAM for our video image one pixel at a time.

10 VGA Display - Numbers(Brandon)

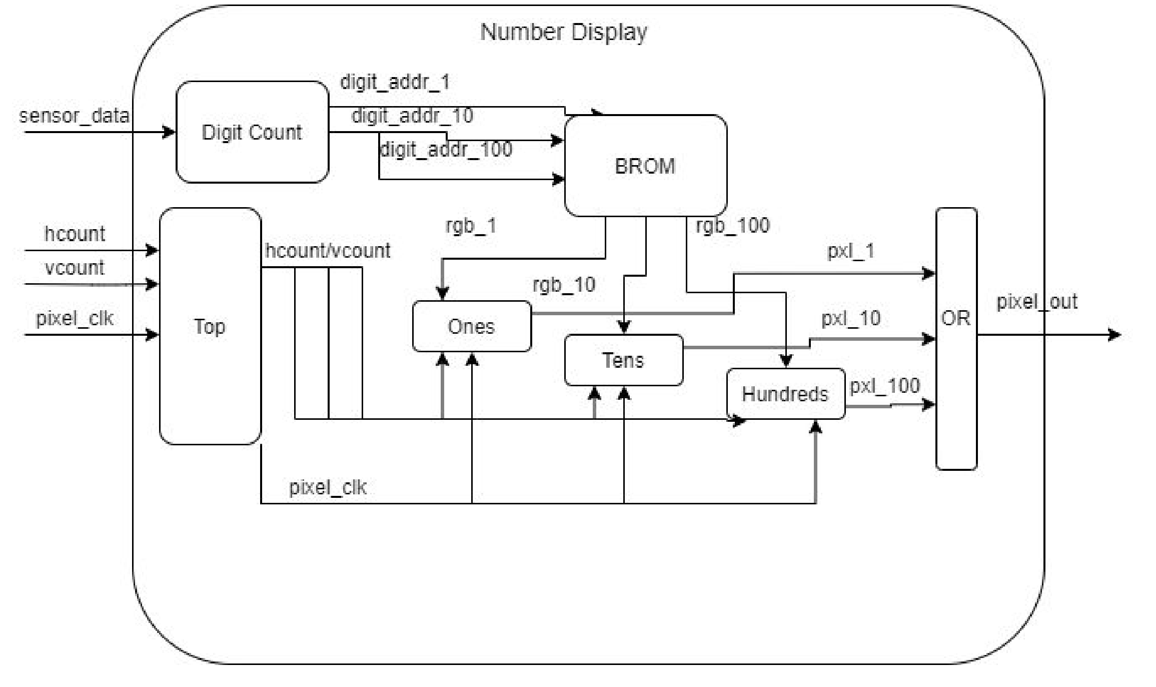


Figure 15: VGA number display block diagram

In order to display the numbers on screen that represented the readings from the sensor, we had to perform the following. First, we generated a single port BROM from the 48x48 provided COE file for integers 0-9. The BROM would be 4 bits wide and 23040 bits deep since each of the COE entries was 4 bits, each number was 48*48 pixels, and there were 10 integers total. From our estimations from experimenting with the sensor, the readings never appeared to go over 400, so we decided that three digits would be sufficient for the representation.

We decided to place the hundreds digit at coordinates 700, 50. This would allow enough room for the 640 pixel wide image for the video on the left. We then shifted the other digits by 50 as each was 48 pixels wide, so their final positions were ((700, 50), (750, 50), (800, 5)).

Each of the digit instantations required the following basic inputs. The row and column position of the digit, the horizontal & vertical screen count, and the corresponding value for the digit. The address of the current corresponding pixel could be found by the following calculation (2304*digit_in) + (hcount_in-x_in) + ((vcount_in-y_in)*WIDTH). The data output by the BROM at this address is a 4 bit value for a grayscale pixel.

Determining the values for each of these digits however is not as straightforward as reading the value directly from the UDP packet. Instead, we have to utilize a counter to determine the current value for the ones place, the tens place, and the hundreds place. Using three separate registers for each of the digits, we increment the values of these registers in the following manner. Every cycle we increment the ones place until it hits 9, then reset it to zero. Whenever the one place value is 9, then the tens place value will increment by 1 unless its value is 9, in which case it will be set to 0. Then, whenever the tens place value is 9, the hundreds place will increment by 1 unless its value is 9, in which case it will be set to 0. All of these registers will continue to increment in this fashion until the counter reaches the given sensor data value or a predetermined refresh threshold (1000 in our case since the maximum sensor reading is 400) is reached.

In order to test this module, we first used the switches in lieu of the data packets to generate a value to be passed to the number display module. To start off, we used switches 0-2 for low value integers, but switches 0-9 could be used to test numbers in the hundreds range since log2(1000)~10. In addition to this testing method, we used a counter that incremented every second (using a divisor parameter of 25 million for a 50mhz clock). We found the latter to be more effective in demonstrating a larger range of numbers for testing.

11 VGA Display - Camera(Brandon)

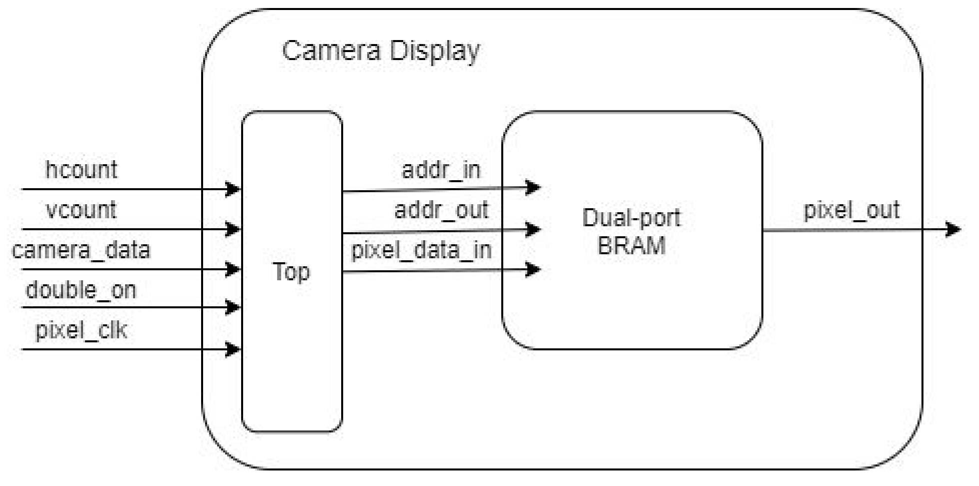


Figure 16: VGA Camera display block diagram

Instead of a single-port BROM that was used for the number display, we required a dual-port BRAM for the camera display. The BRAM had a 12 bit width, and 76800 bit depth. 76800 comes from the 320*240 resolution of the input data. Then there is a 12 bit RGB value for each pixel. The reason that we use a *dual-port* BRAM is so that we can both read and write simultaneously. We write new values from the incoming UDP packet, which contains the corresponding address to the BRAM, and then we read values from the appropriate address according to hcount and vcount. It is important to note that these are performed at different clock speeds. Since all of the data is being transmitted and received at 50mhz, for consistency we decided to also write to the BRAM at 50mhz. However, since the VGA display is running at a 65mhz clock, we are reading the values from the BRAM at 65mhz. Also note to make sure that write-enable is always high.

Also, when assigning the current pixel value, it is important to zero out all other pixels outside of the 320x240 boundaries, otherwise the last pixel value at say for instance position (320,1) will be carried over to position (321,1) and so on.

In addition to the 320x240 resolution, we decided to implement a 640x480 resolution option. We determined that a nearest neighbor interpolation scheme would suffice for our purposes. So, for each pixel in the 320x240 representation, 4 pixels would take on that original pixel's value for the 640x480 representation. I believe that this is most easily seen with the following tables. The upper table is like a 2x2 image. Its values are then interpolated to a 4x4 image.

R	G
B	R

Table 3: original 2x2 image

R	R	G	G
R	R	G	G
B	B	R	R
B	B	R	R

Table 4: Interpolated 4x4 image

This can be done by first assigning a switch to toggle between the two resolutions. Then the image boundaries must be changed to 640x480. Then, rather than creating another 640x480 BRAM, we can just use the same values from the original 320x240 BRAM and use the following algorithm to determine the appropriate pixel address.

1. If hcount is even, divide by 2 for a new hcount. If hcount is odd, subtract 1 and divide by 2 for a new hcount.
2. If vcount is even, divide by 2 for a new vcount. If vcount is odd subtract 1 and divide by 2 for a new vcount.

These "new" hcounts and vcounts will produce the same address as the 320x240 representation for the same pixel value when performing the following calculation, addr = 320*(vcount_in) + hcount_in. In the actual code, the altered address is calculated in one line instead of using "new" hcount and vcount registers which would introduce a one-cycle delay.

A possible discrepancy that we should address is that the above calculation had to be changed because of the position of the camera. It was easier to mount the camera on its side rather than upright, so we swapped hcount and vcount in order to rotate the resulting image 90 degrees. So, the final line that appears in the code is addr = 320*(hcount_in) + vcount_in.

12 Development Process

Overall throughout the whole development process one of the most important things that we used is Version Control with Git. This allowed us to experiment more and to also go back whenever we made a mistake. Additionally it also allowed us to check each other's work if something was wrong. One thing to note however is that merge conflicts may arise if each partner pushes their implementation files. To avoid these, we had to frequently delete those files and pull again.

Since both of us we were apart during the entire development process, communicating chat messages and Facetime also allowed us to keep up to date with each other's work. Next we'll go through some of the development process for the Robot and the Controller.

12.1 Robot

The most difficult part of the Robot development process was developing the Ethernet Transmission and Reception modules. Due to the fact that the documentation is unclear, I had to do a lot of trial and error, multiple simulations, and also consult multiple sources. Some of the most significant problems was to remember that even though you mostly receive and transmit in big endian in byte and bits you must also swap the nibbles and the semi-nibbles(or dibits) before you transmit or before you save the data. Additionally, the CRC is also received and transmitted in little endian in bytes but big endian in bits(you must still also swap nibbles and semi-nibbles).

Aside from the ethernet development, the Motor Controller module was relatively straightforward as it was basically controlled by digital inputs and could be represented by a truth table. The distance sensor was a little bit more difficult because we had both a trigger and an echo signal and you didn't want to send a trigger when the echo was arriving. This however wasn't too much of a problem because the documentation was comprehensive enough to understand what was going on and to fix any bugs. Additionally, code had been written in Arduino C++ to program the HC-SR04 module, so studying that wasn't too difficult. The other module was the camera sensor. Luckily, the microcontroller handled the configuration for the OV7670 so we didn't have to worry about developing a I2C module. However, we still had to read from the camera sensor as well as save it onto a bram module, which took a little bit more time. Next, was the integration of all of the modules and the creation of a state machine to handle a connection to the PC and the server. This took a little bit of time because any bugs that might have been hidden in any of the modules showed up. For example I had to make sure not to send a UDP packet when the other packet was still being transmitted. This required that I create a tx_busy signal. However, due to the timing specification of the UDP protocol this was more difficult that I anticipated because I needed to make it combinational instead of sequential which took a little bit more thought and debugging. Lastly, was the server and PC python files. This part wasn't too difficult because I could test it independently from the FPGA and it didn't need to be compiled. The most difficult part of this was to think about what happens if packets are dropped, because this could lead to an infinite loop. The solution to this was to resend some essential messages if needed.

12.2 Controller

The button input commands were easy enough to test on my side and was really only a matter of integration with the overall top level module. We actually tested the robot commands towards the end of the development process because of our confidence that it would work smoothly. Looking back, I think we made the right choice by focusing on other modules when we were working together on zoom calls.

Having completed lab 3 definitely helped with VGA development, but I did have to go back through and refresh myself on the integration of the xvga module and how hsync, vsync, hcount, vcount, vga_r, etc were all interconnected as I encountered bugs. From there the problem became how do we correctly interpret the incoming binarized sensor data for a base 10 display and calculate the appropriate ROM addresses. I found a viable solution involving counters and was able to get the numbers module working fairly quickly.

The camera display module was probably the module that I had the most concerns about as it was difficult to test unless we had the server running and the camera on the robot side was on. I had no camera on my side, so the only times that I could evaluate the integrity of the module was when we both had set apart time to do so. Luckily, we still managed to test and debug in a timely manner.

13 Ideas/Conclusion

Given more time, we would have liked to do more with the video data. There were some artifacts and even though the quality of the OV7670 is not great in the first place, there are definitely some processing techniques that could have yielded a more impressive picture. We talked about trying to implement a low pass filter, which would smooth some of the jaggedness. Another smoothing technique we could have implemented is to convolve the pixel values with a Gaussian kernel before displaying. One problem that was visible in the demo was a vertical choppiness artifact. This would make if there seemed to be a lag in the horizontal bars as an expected artifact of the rastering pattern. It turns out that the vertical bar artifact actually comes from the horizontal bar artifact since we switched hcount and vcount to compensate for the camera's position. We discussed that one way to fix this problem all-together would be to create two RAMs instead of one. One would continue to update the pixel values for the entire image at the 50 mhz clock rate, but the other would serve as a buffer that would only refresh at a designated framerate. This way, even though the framerate might drop, the video would seem smoother.

For fun, it would be interesting to use a different controller interface like a PS/2 keyboard or even the on-board accelerometer. The robot only takes one command at a time, so it would also be a nice challenge to find a data format and motor scheme that would allow for more flexible steering, making the robot go left *and* forward instead of just left for instance.

In the end, we were happy that we were not only able to establish a connection, but also interact with the other partner's hardware in such tangible ways despite being halfway across the country. Though from the beginning we believed that we could control the robot one way or another, we were not very confident that we could adequately stream video with our setup. However, after hours of debugging and zoom calls we finally were able to see Babu's face on the monitor. This was a great lesson in many areas including ethernet communication, servers and display logic.

14 Verilog Source Code

// camera_read.sv

```
module camera_read(
   input  p_clock_in,
   input  vsync_in,
   input  href_in,
   input  [7:0] p_data_in,
   output logic [15:0] pixel_data_out,
   output logic pixel_valid_out,
   output logic frame_done_out
   );


   logic [1:0] FSM_state = 0;
   logic pixel_half = 0;

   localparam WAIT_FRAME_START = 0;
   localparam ROW_CAPTURE = 1;


   always_ff@(posedge p_clock_in)
   begin
   case(FSM_state)

   WAIT_FRAME_START: begin //wait for VSYNC
      FSM_state <= (!vsync_in) ? ROW_CAPTURE : WAIT_FRAME_START;
      frame_done_out <= 0;
      pixel_half <= 0;
   end

   ROW_CAPTURE: begin
      FSM_state <= vsync_in ? WAIT_FRAME_START : ROW_CAPTURE;
      frame_done_out <= vsync_in ? 1 : 0;
      pixel_valid_out <= (href_in && pixel_half) ? 1 : 0;
      if (href_in) begin
          pixel_half <= ~ pixel_half;
          if (pixel_half) pixel_data_out[7:0] <= p_data_in;
          else pixel_data_out[15:8] <= p_data_in;
      end
   end
   endcase
   end
endmodule
```

// display_8hex.sv

```
module display_8hex(
   input clk_in,                 // system clock
   input [31:0] data_in,         // 8 hex numbers, msb first
   output logic [6:0] seg_out,     // seven segment display output
   output logic [7:0] strobe_out   // digit strobe
   );

   localparam bits = 13;

   logic [bits:0] counter = 0;  // clear on power up

   logic [6:0] segments[15:0]; // 16 7 bit memories
   assign segments[0]  = 7'b100_0000;  // inverted logic
   assign segments[1]  = 7'b111_1001;  // gfedcba
   assign segments[2]  = 7'b010_0100;
   assign segments[3]  = 7'b011_0000;
   assign segments[4]  = 7'b001_1001;
   assign segments[5]  = 7'b001_0010;
   assign segments[6]  = 7'b000_0010;
   assign segments[7]  = 7'b111_1000;
   assign segments[8]  = 7'b000_0000;
   assign segments[9]  = 7'b001_1000;
   assign segments[10] = 7'b000_1000;
   assign segments[11] = 7'b000_0011;
   assign segments[12] = 7'b010_0111;
   assign segments[13] = 7'b010_0001;
   assign segments[14] = 7'b000_0110;
   assign segments[15] = 7'b000_1110;
```

```
always_ff @(posedge clk_in) begin
  // Here I am using a counter and select 3 bits which provides
  // a reasonable refresh rate starting the left most digit
  // and moving left.
  counter <= counter + 1;
  case (counter[bits:bits-2])
      3'b000: begin  // use the MSB 4 bits
              seg_out <= segments[data_in[31:28]];
              strobe_out <= 8'b0111_1111 ;
             end

      3'b001: begin
              seg_out <= segments[data_in[27:24]];
              strobe_out <= 8'b1011_1111 ;
             end

      3'b010: begin
               seg_out <= segments[data_in[23:20]];
               strobe_out <= 8'b1101_1111 ;
              end
      3'b011: begin
              seg_out <= segments[data_in[19:16]];
              strobe_out <= 8'b1110_1111;
             end
      3'b100: begin
              seg_out <= segments[data_in[15:12]];
              strobe_out <= 8'b1111_0111;
             end

      3'b101: begin
              seg_out <= segments[data_in[11:8]];
              strobe_out <= 8'b1111_1011;
             end

      3'b110: begin
               seg_out <= segments[data_in[7:4]];
               strobe_out <= 8'b1111_1101;
              end
```

```
         3'b111: begin
                 seg_out <= segments[data_in[3:0]];
                 strobe_out <= 8'b1111_1110;
                end

      endcase
     end
endmodule
```

// distance_sensor.sv

```
// The default is in inches else it's in centimeters
module distance_sensor#(parameter INCHES = 1)(
       input clk_50mhz,
       input echo,
       input button_reset,
       output logic trigger,
       output logic [9:0] distance
   );

   logic [31:0] us_counter;
   logic temp_trigger;

   logic [5:0] one_us_counter;

   logic [8:0] ten_us_counter;

   logic [20:0] fourty_ms_counter;

   assign trigger = temp_trigger;

   always_ff @(posedge clk_50mhz) begin
       if(button_reset)begin
           one_us_counter <= 0;
           ten_us_counter <= 0;
           fourty_ms_counter <= 0;
           temp_trigger <= 0;
           distance <= 0;
       end else begin
           one_us_counter <= ((one_us_counter == 0) ? 6'd50 : one_us_counter ) - 1;
           ten_us_counter <= ((ten_us_counter == 0) ? 9'd500: ten_us_counter) - 1;
           fourty_ms_counter <= ((fourty_ms_counter == 0) ? 21'd2_000_000: fourty_ms_counter) - 1;

           if((ten_us_counter == 0) && temp_trigger) temp_trigger <= 0;

           if((one_us_counter == 0)) begin
               if(echo)begin
                   us_counter <= us_counter + 1;
               end else if (us_counter) begin
                   distance <= (us_counter /  (INCHES ? 148 : 58));
                   us_counter <= 0;
               end
           end

           if((fourty_ms_counter == 0)) begin
               temp_trigger <= 1;
           end
       end
   end
endmodule
```

// motor_control.sv

```
// Note forward, and backward is based on motor orientation and wiring
//IA    IB  |  OA   OB
//L     L      L    L (OFF)
//H     L      H    L (FORWARD)
//L     H      L    H (BACKWARD)
//H     H      H    H (OFF)

module motor_control(
       input logic [2:0] control,// input control
       output logic [3:0] motor// {AIA,AIB,BIA,BIB}
   );

   // Control States in GrayCode
   localparam  STOP     = 3'b000;
   localparam  FORWARD  = 3'b001;
   localparam  BACKWARD = 3'b010;
   localparam  RIGHT    = 3'b011;
   localparam  LEFT     = 3'b100;

   always_comb begin
       // Assume motor A is on the left and motor B is on the right
       case(control)
           STOP:     motor = {1'b0,1'b0,1'b0,1'b0};
           FORWARD:  motor = {1'b1,1'b0,1'b1,1'b0};
           BACKWARD: motor = {1'b0,1'b1,1'b0,1'b1};
           RIGHT:    motor = {1'b0,1'b0,1'b1,1'b0};
           LEFT:     motor = {1'b1,1'b0,1'b0,1'b0};
           default:  motor = {1'b0,1'b0,1'b0,1'b0}; // Default signal is STOP
       endcase
   end
endmodule
```

// phy_init.sv

```
module phy_init(
   input                                         clk,
   input    logic                                button_reset,
   inout    logic                                eth_crsdv,
   inout    logic                                [1:0] eth_rxd,
   output   logic                                eth_rxerr,
   output   logic                                eth_intn,
   output   logic                                eth_rstn,// Used to reset the PHY
   output   logic                                phy_rst_done
   );
   localparam RESET = 0;
   localparam DONE = 1;

   localparam RESET_BEFORE =  5_000_000;
   localparam RESET_AFTER = 400;

   logic state;
   // Reset
   assign eth_crsdv = (state == RESET) ?  1'b0 : 1'bz;
   assign eth_rxd = (state == RESET)   ?  2'b11: 2'bzz;
   assign eth_rxerr = (state == RESET) ?  1'b0 : 1'bz;
   assign eth_intn = (state == RESET)  ?  1'b1 : 1'bz;
   logic [22:0] counter;

   always_ff @(posedge clk) begin
       if(button_reset) begin
           state <= RESET;
           counter <= 0;
           eth_rstn <= 0;
           phy_rst_done <=0;
       end else begin
           case(state)
               RESET:  begin
                   if (counter == RESET_BEFORE - 1) eth_rstn <= 1;
                   if (counter == (RESET_BEFORE + RESET_AFTER - 1)) begin
                       state <= DONE;
                   end
               end
               DONE:   begin
                   phy_rst_done <= 1;
               end
               default: state <= DONE;
           endcase

           if (state == RESET) counter <= counter + 1;
       end
   end
endmodule
```

// top_level_controller.sv

```
module top_level_controller(
       input clk_100mhz,
       // Ethernet Pins
       input logic eth_mdio,
       inout logic [1:0] eth_rxd,
       inout logic eth_crsdv,// valid receive

       output logic eth_mdc,
       output logic eth_rstn,
       output logic eth_txen,
       output logic [1:0] eth_txd,
       output logic eth_refclk,
       output logic eth_intn,
       output logic eth_rxerr,

       // LEDs
       output logic led16_b,
       output logic led16_r,
       output logic led17_b,
       output logic led17_r,

       output logic [15:0] led,

       // Buttons
       input btnc,btnr,btnd,btnl,btnu,

               //VGA display
          output logic[3:0] vga_r,
          output logic[3:0] vga_b,
          output logic[3:0] vga_g,
          output logic vga_hs,
          output logic vga_vs,

       // Hex display
       output logic ca,cb,cc,cd,ce,cf,cg,
       output logic [7:0] an,

       // switches
       input [15:0] sw,

       // HC-SR04 Pins
       output logic trigger, // ja[0]
       input logic echo    // ja[1]
   );

   // Clock variables
   logic clk_50mhz;

   // 50 and 65 (pixel) mhz clock instance
```

```
   clk_wiz_2 clk50_65divider(.clk_in1(clk_100mhz),.clk_out1(clk_50mhz), .clk_out2(clk_65mhz));

   // Seven segment display
   logic [6:0] segments;
   logic [31:0] display_data;      //  Input data for the display
   assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];

   // Clock assignment
   assign eth_refclk = clk_50mhz;

   // HC-SR04 Distance sensor Logic
   logic [31:0] distance;

   // Codes
   localparam RX_INIT = "FPGA RX INIT";
   localparam RX_CONN = "PC to FPGA RX CONNECTED";
   localparam TX_INIT = "FPGA TX INIT";
   localparam TX_CONN = "PC to FPGA TX CONNECTED";
   localparam ROBOT_TO_CONTROL = "ROBOT -> CONTROL";
   localparam CONTROL_TO_ROBOT = "CONTROL -> ROBOT";
   // Ports
   localparam RX_FPGA_PORT = 5001;
   localparam TX_FPGA_PORT = 5001;
   localparam RX_PC_PORT = 5003;
   localparam TX_PC_PORT = 1024;


   // hex display instance
   display_8hex hex_display(.clk_in(clk_50mhz),.data_in(display_data), .seg_out(segments),
.strobe_out(an));




   // PHY ETHERNET LOGIC ------------------------
   localparam PAYLOAD_BYTES = 548;

   logic phy_rst_done;
```

```
// Restart and initialize the PHY
phy_init init(
    .clk(clk_50mhz),
    .button_reset(btnc),
    .eth_rxerr(eth_rxerr),
    .eth_intn(eth_intn),
    .eth_crsdv(eth_crsdv),
    .eth_rxd(eth_rxd),
    .eth_rstn(eth_rstn),
    .phy_rst_done(phy_rst_done)
);



logic reset;
logic left;
logic right;
logic forward;
logic backward;



//BUTTON LOGIC
debounce deb_c(.reset_in(reset), .clock_in(clk_100mhz), .noisy_in(btnc),
             .clean_out(reset));

debounce deb_l(.reset_in(reset), .clock_in(clk_100mhz), .noisy_in(btnl),
             .clean_out(left));
debounce deb_r(.reset_in(reset), .clock_in(clk_100mhz), .noisy_in(btnr),
             .clean_out(right));
debounce deb_u(.reset_in(reset), .clock_in(clk_100mhz), .noisy_in(btnu),
             .clean_out(forward));
debounce deb_d(.reset_in(reset), .clock_in(clk_100mhz), .noisy_in(btnd),
             .clean_out(backward));

logic [(PAYLOAD_BYTES << 3) - 1 : 0] payload_out;

 logic [(PAYLOAD_BYTES << 3) - 1 : 0] payload_buffer =0;
 //DRIVER CONTROLS

 //DIRECTION MAP
 // forward = 1
 // backward = 2
 // right = 3
 // left = 4
 // stop = 0
 always_ff @(posedge clk_50mhz) begin
    if (forward) begin
        payload_buffer[3:0] <= 4'd1;
    end

    else if (backward) begin
        payload_buffer[3:0] <= 4'd2;
    end

    else if (right) begin
        payload_buffer[3:0] <= 4'd3;
    end

    else if (left) begin
        payload_buffer[3:0] <= 4'd4;
    end

    else begin
        payload_buffer[3:0] <= 0; //0 means stop
    end
 end




udp_pkt_receive#(.PAYLOAD_BYTES(PAYLOAD_BYTES),
                 .FPGA_IP_1(169),
                 .FPGA_IP_2(254),
                 .FPGA_IP_3(255),
                    .FPGA_IP_4(255),
                    .FPGA_PORT(RX_FPGA_PORT),

                    .PC_IP_1(169),
                    .PC_IP_2(254),
                    .PC_IP_3(63),
                    .PC_IP_4(159),
                    .PC_PORT(RX_PC_PORT)
                       ) rec(
                       .clk(clk_50mhz),
                       .rxd(eth_rxd),
                       .rx_valid(eth_crsdv),
                       .payload_out(payload_out),
                       .button_reset(btnc),
                       .phy_rst_done(phy_rst_done));

    logic tx_busy;
    logic input_valid;
    logic [239:0] payload_in;
    logic [15:0] send_port;
    udp_pkt_send_test#(.PAYLOAD_BYTES(PAYLOAD_BYTES),
                       .IFG_BYTES(250),

                       .FPGA_IP_1(169),
                       .FPGA_IP_2(254),
                       .FPGA_IP_3(255),
                       .FPGA_IP_4(255),
                       .FPGA_PORT(TX_FPGA_PORT),

                       .PC_IP_1(169),
                       .PC_IP_2(254),
                       .PC_IP_3(63),
                       .PC_IP_4(159)

                       ) send_test (
                       .clk(clk_50mhz),
                       .button_reset(btnc),
                       .txen(eth_txen),
                       .txd(eth_txd),
                       .t_payload(payload_in),
                       .input_valid(input_valid),
                       .tx_busy(tx_busy),
                       .send_port(send_port),
                       .phy_rst_done(phy_rst_done)

                       );

      logic [26:0] counter;
      logic [31:0] timer;
      logic [3:0] state;

      localparam START_RX = 0;
      localparam START_TX = 1;

      // Send a message to the Receive and Transmit Ports
      localparam SEND_RX = 2;
      localparam SEND_TX = 3;

      // Receive a message to the Receive and Transmit Ports
      localparam RECEIVE_RX = 4;
      localparam RECEIVE_TX = 5;

      // Send and Recieve a message from the other FPGA
      localparam FPGA_TO_FPGA_CONN = 6;

      // Normal state for receiving and transmitting data
      localparam NORMAL = 7;

      localparam END = 8;

      localparam TIME_OUT = 5_000_000;// .5 sec timeout

      assign display_data = state;
//       assign display_data = payload_out[31:0];
      // Server Initialization
      always_ff @(posedge clk_50mhz) begin
```

```
        if(btnc) begin
            counter <= 0;
            timer <= 0;
            payload_in <= 0;
            input_valid <= 0;
            state <= START_RX;
            send_port <= RX_PC_PORT;
            led16_b <= 0;
            led16_r <= 0;
            led17_b <= 0;
            led17_r <= 0;
        end else begin
            case (state)
                START_RX: begin
                    if(~tx_busy) begin
                        send_port <= RX_PC_PORT;
                        state <= SEND_RX;
                        payload_in <= RX_INIT;
                        input_valid <= 1;
                    end else begin
                        input_valid <= 0;
                    end
                end
                SEND_RX: begin
                    input_valid <= 0;
                    if(~tx_busy) begin
                        state <= RECEIVE_RX;
                        timer <= 0;
                    end
                end
                RECEIVE_RX: begin
                    if(payload_out[(23 << 3) - 1 : 0] == RX_CONN) begin
                        led16_b <= 1;
                        led16_r <= 0;
                        state <= START_TX;
                    end else if (timer == TIME_OUT - 1) begin
                        timer <= 0;
                        state <= START_RX;
```

```
                       end else begin
                           timer <= timer + 1;
                       end
                   end
                   START_TX: begin
                       if(~tx_busy) begin
                           send_port <= TX_PC_PORT;
                           state <= SEND_TX;
                           payload_in <= TX_INIT;
                           input_valid <= 1;
                       end else begin
                           input_valid <= 0;
                       end
                   end
                   SEND_TX: begin
                       input_valid <= 0;
                       if(~tx_busy) begin
                           state <= RECEIVE_TX;
                           timer <= 0;
                       end
                   end
                   RECEIVE_TX: begin
                       if(payload_out[(23 << 3) - 1 : 0] == TX_CONN) begin
                           led17_b <= 1;
                           led17_r <= 0;
//                             state <= FPGA_TO_FPGA_CONN;
                           state <= NORMAL;
                       end else if (timer == TIME_OUT - 1) begin
                           timer <= 0;
                           state <= START_TX;
                       end else begin
                           timer <= timer + 1;
                       end
                   end
                   NORMAL: begin
                           if(payload_out[(4 << 3) - 1 : 0] == "STOP")begin
                               led16_b <= 0;
                               led16_r <= 1;
```

```
                             led17_b <= 0;
                             led17_r <= 1;
                             state <= END;
                         end
                         if(~tx_busy) begin
                             input_valid <= 1;
                            // payload_in <= "yo";
                            payload_in <= payload_buffer;
                         end else begin
                             input_valid <= 0;
                         end
                 end
                 END: begin
                     // Do Nothing
                 end
             endcase
         end
    end

 //Beginning of VGA logic (546 bytes for pixel input data)

 logic [(PAYLOAD_BYTES << 3)-1-34:0] camera_packet; //first 34 bits reserved for sensor data
 assign camera_packet = payload_out[(PAYLOAD_BYTES)-1:34];
 logic [33:0] sensor_data;
 assign sensor_data = payload_out[33:0];



 //VGA Display
       vga_display vga_disp(
    .clk_50mhz(clk_50mhz),
    .clk_65mhz(clk_65mhz),
.sw(sw),

.vga_r(vga_r),
.vga_b(vga_b),
.vga_g(vga_g),
.vga_hs(vga_hs),
  .vga_vs(vga_vs),

  .reset(reset),


  .camera_data_in(camera_packet),
   .data_in(sensor_data_vga),
  .led(led)


   );
endmodule
```

// top_level_robot.sv

```
module top_level_robot(
       input clk_100mhz,
       // Ethernet Pins
       input logic eth_mdio,
       inout logic [1:0] eth_rxd,
       inout logic eth_crsdv,// valid receive

       output logic eth_mdc,
       output logic eth_rstn,
       output logic eth_txen,
       output logic [1:0] eth_txd,
       output logic eth_refclk,
       output logic eth_intn,
       output logic eth_rxerr,

       // LEDs
       output logic led16_b,
       output logic led16_r,
       output logic led17_b,
       output logic led17_r,

       output logic [15:0] led,

       // Buttons
       input btnc,
    // Hex display
    output logic ca,cb,cc,cd,ce,cf,cg,
    output logic [7:0] an,

    // switches
    input [15:0] sw,

    // HC-SR04 Pins
    output logic trigger, // jc[0]
    input logic echo,    // jc[1]

    // Camera pins
   input [7:0] ja, //pixel data from camera
   input [2:0] jb, //other data from camera (including clock return)
   output   jbclk, //clock FPGA drives the camera with

   // VGA display
   output logic[3:0] vga_r,
   output logic[3:0] vga_b,
   output logic[3:0] vga_g,
   output logic vga_hs,
   output logic vga_vs,

   // MOTOR A and MOTOR B Pins
   output logic AIA,
   output logic AIB,
   output logic BIA,
   output logic BIB,

   // Button Inputs
   input btnr,
   input btnd,
   input btnl,
   input btnu
);

// Clock variables
```

```
logic clk_50mhz;
logic clk_65mhz;

// 50 mhz and 65 mhz clock instance
clk_wiz_3 clk50_65divider(.clk_in1(clk_100mhz),.clk_out1(clk_50mhz), .clk_out2(clk_65mhz));

// Seven segment display
logic [6:0] segments;
logic [31:0] display_data;      //  Input data for the display
assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];

// Clock assignment
assign eth_refclk = clk_50mhz;

// HC-SR04 Distance sensor Logic
logic [9:0] distance;

// Connection Codes
localparam RX_INIT = "FPGA RX INIT";
localparam RX_CONN = "PC to FPGA RX CONNECTED";
localparam TX_INIT = "FPGA TX INIT";
localparam TX_CONN = "PC to FPGA TX CONNECTED";
localparam ROBOT_TO_CONTROL = "ROBOT -> CONTROL";
localparam CONTROL_TO_ROBOT = "CONTROL -> ROBOT";
// Ports
localparam RX_FPGA_PORT = 5001;
localparam TX_FPGA_PORT = 5001;
localparam RX_PC_PORT = 5003;
localparam TX_PC_PORT = 1024;

// MOTOR CONTROLLER LOGIC

// Commands
// STOP     = 3'b000;
// FORWARD  = 3'b001;
// BACKWARD = 3'b010;
// RIGHT    = 3'b011;
// LEFT     = 3'b100;
```

```
 logic [2:0] motor_ctrl;
 logic [3:0] motor_pins;
 assign motor_pins = {AIA,AIB,BIA,BIB};

 motor_control motors(.control(motor_ctrl),
                      .motor(motor_pins));

 // CAMERA SENSOR LOGIC -------------------------------
 logic xclk;
 logic xclk_count;

 assign xclk = (xclk_count == 2'b01);
 assign jbclk = xclk;

 logic [16:0] pixel_addr_in;
 logic [16:0] pixel_addr_out;

 logic pclk_buff, pclk_in;
 logic vsync_buff, vsync_in;
 logic href_buff, href_in;
 logic[7:0] pixel_buff, pixel_in;

 logic [11:0] frame_buff_out;
 logic [15:0] output_pixels;
 logic [15:0] old_output_pixels;
 logic [12:0] processed_pixels;
 logic valid_pixel;
 logic frame_done_out;

 localparam CAMX_BUFF_SIZE = 320;
 localparam CAMY_BUFF_SIZE = 240;
 localparam CAM_BUFF_SIZE = 76800; // 320 * 240

camera_read  my_camera(.p_clock_in(pclk_in),
                       .vsync_in(vsync_in),
                       .href_in(href_in),
                       .p_data_in(pixel_in),
                      .pixel_data_out(output_pixels),
                      .pixel_valid_out(valid_pixel),
                      .frame_done_out(frame_done_out));

blk_mem_gen_0 bram(.addra(pixel_addr_in),
                         .clka(pclk_in),
                         .dina(processed_pixels),
                         .wea(valid_pixel),
                         .addrb(pixel_addr_out),
                         .clkb(clk_50mhz),
                         .doutb(frame_buff_out));

always_ff @(posedge pclk_in)begin
    if (frame_done_out)begin
        pixel_addr_in <= 17'b0;
    end else if (valid_pixel)begin
        pixel_addr_in <= pixel_addr_in +1;
    end
end

always_ff @(posedge clk_50mhz) begin
    pclk_buff <= jb[0];
    vsync_buff <= jb[1];
    href_buff <= jb[2];
    pixel_buff <= ja;
    pclk_in <= pclk_buff;
    vsync_in <= vsync_buff;
    href_in <= href_buff;
    pixel_in <= pixel_buff;
    old_output_pixels <= output_pixels;
    xclk_count <= xclk_count + 2'b01;
    processed_pixels <= {output_pixels[15:12],output_pixels[10:7],output_pixels[4:1]};
end
 // DISTANCE SENSOR LOGIC ------------------------------
 distance_sensor dist_sensor(.clk_50mhz(clk_50mhz),
                          .button_reset(btnc),
                          .echo(echo),
                          .trigger(trigger),
                             .distance(distance));

   // hex display instance
   display_8hex hex_display(.clk_in(clk_50mhz),.data_in(display_data), .seg_out(segments),
.strobe_out(an));

   // PHY ETHERNET LOGIC ------------------------
   localparam PAYLOAD_BYTES = 548;

   logic phy_rst_done;
   // Restart and initialize the PHY
   phy_init init(
       .clk(clk_50mhz),
       .button_reset(btnc),
       .eth_rxerr(eth_rxerr),
       .eth_intn(eth_intn),
       .eth_crsdv(eth_crsdv),
       .eth_rxd(eth_rxd),
       .eth_rstn(eth_rstn),
       .phy_rst_done(phy_rst_done)
   );

   logic [(PAYLOAD_BYTES << 3) - 1 : 0] payload_out;
   udp_pkt_receive#(.PAYLOAD_BYTES(PAYLOAD_BYTES),
                    .FPGA_IP_1(169),
                    .FPGA_IP_2(254),
                    .FPGA_IP_3(255),
                    .FPGA_IP_4(255),
                    .FPGA_PORT(RX_FPGA_PORT),

                    .PC_IP_1(169),
                    .PC_IP_2(254),
                    .PC_IP_3(70),
                    .PC_IP_4(191),
                    .PC_PORT(RX_PC_PORT)
                       ) rec(
                       .clk(clk_50mhz),
                       .rxd(eth_rxd),
                   .rx_valid(eth_crsdv),
                   .payload_out(payload_out),
                   .button_reset(btnc),
                   .phy_rst_done(phy_rst_done));

logic tx_busy;
logic input_valid;
logic [(PAYLOAD_BYTES << 3) - 1 : 0] payload_in;
logic [15:0] send_port;
udp_pkt_send_test#(.PAYLOAD_BYTES(PAYLOAD_BYTES),
                   .IFG_BYTES(250),

                    .FPGA_IP_1(169),
                    .FPGA_IP_2(254),
                    .FPGA_IP_3(255),
                    .FPGA_IP_4(255),
                   .FPGA_PORT(TX_FPGA_PORT),

                   .PC_IP_1(169),
                   .PC_IP_2(254),
                   .PC_IP_3(70),
                   .PC_IP_4(191)

                   ) send_test (
                   .clk(clk_50mhz),
                   .button_reset(btnc),
                   .txen(eth_txen),
                   .txd(eth_txd),
                   .t_payload(payload_in),
                   .input_valid(input_valid),
                   .tx_busy(tx_busy),
                   .send_port(send_port),
                   .phy_rst_done(phy_rst_done)

                   );

  logic [16:0] counter;
  logic [7:0] count_pixel;
```

```
    logic [31:0] timer;
    logic [3:0] state;

    localparam START_RX = 0;
    localparam START_TX = 1;

    // Send a message to the Receive and Transmit Ports
    localparam SEND_RX = 2;
    localparam SEND_TX = 3;

    // Receive a message to the Receive and Transmit Ports
    localparam RECEIVE_RX = 4;
    localparam RECEIVE_TX = 5;

    // Send and Recieve a message from the other FPGA
    localparam FPGA_TO_FPGA_CONN = 6;

    // Normal state for receiving and transmitting data
    localparam NORMAL = 7;

    localparam END = 8;

    localparam TIME_OUT = 5_000_000;// .5 sec timeout
    localparam PIXELS_PER_PKT = 150;// (548 bytes * 8 - 34 bits)*8/(9 + 8 + 12)

    assign pixel_addr_out = counter;

    assign display_data = payload_out[3:0];

    // Server Initialization
    always_ff @(posedge clk_50mhz) begin
         if(btnc) begin
             counter <= 0;
             count_pixel <= 0;
             payload_in <= 0;
             input_valid <= 0;
             state <= START_RX;
             send_port <= RX_PC_PORT;
```

```
            led16_b <= 0;
            led16_r <= 0;
            led17_b <= 0;
            led17_r <= 0;
            timer <= 0;
        end else begin
            case (state)
                START_RX: begin
                    if(~tx_busy) begin
                        send_port <= RX_PC_PORT;
                        state <= SEND_RX;
                        payload_in <= RX_INIT;
                        input_valid <= 1;
                    end else begin
                        input_valid <= 0;
                    end
                end
                SEND_RX: begin
                    input_valid <= 0;
                    if(~tx_busy) begin
                        state <= RECEIVE_RX;
                        timer <= 0;
                    end
                end
                RECEIVE_RX: begin
                    if(payload_out[(23 << 3) - 1 : 0] == RX_CONN) begin
                        led16_b <= 1;
                        led16_r <= 0;
                        state <= START_TX;
                    end else if (timer == TIME_OUT - 1) begin
                        timer <= 0;
                        state <= START_RX;
                    end else begin
                        timer <= timer + 1;
                    end
                end
                START_TX: begin
                    if(~tx_busy) begin
```

```
                            send_port <= TX_PC_PORT;
                            state <= SEND_TX;
                            payload_in <= TX_INIT;
                            input_valid <= 1;
                        end else begin
                            input_valid <= 0;
                        end
                    end
                    SEND_TX: begin
                        input_valid <= 0;
                        if(~tx_busy) begin
                            state <= RECEIVE_TX;
                            timer <= 0;
                        end
                    end
                    RECEIVE_TX: begin
                        if(payload_out[(23 << 3) - 1 : 0] == TX_CONN) begin
                            led17_b <= 1;
                            led17_r <= 0;
//                              state <= FPGA_TO_FPGA_CONN;
                            state <= NORMAL;
                            timer <= 0;
                        end else if (timer == TIME_OUT - 1) begin
                            timer <= 0;
                            state <= START_TX;
                        end else begin
                            timer <= timer + 1;
                        end
                    end
                    NORMAL: begin
                            //Receiving
                            if(payload_out[(4 << 3) - 1 : 0] == "STOP")begin
                                led16_b <= 0;
                                led16_r <= 1;
                                led17_b <= 0;
                                led17_r <= 1;
                                state <= END;
                            end else begin
```

```
                               // Motor Control Information
                               motor_ctrl <= payload_out[2:0];
                           end

                           // Sending
                           if(~tx_busy && count_pixel == PIXELS_PER_PKT - 1) begin
                               input_valid <= 1;
                               if(sw[0]) begin
                                   payload_in <= "STOP";
                               end
                           end else begin
                               input_valid <= 0;
                           end
                           if(count_pixel == 0 && ~sw[0]) begin
                               // Distance sensor 34 bits(happens once per packet sent)
                               payload_in[0+:34] <= distance;
                               // counter is x_pos + y_pos*320 (17 bits)
                               // Pixel data 12 bits
                               payload_in[34+:17] <= counter;
                               payload_in[51+:12] <= frame_buff_out;
                           end else if(~sw[0]) begin
                               payload_in[(63 + 5'd29 * (count_pixel - 1))+:17] <= counter;
                               payload_in[(80 + 5'd29 * (count_pixel - 1))+:12] <= frame_buff_out;
                           end

                           counter <= (counter == CAM_BUFF_SIZE - 1) ? 0 : counter + 1;
                           count_pixel <= (count_pixel == PIXELS_PER_PKT - 1) ? 0 : count_pixel + 1;
                   end
                   END: begin
                       // Do Nothing
                   end
               endcase
           end
      end
endmodule
```

// udp_pkt_receive.py

```
// Note nibbles is 4-bits. snibbles means semi-nibble(or di-bits)which is 2-bits
module udp_pkt_receive
#(
   // "FPGA IP" - put an unused IP
   parameter FPGA_IP_1 = 0,
   parameter FPGA_IP_2 = 0,
   parameter FPGA_IP_3 = 0,
   parameter FPGA_IP_4 = 0,
   // FPGA Port
   parameter FPGA_PORT = 5000,
   // PC IP address for incoming data
   parameter PC_IP_1 = 0,
   parameter PC_IP_2 = 0,
   parameter PC_IP_3 = 0,
   parameter PC_IP_4 = 0,
   // PC Port
   parameter PC_PORT = 5000,
   // Payload size parameter
   parameter PAYLOAD_BYTES = 30,
   // Power Up parameter
   parameter POWER_UP_CYCLES = 23'd8_000_000
)
(
   input                  clk,
   input     logic        button_reset,
   input     logic [1:0]  rxd,
   input     logic        rx_valid,
   input     logic        phy_rst_done,

   output    logic [(PAYLOAD_BYTES << 3) - 1 : 0] payload_out,
   output    logic        valid_indicator,
   output    logic        error_indicator,
   output    logic        display_data
);
   // The number of bytes in parts of the Ethernet Frame
   localparam int unsigned PREAMBLE_SFD_BYTES = 8;
   localparam int unsigned MAC_HEADER_BYTES   = 14;
   localparam int unsigned IP_HEADER_BYTES    = 20;
   localparam int unsigned UDP_HEADER_BYTES   = 8;
   localparam int unsigned CRC_BYTES          = 4;
   localparam int unsigned IFG_BYTES          = 12; // Ethernet Interframe Gap

   // The number of semi-nibbles in parts of the frame
   localparam int unsigned PREAMBLE_SFD_SNIBBLES = 4 * PREAMBLE_SFD_BYTES;
   localparam int unsigned MAC_HEADER_SNIBBLES   = 4 * MAC_HEADER_BYTES;
   localparam int unsigned IP_HEADER_SNIBBLES    = 4 * IP_HEADER_BYTES;
   localparam int unsigned UDP_HEADER_SNIBBLES   = 4 * UDP_HEADER_BYTES;
   localparam int unsigned CRC_SNIBBLES          = 4 * CRC_BYTES;// CRC
   localparam int unsigned IFG_SNIBBLES          = 4 * IFG_BYTES;

   logic [31:0] ip_checksum;
   logic [17:0] payload_length_snibbles;

   logic [(PREAMBLE_SFD_BYTES << 3) - 1 : 0] preamble;
   logic [(MAC_HEADER_BYTES << 3) - 1 : 0]   mac_header;
   logic [(IP_HEADER_BYTES << 3) - 1 : 0]   ip_header;
   logic [(UDP_HEADER_BYTES << 3) - 1 : 0]   udp_header;
   logic [(PAYLOAD_BYTES << 3) - 1 : 0] payload_data;

   // Received CRC
   logic [(CRC_BYTES << 3) - 1 : 0] crc32_rx;
   // Calculated CRC for verification
   logic [(CRC_BYTES << 3) - 1 : 0] crc32_cal;

   // Counters
   logic [31:0] global_counter;
   logic [1:0] snibble_counter;
   logic [22:0] power_up_counter;
   logic [5:0] ifg_counter;

   // States
   localparam POWER_UP = 0;
   localparam IDLE = 1;
   localparam PREAMBLE = 2;
   localparam MAC_HEADER = 3;// same as the ethernet header
   localparam IP_HEADER = 4;
   localparam UDP_HEADER = 5;
   localparam PAYLOAD = 6;
   localparam CRC = 7;
   localparam IFG = 8;    // Ethernet Interframe Gap

   logic [3:0] state;

   logic [7:0] temp_reg;

   always_ff @(posedge clk)begin
       if(button_reset)begin
           // Indicators
           valid_indicator <= 0;
           error_indicator <= 0;
           display_data <= 0;
           // Counters
           global_counter <= 0;
           snibble_counter <= 0;
           power_up_counter <= 0;
           ifg_counter <= 0;
           // State
           state <= POWER_UP;
           // Incoming data
           preamble <=0;
           mac_header <= 0;
           ip_header <= 0;
           udp_header <= 0;
           payload_data <= 0;
           crc32_rx <= 0;
           crc32_cal <= 32'hFF_FF_FF_FF;
           temp_reg <= 0;
           // output
           payload_out <= 0;
       end
       else begin
           case(state)
               // Reception Power Up
               POWER_UP: begin
                   if(power_up_counter == POWER_UP_CYCLES - 1)begin
                       state <= IDLE;
                   end
               end
               IDLE: begin
                   if(rx_valid)begin
                       if (rxd == 2'b01)begin
                           state <= PREAMBLE;
                           global_counter <= 1;// start the global counter
                           snibble_counter <= 1;// start the snibble counter
                           ifg_counter <= 0;  // clear the ifg counter

                           preamble <=0; // Clear the preamble
                           crc32_cal <= 32'hFF_FF_FF_FF;// Reset the CRC
                       end
                       temp_reg <= (temp_reg << 2) | rxd;
                   end
               end
               PREAMBLE: begin
                   if(snibble_counter == 0)begin
                       preamble <= (preamble << 8) | swap_nibbles(temp_reg);
                   end
                   temp_reg <= (temp_reg << 2) | rxd;
                   // Verify the correct preamble
                   if(((preamble << 8) | swap_nibbles(temp_reg)) == 64'h55_55_55_55_55_55_55_D5)
                            state <= MAC_HEADER;
                   global_counter <= PREAMBLE_SFD_SNIBBLES + 4 - 1;
               end
               MAC_HEADER: begin
                   if(snibble_counter == 0)begin
                       mac_header <= (mac_header << 8) | swap_nibbles(temp_reg);
                       crc32_cal <= compute_crc(crc32_cal, swap_nibbles(temp_reg));
                   end
                   temp_reg <= (temp_reg << 2) | rxd;
                   if(global_counter == PREAMBLE_SFD_SNIBBLES +
                                        MAC_HEADER_SNIBBLES   + 4 - 1) state <= IP_HEADER;
               end
               IP_HEADER: begin
```

```
            if(snibble_counter == 0)begin
                ip_header <= (ip_header << 8) | swap_nibbles(temp_reg);
                crc32_cal <= compute_crc(crc32_cal, swap_nibbles(temp_reg));
            end
            temp_reg <= (temp_reg << 2) | rxd;
            if(global_counter == PREAMBLE_SFD_SNIBBLES +
                                 MAC_HEADER_SNIBBLES   +
                                 IP_HEADER_SNIBBLES    + 4 - 1) state <= UDP_HEADER;
        end
        UDP_HEADER: begin
            // Save the checksum
            ip_checksum <=          (ip_header[144+:16] +
                                     ip_header[128+:16] +
                                     ip_header[112+:16] +
                                     ip_header[96+:16]  +
                                     ip_header[80+:16]  +
                                     ip_header[48+:16]  +
                                     ip_header[32+:16]  +
                                     ip_header[16+:16]  +
                                     ip_header[0+:16]    );
            payload_length_snibbles <= (ip_header[143:128] -
                                        IP_HEADER_BYTES    -
                                        UDP_HEADER_BYTES) << 2;
            if(snibble_counter == 0)begin
                udp_header <= (udp_header << 8) | swap_nibbles(temp_reg);
                crc32_cal <= compute_crc(crc32_cal, swap_nibbles(temp_reg));
            end
            temp_reg <= (temp_reg << 2) | rxd;
            if(global_counter == PREAMBLE_SFD_SNIBBLES +
                                 MAC_HEADER_SNIBBLES   +
                                 IP_HEADER_SNIBBLES    +
                                 UDP_HEADER_SNIBBLES   + 4 - 1) state <= PAYLOAD;
        end
        PAYLOAD: begin
            // Skip the payload if the checksum is invalid
            if(ip_header[64+:16] != ~(ip_checksum[31:16] + ip_checksum[15:0]))begin
                state <= IFG;
            end
```

```
                   if(snibble_counter == 0)begin
                       payload_data <= (payload_data << 8) | swap_nibbles(temp_reg);
                       crc32_cal <= compute_crc(crc32_cal, swap_nibbles(temp_reg));
                   end
                   temp_reg <= (temp_reg << 2) | rxd;
                   if(global_counter == PREAMBLE_SFD_SNIBBLES +
                                        MAC_HEADER_SNIBBLES   +
                                        IP_HEADER_SNIBBLES    +
                                        UDP_HEADER_SNIBBLES   +
                                        payload_length_snibbles + 4 - 1) state <= CRC;
               end
               CRC: begin
                   if(snibble_counter == 0)begin
                       crc32_rx <= (crc32_rx << 8) | swap_nibbles(temp_reg);
                   end
                   temp_reg <= (temp_reg << 2) | rxd;
                   if(global_counter == PREAMBLE_SFD_SNIBBLES +
                                        MAC_HEADER_SNIBBLES   +
                                        IP_HEADER_SNIBBLES    +
                                        UDP_HEADER_SNIBBLES   +
                                        payload_length_snibbles +
                                        CRC_SNIBBLES + 4 - 1) state <= IFG;
               end
               IFG: begin // Ethernet Interframe gap or END stage
                   // Verification
                   if(preamble == 64'h55_55_55_55_55_55_55_D5
                       && mac_header[15:0] == 16'h08_00
                       && ip_header[159:144] == 16'h45_00
                       // CRC authentication.
                       && reverse_and_invert(crc32_cal) ==
{crc32_rx[0+:8],crc32_rx[8+:8],crc32_rx[16+:8],crc32_rx[24+:8] }
                       // IP checksum authentication
                       && ip_header[64+:16]  == ~(ip_checksum[31:16] + ip_checksum[15:0])
                       // PC IP authentication
                       && ip_header[56+:8]   == PC_IP_1
                       && ip_header[48+:8]   == PC_IP_2
                       && ip_header[40+:8]   == PC_IP_3
                       && ip_header[32+:8]   == PC_IP_4
```

```
             //FPGA Port authentication
            && udp_header[32+:16] == FPGA_PORT
             //FPGA IP authentication
            && ip_header[24+:8]   == FPGA_IP_1
            && ip_header[16+:8]   == FPGA_IP_2
            && ip_header[8+:8]    == FPGA_IP_3
            && ip_header[0+:8]    == FPGA_IP_4
            )
        begin
            // Valid data that can be outputted
            payload_out <= payload_data;
        end

        // End this one clock cycle before
        // you need to go to IDLE
        if(ifg_counter == IFG_SNIBBLES - 4) begin
            // Need to reset these
            state <= IDLE;
            preamble <=0;
            payload_data <= 0;
            crc32_cal <= 32'hFF_FF_FF_FF;

            // reset for good practice
            global_counter <= 1;
            snibble_counter <= 1;
            ifg_counter <=0;

            mac_header <= 0;
            ip_header <= 0;
            udp_header <= 0;
            crc32_rx <= 0;
            temp_reg <= 0;
        end else if (~rx_valid) begin
            ifg_counter <= ifg_counter + 1;
        end
    end
    default: state <= IDLE;
endcase
```

```
        if (state != IDLE && state != IFG) begin
            snibble_counter <= snibble_counter + 1;
        end
        if (state != IDLE && state != PREAMBLE && state != IFG)begin
            global_counter <= global_counter + 1;
        end
        if(state == POWER_UP && phy_rst_done) begin
            power_up_counter <= power_up_counter + 1;
        end
    end
end

//Reverse nibbles in a byte and semi-nibbles(di-bits) in a nibble
// ex. In: ab_cd_ef_gh ; Out: gh_ef_cd_ab
function logic [7:0] swap_nibbles(input logic [7:0] data);
    swap_nibbles = {data[1:0],data[3:2],
                         data[5:4],data[7:6]};
endfunction

// CRC-32 algorithm: Github Adam Christiansen MIT License
// For computing crc32
function [31:0] compute_crc(input logic [31:0] crc,
                                  input logic [7:0] data);
         localparam int unsigned POLYNOMIAL = 32'h04_C1_1D_B7;
         compute_crc = crc;
         for(int j = 0;j<8;j++)begin
                compute_crc = {compute_crc[30:0], 1'b0} ^
                                (data[j] == compute_crc[31] ? 0 : POLYNOMIAL);
         end
endfunction

//Reverse and invert bits in a 32-bit word
function [31:0] reverse_and_invert(input logic [31:0] data);
    for(int i =0; i< 32; i++)begin
        reverse_and_invert[i] = ~data[31-i];
    end
endfunction
```

```
endmodule
```

//udp_pkt_send_test.sv

```
module udp_pkt_send_test
#(
   // "IP source" - put an unused IP
   parameter FPGA_IP_1 = 0,
   parameter FPGA_IP_2 = 0,
   parameter FPGA_IP_3 = 0,
   parameter FPGA_IP_4 = 0,
   // FPGA PORT
   parameter FPGA_PORT = 5000,
   // "IP destination" - put the IP of the PC/Server you want to send to
   parameter PC_IP_1 = 0,
   parameter PC_IP_2 = 0,
   parameter PC_IP_3 = 0,
   parameter PC_IP_4 = 0,
   // "Physical Address" - put the address of the PC/Server you want to send to - default broadcast
   parameter PC_ADDR_1 = 8'hff,
   parameter PC_ADDR_2 = 8'hff,
   parameter PC_ADDR_3 = 8'hff,
   parameter PC_ADDR_4 = 8'hff,
   parameter PC_ADDR_5 = 8'hff,
   parameter PC_ADDR_6 = 8'hff,
   //FPGA Physical address: Can be anything really
   parameter FPGA_ADDR_1 = 8'haa,
   parameter FPGA_ADDR_2 = 8'hde,
   parameter FPGA_ADDR_3 = 8'had,
   parameter FPGA_ADDR_4 = 8'hbe,
   parameter FPGA_ADDR_5 = 8'hef,
   parameter FPGA_ADDR_6 = 8'haa,
   // Ethernet Parameters
   parameter PAYLOAD_BYTES = 30, // Must be at least 18 for a total of at least 64 bytes
   parameter POWER_UP_CYCLES = 5_000_000 + 400,
    // Ethernet Interframe Gap for processing payload data
   parameter IFG_BYTES       = 12,// Min is 12. Increase for more latency between packets
   //Image params for later.
   parameter IM_X = 640,
   parameter IM_Y = 480,
   parameter COLOR_MODE = 1
)
(
   input                                        clk,
   input    logic                               button_reset,
   input    logic[(PAYLOAD_BYTES << 3) - 1:0]   t_payload,
   input    logic                               input_valid,    // should be a pulse
   input    logic                               phy_rst_done,
   input    logic                               [15:0] send_port,

   output   logic                               tx_busy,
   output   logic                               txen,
   output   logic                               [1:0] txd
);

       // The number of bytes in parts of the Ethernet Frame
   localparam int unsigned PREAMBLE_SFD_BYTES = 8;
   localparam int unsigned MAC_HEADER_BYTES   = 14;
   localparam int unsigned IP_HEADER_BYTES    = 20;
   localparam int unsigned UDP_HEADER_BYTES   = 8;
   localparam int unsigned CRC_BYTES          = 4;

   localparam int unsigned UDP_LENGTH = UDP_HEADER_BYTES + PAYLOAD_BYTES;// in bytes
   localparam int unsigned IP_LENGTH = IP_HEADER_BYTES + UDP_HEADER_BYTES + PAYLOAD_BYTES;// in bytes

   // The number of semi-nibbles in parts of the frame
   localparam int unsigned PREAMBLE_SFD_SNIBBLES = 4 * PREAMBLE_SFD_BYTES;
   localparam int unsigned MAC_HEADER_SNIBBLES   = 4 * MAC_HEADER_BYTES;
   localparam int unsigned IP_HEADER_SNIBBLES    = 4 * IP_HEADER_BYTES;
   localparam int unsigned UDP_HEADER_SNIBBLES   = 4 * UDP_HEADER_BYTES;
   localparam int unsigned CRC_SNIBBLES          = 4 * CRC_BYTES;// CRC
   localparam int unsigned IFG_SNIBBLES          = 4 * IFG_BYTES;// interframe gap
   localparam int unsigned PAYLOAD_SNIBBLES      = 4 * PAYLOAD_BYTES;// interframe gap


   // Payload SNIBBLE Length is (IP_LENGTH(read from the header) - IP_HEADER_BYTES -
UDP_HEADER_BYTES)*4
```

```
// Payload SNIBBLE Length is also (UDP_LENGTH(read from the header) - UDP_HEADER_BYTES) * 4
// The Ethernet type for the Ethernet header. This value indicates that
// IPv4 is used.
localparam int unsigned ETHER_TYPE = 16'h0800;

// The IP version to use.
localparam int unsigned IP_VERSION = 4;

// The IP header length in 32-bit words.
localparam int unsigned IP_HEADER_LENGTH = 5;

// The IP type of service.
localparam int unsigned IP_TOS = 8'h00;

// The IP fragment identification.
localparam int unsigned IP_ID = 16'h0000;

// The IP flags.
localparam int unsigned IP_FLAGS = 4'h0;

// The IP fragmentation offset.
localparam int unsigned IP_FRAGMENTATION_OFFSET = 12'h000;

// The IP time to live.
localparam int unsigned IP_TTL = 255;

// The IP next level protocol to use. This is the User Datagram Protocol.
localparam int unsigned IP_PROTOCOL = 8'h11;

// verifies the validity of the ip header
localparam int unsigned IP_CHECKSUM = {IP_VERSION[3:0],IP_HEADER_LENGTH[3:0], IP_TOS[7:0]} +
                                      IP_LENGTH[15:0]+
                                      IP_ID[15:0] +
                                      {IP_FLAGS[3:0],IP_FRAGMENTATION_OFFSET[11:0]} +
                                      {IP_TTL[7:0],IP_PROTOCOL[7:0]} +
                                      {FPGA_IP_1[7:0],FPGA_IP_2[7:0]}  +
                                      {FPGA_IP_3[7:0], FPGA_IP_4[7:0]}  +
                                      {PC_IP_1[7:0],PC_IP_2[7:0]} +
                                      {PC_IP_3[7:0],PC_IP_4[7:0]} ;

localparam int unsigned UDP_CHECKSUM = 16'h0000;// Optional but it's normally zero

logic [(CRC_BYTES << 3) - 1 : 0] crc32;

logic [31:0] global_counter;// The size should actually be around 17 bits max

logic [1:0] snibble_counter;
logic [22:0] power_up_counter;
logic [10:0] ifg_counter;

localparam POWER_UP = 0;
localparam IDLE = 1;
localparam PREAMBLE = 2;
localparam IP_HEADER = 4;
localparam MAC_HEADER = 3;// same as the ethernet header
localparam UDP_HEADER = 5;
localparam PAYLOAD = 6;
localparam CRC = 7;
// Ethernet Interframe gap where you can chill for 12 bytes time or 48 snibbles
// It is also the end stage so if you have corrupted data it'll just to this stage as well
localparam IFG = 8;

logic [3:0] state;
// Temporary register to hold a 4-bit nibble
logic [7:0] temp_reg;

assign state_out = state;
logic [7:0] pkt_data[0:PREAMBLE_SFD_BYTES +
                       MAC_HEADER_BYTES   +
                       IP_HEADER_BYTES    +
                       UDP_HEADER_BYTES   +
                       PAYLOAD_BYTES      +
                       CRC_BYTES - 1];

assign tx_busy = (state == IDLE && ~input_valid) ? 0 : 1;
```

```
always_ff @(posedge clk)begin
    if(button_reset)begin
        //Counters
        global_counter <= 0;
        snibble_counter <= 0;
        power_up_counter <= 0;
        ifg_counter <=0;
        // State
        state <= POWER_UP;
        temp_reg <= 0;
        // TX init
        txen <= 0;
        txd <= 0;
        // crc
        crc32 <= 32'hFF_FF_FF_FF;
    end
    else begin
        case(state)
            // Rough power up time for the Ethernet PHY
            POWER_UP: begin
                if(power_up_counter == POWER_UP_CYCLES - 1)begin
                    state <= IDLE;
                end
            end
            IDLE: begin
                // Preamble
                pkt_data[0] <= swap_nibbles(8'h55);
                pkt_data[1] <= swap_nibbles(8'h55);
                pkt_data[2] <= swap_nibbles(8'h55);
                pkt_data[3] <= swap_nibbles(8'h55);
                pkt_data[4] <= swap_nibbles(8'h55);
                pkt_data[5] <= swap_nibbles(8'h55);
                pkt_data[6] <= swap_nibbles(8'h55);
                pkt_data[7] <= swap_nibbles(8'hD5);
                // Ethernet MAC Header
                pkt_data[8] <= swap_nibbles(PC_ADDR_1);
                pkt_data[9] <= swap_nibbles(PC_ADDR_2);
                pkt_data[10] <= swap_nibbles(PC_ADDR_3);
            pkt_data[11] <= swap_nibbles(PC_ADDR_4);
            pkt_data[12] <= swap_nibbles(PC_ADDR_5);
            pkt_data[13] <= swap_nibbles(PC_ADDR_6);
            pkt_data[14] <= swap_nibbles(FPGA_ADDR_1);
            pkt_data[15] <= swap_nibbles(FPGA_ADDR_2);
            pkt_data[16] <= swap_nibbles(FPGA_ADDR_3);
            pkt_data[17] <= swap_nibbles(FPGA_ADDR_4);
            pkt_data[18] <= swap_nibbles(FPGA_ADDR_5);
            pkt_data[19] <= swap_nibbles(FPGA_ADDR_6);
            pkt_data[20] <= swap_nibbles(ETHER_TYPE[15:8]);
            pkt_data[21] <= swap_nibbles(ETHER_TYPE[7:0]);
            //IP header
            pkt_data[22] <= swap_nibbles({IP_VERSION[3:0],IP_HEADER_LENGTH[3:0]});
            pkt_data[23] <= swap_nibbles(IP_TOS[7:0]);
            pkt_data[24] <= swap_nibbles(IP_LENGTH[15:8]);
            pkt_data[25] <= swap_nibbles(IP_LENGTH[7:0]);
            pkt_data[26] <= swap_nibbles(IP_ID[15:8]);
            pkt_data[27] <= swap_nibbles(IP_ID[7:0]);
            pkt_data[28] <= swap_nibbles({IP_FLAGS[3:0],IP_FRAGMENTATION_OFFSET[11:8]});
            pkt_data[29] <= swap_nibbles(IP_FRAGMENTATION_OFFSET[7:0]);
            pkt_data[30] <= swap_nibbles(IP_TTL[7:0]);
            pkt_data[31] <= swap_nibbles(IP_PROTOCOL[7:0]);
            pkt_data[32] <= swap_nibbles((~(IP_CHECKSUM[31:16] + IP_CHECKSUM[15:0])) >> 8);
            pkt_data[33] <= swap_nibbles((~(IP_CHECKSUM[31:16] + IP_CHECKSUM[15:0])));
            pkt_data[34] <= swap_nibbles(FPGA_IP_1);
            pkt_data[35] <= swap_nibbles(FPGA_IP_2);
            pkt_data[36] <= swap_nibbles(FPGA_IP_3);
            pkt_data[37] <= swap_nibbles(FPGA_IP_4);
            pkt_data[38] <= swap_nibbles(PC_IP_1);
            pkt_data[39] <= swap_nibbles(PC_IP_2);
            pkt_data[40] <= swap_nibbles(PC_IP_3);
            pkt_data[41] <= swap_nibbles(PC_IP_4);
            //UDP Header
            pkt_data[42] <= swap_nibbles(FPGA_PORT[15:8]);
            pkt_data[43] <= swap_nibbles(FPGA_PORT[7:0]);
            pkt_data[44] <= swap_nibbles(send_port[15:8]);
            pkt_data[45] <= swap_nibbles(send_port[7:0]);
            pkt_data[46] <= swap_nibbles(UDP_LENGTH[15:8]);
```

```
                pkt_data[47] <= swap_nibbles(UDP_LENGTH[7:0]);
                pkt_data[48] <= swap_nibbles(UDP_CHECKSUM[15:8]);
                pkt_data[49] <= swap_nibbles(UDP_CHECKSUM[7:0]);
                // PAYLOAD INFO
                for(int i = 0;i < PAYLOAD_BYTES;i++ )begin
                    pkt_data[i + 50] <= swap_nibbles(t_payload[((PAYLOAD_BYTES - 1 - i) <<
3)+:8]);
                end
                // CRC32 calculate later (4 bytes)
                pkt_data[PAYLOAD_BYTES + 50 + 0] <= 8'h00;
                pkt_data[PAYLOAD_BYTES + 50 + 1] <= 8'h00;
                pkt_data[PAYLOAD_BYTES + 50 + 2] <= 8'h00;
                pkt_data[PAYLOAD_BYTES + 50 + 3] <= 8'h00;

                if(input_valid) begin
                    state <= PREAMBLE;
                    temp_reg <= pkt_data[global_counter >> 2];

                    global_counter <= 1;// start the global counter at 1
                    snibble_counter <= 1;// start the snibble counter  at 1
                    ifg_counter <=0;
                end
            end
            PREAMBLE: begin
                txen <= 1;
                if (snibble_counter==0) begin
                    //next byte
                    temp_reg <= pkt_data[global_counter >> 2];
                end else begin
                    temp_reg <= (temp_reg << 2);
                end
                txd <= temp_reg[7:6];
                if(global_counter == PREAMBLE_SFD_SNIBBLES - 1) state<=MAC_HEADER;
            end
            MAC_HEADER: begin
                if (snibble_counter==0) begin
                    //next byte
                    temp_reg <= pkt_data[global_counter >> 2];
                // return the pkt data to normal for the crc calculation
                crc32 <= compute_crc(crc32, swap_nibbles(pkt_data[global_counter >> 2]));
            end else begin
                temp_reg <= (temp_reg << 2);
            end
            txd <= temp_reg[7:6];
            if(global_counter == PREAMBLE_SFD_SNIBBLES +
                                 MAC_HEADER_SNIBBLES   - 1)  state<=IP_HEADER;
        end
        IP_HEADER: begin
            if (snibble_counter==0) begin
                //next byte
                temp_reg <= pkt_data[global_counter >> 2];
                crc32 <= compute_crc(crc32, swap_nibbles(pkt_data[global_counter >> 2]));
            end else begin
                temp_reg <= (temp_reg << 2);
            end
            txd <= temp_reg[7:6];
            if(global_counter == PREAMBLE_SFD_SNIBBLES +
                                 MAC_HEADER_SNIBBLES  +
                                 IP_HEADER_SNIBBLES - 1) state<=UDP_HEADER;
        end
        UDP_HEADER: begin
            if (snibble_counter==0) begin
                //next byte
                temp_reg <= pkt_data[global_counter >> 2];
                crc32 <= compute_crc(crc32, swap_nibbles(pkt_data[global_counter >> 2]));
            end else begin
                temp_reg <= (temp_reg << 2);
            end
            txd <= temp_reg[7:6];
            if(global_counter == PREAMBLE_SFD_SNIBBLES +
                                 MAC_HEADER_SNIBBLES   +
                                 IP_HEADER_SNIBBLES    +
                                 UDP_HEADER_SNIBBLES    - 1) state <= PAYLOAD;
        end
        PAYLOAD: begin
            if (snibble_counter==0) begin
```

```
                           temp_reg <=  pkt_data[global_counter >> 2];
                           crc32 <=  compute_crc(crc32, swap_nibbles(pkt_data[global_counter >> 2]));
                   end else begin
                       temp_reg <= (temp_reg << 2);
                   end
                   txd <= temp_reg[7:6];
                   if(global_counter == PREAMBLE_SFD_SNIBBLES +
                                        MAC_HEADER_SNIBBLES   +
                                        IP_HEADER_SNIBBLES    +
                                        UDP_HEADER_SNIBBLES   +
                                        PAYLOAD_SNIBBLES  - 1) begin
                       state <= CRC;
                       // CRC is in little-endian by bytes
                       pkt_data[PAYLOAD_BYTES + 50 + 0] <= swap_nibbles(reverse_and_invert(crc32));
                       pkt_data[PAYLOAD_BYTES + 50 + 1] <= swap_nibbles(reverse_and_invert(crc32) >>
8);
                       pkt_data[PAYLOAD_BYTES + 50 + 2] <= swap_nibbles(reverse_and_invert(crc32) >>
16);
                       pkt_data[PAYLOAD_BYTES + 50 + 3] <= swap_nibbles(reverse_and_invert(crc32) >>
24);
                   end
               end
               CRC: begin
                   if (snibble_counter==0) begin
                       //next byte
                       temp_reg <= pkt_data[global_counter >> 2];
                   end else begin
                       temp_reg <= (temp_reg << 2);
                   end
                   txd <= temp_reg[7:6];
                   if(global_counter == PREAMBLE_SFD_SNIBBLES +
                                        MAC_HEADER_SNIBBLES   +
                                        IP_HEADER_SNIBBLES    +
                                        UDP_HEADER_SNIBBLES   +
                                        PAYLOAD_SNIBBLES +
                                        CRC_SNIBBLES  - 1) state <= IFG;
               end
               IFG: begin // Ethernet Interframe gap or END stage
```

```
                txd <=   (ifg_counter == 0) ? temp_reg[7:6] : 0 ;
                txen <=  (ifg_counter == 0) ? 1 : 0 ;
                if(ifg_counter == IFG_SNIBBLES - 1) begin
                    // Need to reset these
                    state <= IDLE;
                    crc32 <= 32'hFF_FF_FF_FF;
                    // These ones are cleared for good practice
                    temp_reg <= 0;
                    snibble_counter <= 0;
                    global_counter <= 0;
                    power_up_counter <= 0;
                    ifg_counter <=0;
                end else begin
                    ifg_counter <= ifg_counter + 1;
                end
            end
            default: state <= IDLE;
        endcase

        if (state != IDLE && state != IFG) begin
            snibble_counter <= snibble_counter + 1;
            global_counter <= global_counter + 1;
        end

        if(state == POWER_UP && phy_rst_done) begin
            power_up_counter <= power_up_counter + 1;
        end
    end
end

// CRC-32 algorithm: Github Adam Christiansen MIT License
// For computing the checksum
function [31:0] compute_crc(input logic [31:0] crc,
                                  input logic [7:0] data);
         localparam int unsigned POLYNOMIAL = 32'h04_C1_1D_B7;
         compute_crc = crc;
         for(int j = 0;j<8;j++)begin
                compute_crc = {compute_crc[30:0], 1'b0} ^
```

```
                                    (data[j] == compute_crc[31] ? 0 : POLYNOMIAL);
            end
   endfunction

   //Reverse nibbles in a byte and semi-nibbles(di-bits) in a nibble
   // ex. In: ab_cd_ef_gh ; Out: gh_ef_cd_ab
   function [7:0] swap_nibbles(input logic [7:0] data);
       swap_nibbles = {data[1:0],data[3:2],
                            data[5:4],data[7:6]};
   endfunction
    //Reverse and invert bits in a byte
   function [31:0] reverse_and_invert(input logic [31:0] data);
       for(int i =0; i< 32; i++)begin
           reverse_and_invert[i] = ~data[31-i];
       end
   endfunction
endmodule
```

// vga_display.sv

```
localparam CAMERA_BITS = 548*8-34; //number of bits for packet of camera data

module vga_display(
  input clk_50mhz,
  input clk_65mhz,
  input[15:0] sw,
  input reset,
//   input btnc, btnu, btnl, btnr, btnd,

   input [9:0] data_in, //sensor data to be displayed



   input [CAMERA_BITS-1:0] camera_data_in, //to be displayed (addr, raw data)

 //VGA interface logics
  output logic[3:0] vga_r,
  output logic[3:0] vga_b,
  output logic[3:0] vga_g,
  output logic vga_hs,
  output logic vga_vs,
// leds for switches
  output logic [15:0] led



   );




   assign led = sw;                        // turn leds on

//general VGA registers
   logic [10:0] hcount;    // pixel on current line
   logic [9:0] vcount;     // line number
   logic hsync, vsync;
   logic [11:0] rgb; //rgb value of current pixel

//rgb value of the current pixel of the numbers being displayed
   logic [11:0] number_pixel;




   ///CAMERA PIXEL REGISTERS

   logic [11:0] camera_pixel;
   logic [11:0] double_camera_pixel; //for 2x resolution



   logic [11:0] raw_cam_data; //from the packet of camera data
   logic [16:0] cam_addr; //the address of the current pixel to be used to read and write from the
RAM



   logic [12:0] counter = 0; //counter for iterating through all of the pixel data values in a
packet
   always_ff @(posedge clk_50mhz) begin

      cam_addr <= camera_data_in[counter+:17];   //address requires 17 bits
      raw_cam_data<=camera_data_in[(counter+17)+:12]; //pixel data requires 12 bits

      if (counter>=149*29) begin //restart after the 150th (or last) pixel in the data packet
          counter<=0;

      end
      else begin
          counter <= counter+29;  //increase index by 29 (12 bits for pixel value, 17 for address
value)
      end

   end



   xvga xvga1(.vclock_in(clk_65mhz),.reset_in(reset), .hcount_out(hcount),.vcount_out(vcount),
         .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));




   logic phsync,pvsync,pblank; //number sync


   logic chsync, cvsync, cblank; //camera sync


   number_image number(
      .vclock_in(clk_65mhz),
      .reset_in(reset),

      .data_in(data_in),
               .hcount_in(hcount),.vcount_in(vcount),
               .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
               .phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),.pixel_out(number_pixel));
```

```
camera_to_picture camera_1(
 .clk_50mhz(clk_50mhz),
 .pixel_clk_in(clk_65mhz),
 .hcount_in(hcount),
 .vcount_in(vcount),

 .addr_in(cam_addr),
 .raw_data_in(raw_cam_data),
 .double_on(sw[2]), //turn on switch 2 for 640x320 image


 .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
 .chsync_out(chsync),.cvsync_out(cvsync),.cblank_out(cblank),



.pixel_out(camera_pixel),
.double_pixel_out(double_camera_pixel)
        );





logic b,hs,vs;
always_ff @(posedge clk_65mhz) begin


  if (sw[1:0] == 2'b01) begin // just number setting
     hs <= phsync;
     vs <= pvsync;
     b <= pblank;

     rgb<= number_pixel;
  end else if (sw[1:0] == 2'b10) begin // just camera  setting
     hs <= phsync;
     vs <= pvsync;
     b <= pblank;
     rgb<= camera_pixel;

  end else if (sw[1:0] == 2'b11) begin //camera and number setting
     hs <= phsync;
     vs <= pvsync;
     b <= pblank;

     rgb<= number_pixel | camera_pixel;


  end else if  (sw[1:0] == 2'b0) begin  //camera off
     hs <= phsync;
     vs <= pvsync;
     b <= pblank;


     rgb<=0;
  end else begin
     // default: camera and sensor data displaying
     hs <= phsync;
     vs <= pvsync;
     b <= pblank;

     rgb<= number_pixel | camera_pixel;
  end
end


//From lab 3
// the following lines are required for the Nexys4 VGA circuit - do not change
assign vga_r = ~b ? rgb[11:8]: 0;
assign vga_g = ~b ? rgb[7:4] : 0;
assign vga_b = ~b ? rgb[3:0] : 0;

assign vga_hs = ~hs;
assign vga_vs = ~vs;
```

```
endmodule



module number_image (
  input vclock_in,        // 65MHz clock
  input reset_in,         // 1 to initialize module

   //from lab 3
  input [10:0] hcount_in, // horizontal index of current pixel (0..1023)
  input [9:0]  vcount_in, // vertical index of current pixel (0..767)
  input hsync_in,         // XVGA horizontal sync signal (active low)
  input vsync_in,         // XVGA vertical sync signal (active low)
  input blank_in,         // XVGA blanking (1 means output black pixel)

  output logic phsync_out,       // horizontal sync
  output logic pvsync_out,       // vertical sync
  output logic pblank_out,       // blanking
  output logic [11:0] pixel_out,  // pixel value  // r=11:8, g=7:4, b=3:0



  input [9:0] data_in

  );

   assign phsync_out = hsync_in;
  assign pvsync_out = vsync_in;
  assign pblank_out = blank_in;
  //x and y positions of numbers
  logic [10:0] num_x = 11'd700; //start numbers in center-right of screen (coords 700, 50)
  logic [9:0] num_y = 10'd50;

  logic [11:0] num_pixel_ones; //for the ones digit
  logic [11:0] num_pixel_tens; //for the tens digit
  logic [11:0] num_pixel_hundreds; //for the hundreds digit
logic       done;

logic  [13:0] counter;
logic   [3:0]  ones;
logic   [3:0]  tens;
logic  [3:0]  hundreds;
logic   [3:0]  thousands;


logic [9:0] refresh_counter = 0;

//MUST COUNT IN ORDER TO DISPLAY
always @(posedge vclock_in) begin
    if(refresh_counter>=1000) begin //refresh every thousand cycles
        counter <= 0;
        refresh_counter<=0;
        ones<=0;
        tens<=0;
        hundreds<=0;
        thousands<=0;
    end
    else if(counter == data_in)
        done <= 1;
    else begin
        counter <= counter + 1;
        refresh_counter<=refresh_counter+1;
        ones <= ones == 9 ? 0 : ones + 1;
        if(ones == 9) begin
            tens <= tens == 9 ? 0 : tens + 1;
            if(tens == 9) begin
                hundreds <= hundreds == 9 ? 0 : hundreds + 1;
                if(hundreds == 9) begin
                     thousands <= thousands + 1;
                end
            end
        end
    end
end
```

```
  picture_number number_ones(
   .pixel_clk_in(vclock_in),
   .digit_in(hundreds),
   .x_in(num_x),.hcount_in(hcount_in),
   .y_in(num_y),.vcount_in(vcount_in),

   .pixel_out(num_pixel_ones));

      picture_number number_tens(
   .pixel_clk_in(vclock_in),
   .digit_in(tens),
   .x_in(num_x + 50),.hcount_in(hcount_in),
   .y_in(num_y),.vcount_in(vcount_in),

   .pixel_out(num_pixel_tens));

    picture_number number_hundreds(
   .pixel_clk_in(vclock_in),
   .digit_in(ones),
   .x_in(num_x + 100),.hcount_in(hcount_in),
   .y_in(num_y),.vcount_in(vcount_in),

   .pixel_out(num_pixel_hundreds));

  always_comb begin
      pixel_out = num_pixel_ones | num_pixel_tens | num_pixel_hundreds;
  end

endmodule



 ////////////////////////////////////////////////////////////////////////////////
//
```

```
// Pushbutton Debounce Module (video version - 24 bits)
//
/////////////////////////////////////////////////////////////////////////////////

module debounce (input reset_in, clock_in, noisy_in,
                output logic clean_out);

  logic [19:0] count;
  logic new_input;

  always_ff @(posedge clock_in)
    if (reset_in) begin
       new_input <= noisy_in;
       clean_out <= noisy_in;
       count <= 0; end
    else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0; end
    else if (count == 1000000) clean_out <= new_input;
    else count <= count+1;


endmodule

   //from lab 3
module xvga(input vclock_in,
           input reset_in,
           output logic [10:0] hcount_out,    // pixel number on current line
           output logic [9:0] vcount_out,     // line number
           output logic vsync_out, hsync_out,
           output logic blank_out);

  parameter DISPLAY_WIDTH  = 1024;      // display width
  parameter DISPLAY_HEIGHT = 768;       // number of lines

  parameter  H_FP = 24;                 // horizontal front porch
  parameter  H_SYNC_PULSE = 136;        // horizontal sync
  parameter  H_BP = 160;                // horizontal back porch

  parameter  V_FP = 3;                  // vertical front porch
  parameter  V_SYNC_PULSE = 6;          // vertical sync
  parameter  V_BP = 29;                 // vertical back porch

  // horizontal: 1344 pixels total
  // display 1024 pixels per line
  logic hblank,vblank;
  logic hsyncon,hsyncoff,hreset,hblankon;
  assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
  assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
  assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));  // 1183
  assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP - 1)|reset_in);  //1343

  // vertical: 806 lines total
  // display 768 lines
  logic vsyncon,vsyncoff,vreset,vblankon;
  assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
  assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));  // 771
  assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE - 1));  // 777
  assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE + V_BP -
1)|reset_in); // 805

  // sync and blanking
  logic next_hblank,next_vblank;
  assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
  assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
  always_ff @(posedge vclock_in) begin
     hcount_out <= hreset ? 0 : hcount_out + 1;
     hblank <= next_hblank;
     hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

     vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) : vcount_out;
     vblank <= next_vblank;
     vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  // active low

     blank_out <= next_vblank | (next_hblank & ~hreset);
  end
endmodule
```

```
module camera_to_picture(
  input pixel_clk_in,

  input clk_50mhz,
   input [14:0] hcount_in,
  input [14:0] vcount_in,


  input [16:0] addr_in,
  input [11:0] raw_data_in,

  input double_on, //boolean of whether to double size or not

  input hsync_in,         // XVGA horizontal sync signal (active low)
  input vsync_in,         // XVGA vertical sync signal (active low)
  input blank_in,         // XVGA blanking (1 means output black pixel)

  output logic chsync_out,       // camera horizontal sync
  output logic cvsync_out,       // camera vertical sync
  output logic cblank_out,       // camera blanking


  output logic [11:0] pixel_out,
  output logic [11:0] double_pixel_out


           );



  assign chsync_out = hsync_in;
  assign cvsync_out = vsync_in;
  assign cblank_out = blank_in;



   logic [11:0] video_pixel;
   logic [16:0] addr_out;



   always_comb begin
       //for 2x resolution
       if (double_on) begin //double size interpolation
           if (vcount_in[0] == 1'b0 && hcount_in[0] ==1'b0) begin //even case
               addr_out = 320*(240-(hcount_in>>1)) + (vcount_in>>1);
           end

           else if (vcount_in[0] == 1'b1 && hcount_in[0]==1'b0) begin
               addr_out = 320*((hcount_in>>1)) + ((vcount_in-1)>>1);
           end

           else if (vcount_in[0]==1'b0 && hcount_in[0]==1'b1) begin
               addr_out = 320*(((hcount_in-1)>>1)) + (vcount_in>>1);
           end

           else if (vcount_in[0]==1'b1 && hcount_in[0]==1'b1) begin
               addr_out = 320*(((hcount_in-1)>>1)) + ((vcount_in-1)>>1);
           end
       end

       else begin //not double size
           addr_out = 320*(hcount_in) + vcount_in;
       end
   end


   //simultaneously write new data, and read current pixel according to vcount and hcount
   camera_dual_ram    cam_1(.clka(clk_50mhz), .wea(1), .addra(addr_in), .dina(raw_data_in),
                           .clkb(pixel_clk_in), .addrb(addr_out), .doutb(video_pixel) );

   always_ff @(posedge pixel_clk_in) begin


       if (~double_on) begin  //normal
```

```
           if ((hcount_in<320) && (vcount_in<240)) begin  //only allow one 320x240 image
               pixel_out<= video_pixel;
           end
           else begin // outside of border
               pixel_out<=0;
           end
       end

       //if double size
       else begin
           if ((hcount_in<640) && (vcount_in<480)) begin  //only allow one 640x320 image
               pixel_out<= video_pixel;

           end
           else begin
               pixel_out<=0;
           end
       end
   end





endmodule


module picture_number
  #(parameter WIDTH = 48,     // default picture width
              HEIGHT = 48)    // default picture height
  (input pixel_clk_in,
   input [10:0] x_in,hcount_in,
   input [9:0] y_in,vcount_in,
   input [3:0] digit_in,
    output logic [11:0] pixel_out);
  logic [14:0] image_addr;   // num of bits for 256*240 ROM
  logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

  logic [3:0] greyscale_bits;

  //each number is 48*48 = 2304
  assign image_addr = (2304*digit_in) + (hcount_in-x_in) + ((vcount_in-y_in)*WIDTH);




   blk_mem_gen_1  rom2(.clka(pixel_clk_in), .addra(image_addr), .douta(greyscale_bits));



  always_ff @ (posedge pixel_clk_in) begin
    if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
         (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))



      pixel_out <= {greyscale_bits, greyscale_bits, greyscale_bits};

      else pixel_out <= 0;
  end
endmodule
```

14 Verilog Simulation Code

// udp_pkt_receive_tb.sv

```
module udp_pkt_receive_tb;
```

```
//make logics for inputs and outputs!
logic clk;
logic rst;
logic txen;
logic [1:0] txd;
logic tx_busy;
logic [239:0] payload_in;
logic [239:0] payload_out;
logic [31:0] counter;
logic input_valid;
logic phy_rst_done;
 udp_pkt_send_test#(.POWER_UP_CYCLES(2), .PAYLOAD_BYTES(30)) send_test (
                    .clk(clk),
                    .button_reset(rst),
                    .txen(txen),
                    .txd(txd),
                    .t_payload(payload_in),
                    .input_valid(input_valid),
                    .tx_busy(tx_busy),
                    .send_port(16'd5001),
                    .phy_rst_done(phy_rst_done)
                    );


udp_pkt_receive#(.POWER_UP_CYCLES(2), .PAYLOAD_BYTES(30)) rec(.clk(clk),
                    .rxd(txd),
                    .rx_valid(txen),
                    .payload_out(payload_out),
                    .button_reset(rst),
                    .phy_rst_done(phy_rst_done));

//An always block in simulation **always** runs in the background
//this is your standard way of making a clock below:
//it says: every 5 ns, make clk be !clk
//still need to initialize clk in an initial block
always begin
    #5;  //every 5 ns switch...so period of clock is 10 ns...100 MHz clock
    clk = !clk;
end
always @(posedge clk) begin

     if(~tx_busy) begin
         payload_in <= counter;
         input_valid <= 1;
         counter <= counter + 1;
     end else begin
        input_valid <= 0;
     end
end

//initial block...this is our test simulation
initial begin
    $display("Starting Sim"); //print nice message
    clk = 0; //initialize clk (super important)
    rst = 0; //initialize rst (super important)
    counter = 0;
    payload_in = 0;
    input_valid = 0;
    phy_rst_done = 0;
    #20  //wait a little bit of time at beginning
    rst = 1; //reset system
    payload_in = 0;
    counter = 0;
    input_valid = 0;
    phy_rst_done = 0;
    #20; //hold high for a few clock cycles
    rst=0; //pull low
    counter = 0;
    phy_rst_done = 1;
    input_valid = 0;
    #200000000; //wait a little bit

    $finish;

 end
```

```
endmodule
```

// udp_pkt_send_test_tb.sv

```
module udp_pkt_send_test_tb;

   //make logics for inputs and outputs!
   logic clk;
   logic rst;
   logic txen;
   logic [1:0] txd;
   logic tx_busy;
   logic [239:0] payload_in;
   logic [31:0] counter;
   logic input_valid;
   logic phy_rst_done;
    udp_pkt_send_test#(.POWER_UP_CYCLES(2)) send_test (
                       .clk(clk),
                       .button_reset(rst),
                       .txen(txen),
                       .txd(txd),
                       .t_payload(payload_in),
                       .input_valid(input_valid),
                       .tx_busy(tx_busy),
                       .send_port(16'd5001),
                       .phy_rst_done(phy_rst_done)
                       );

   //An always block in simulation **always** runs in the background
   //this is your standard way of making a clock below:
   //it says: every 5 ns, make clk be !clk
   //still need to initialize clk in an initial block
   always begin
       #5;  //every 5 ns switch...so period of clock is 10 ns...100 MHz clock
       clk = !clk;
   end
   always @(posedge clk) begin
```

```
        if(~tx_busy) begin
            payload_in <= counter;
            input_valid <= 1;
            counter <= counter + 1;
        end else begin
           input_valid <= 0;
        end
   end

   //initial block...this is our test simulation
   initial begin
       $display("Starting Sim"); //print nice message
       clk = 0; //initialize clk (super important)
       rst = 0; //initialize rst (super important)
       counter = 0;
       payload_in = 0;
       input_valid = 0;
       phy_rst_done = 0;
       #20  //wait a little bit of time at beginning
       rst = 1; //reset system
       payload_in = 0;
       counter = 0;
       input_valid = 0;
       phy_rst_done = 0;
       #20; //hold high for a few clock cycles
       rst=0; //pull low
       counter = 0;
       phy_rst_done = 1;
       input_valid = 0;
       #200000000; //wait a little bit
       $finish;
    end
endmodule
```

15 Python Server Code

// RobotRX_ControlTX.py

```
import socket
import time
"""
Establishes a connection from the two PCs
Using the Server as a middle man
1. Get a message from RX and TX in any order
2. Send message to TX
3. Send message to RX
5. Go in to a loop where TX -> SERVER -> RX
6. End the connection if you get a STOP message
"""

DATA_SIZE = 548
RESEND = 1
# Server address and port
SERVER_IP = "45.79.176.240"
SERVER_PORT = 5004

# TX IP address and port.
TX_IP = None
TX_PORT = 0

# RX IP address and port.
RX_IP = None
RX_PORT = 0

# Socket Setup
sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.bind((SERVER_IP,SERVER_PORT))
sock.setblocking(True)

STOP_MESSAGE = int.from_bytes(bytes("STOP","ascii"), "big")

# STEP 1: Get message from RX and TX and in effect their addresses
print("Starting Server 1")
while (TX_IP is None) or (RX_IP is None):
   data, addr = sock.recvfrom(2)
   if data == bytes("TX","ascii"):
       (TX_IP,TX_PORT) = addr
       # STEP 2: Send message to TX
       for i in range(RESEND + 1):
           sock.sendto(bytes("Server to PC","ascii"), (TX_IP,TX_PORT))
   elif data == bytes("RX","ascii"):
       (RX_IP,RX_PORT) = addr
       # STEP 3: Send message to RX
       for i in range(RESEND + 1):
           sock.sendto(bytes("Server to PC","ascii"), (RX_IP,RX_PORT))

print("CONTROL -> ROBOT init")

# # STEP 4 & STEP 5: Go into an infinite loop and stop is you get the stop message
while True:
   data = sock.recv(DATA_SIZE)
   sock.sendto(data, (RX_IP,RX_PORT))
   if(int.from_bytes(data,"big") & 0xFFFFFFFF == STOP_MESSAGE):
       break
   elif data == bytes("TX","ascii"):
       (TX_IP,TX_PORT) = addr
       for j in range(RESEND + 1):
           sock.sendto(bytes("Server to PC","ascii"), (TX_IP,TX_PORT))
   elif data == bytes("RX","ascii"):
       (RX_IP,RX_PORT) = addr
       for j in range(RESEND + 1):
           sock.sendto(bytes("Server to PC","ascii"), (RX_IP,RX_PORT))
print("Server 1 Closed")
sock.close()
```

// RobotTX_ControlRX.py

```
import socket
import time
"""
Establishes a connection from the two PCs
Using the Server as a middle man
1. Get a message from RX and TX in any order
2. Send message to TX
3. Send message to RX
```

```
5. Go in to a loop where TX -> SERVER -> RX
6. End the connection if you get a STOP message
"""
RESEND = 1
DATA_SIZE = 548

# Server address and port
SERVER_IP = "45.79.176.240"
SERVER_PORT = 5010

# TX IP address and port.
TX_IP = None
TX_PORT = 0

# RX IP address and port.
RX_IP = None
RX_PORT = 0

# Socket Setup
sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.bind((SERVER_IP,SERVER_PORT))
sock.setblocking(True)

STOP_MESSAGE = int.from_bytes(bytes("STOP","ascii"), "big")

# STEP 1: Get message from RX and TX and in effect their addresses
print("Starting Server 2")
while (TX_IP is None) or (RX_IP is None):
   data, addr = sock.recvfrom(2)
   if data == bytes("TX","ascii"):
       (TX_IP,TX_PORT) = addr
       # STEP 2: Send message to TX
       for i in range(RESEND + 1):
           sock.sendto(bytes("Server to PC","ascii"), (TX_IP,TX_PORT))
   elif data == bytes("RX","ascii"):
       (RX_IP,RX_PORT) = addr
       # STEP 3: Send message to RX
```

```
        for i in range(RESEND + 1):
            sock.sendto(bytes("Server to PC","ascii"), (RX_IP,RX_PORT))

print("ROBOT -> CONTROL init")

# STEP 4 & STEP 5: Go into an infinite loop and stop is you get the stop message
while True:
    data = sock.recv(DATA_SIZE)
    sock.sendto(data, (RX_IP,RX_PORT))
    if(int.from_bytes(data,"big") & 0xFFFFFFFF == STOP_MESSAGE):
        break
    elif data == bytes("TX","ascii"):
        (TX_IP,TX_PORT) = addr
        for j in range(RESEND + 1):
            sock.sendto(bytes("Server to PC","ascii"), (TX_IP,TX_PORT))
    elif data == bytes("RX","ascii"):
        (RX_IP,RX_PORT) = addr
        for j in range(RESEND + 1):
            sock.sendto(bytes("Server to PC","ascii"), (RX_IP,RX_PORT))
print("Server 2 Closed")
sock.close()
```

16 Python Application Code

16.1 Robot

// reception.py

```
import socket
import time

"""
Uses WIFI on Mac OSX 11.0.1
Establishes a connection from the FPGA to the Server
Using the PC as a middle man
1. Get a message from the FPGA
2. Send message to the Server
```

```
3. Get message from Server
4. Send a message to the FPGA
5. Go in to a loop where Server -> PC -> FPGA
6. End the connection if you get a STOP message
Note: Current Uncommented configuration is using a WIFI for the FPGA and
connecting an Ethernet to the MacBook
"""
RESEND = 1
DATA_SIZE = 548
# PC IP address and port
RX_PC_IP = ""
RX_PC_PORT = 5003

# FPGA IP address and port.
# FPGA_IP = "224.0.0.246" # WIFI
FPGA_IP = '169.254.255.255' #Ethernet
FPGA_PORT = 5001

# Server IP address and port
RX_SERVER_IP = "45.79.176.240"
RX_SERVER_PORT = 5004

# Setup Socket
sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.bind((RX_PC_IP,RX_PC_PORT))
sock.setblocking(True)

# Verification Codes
FPGA_TO_PC = int.from_bytes(bytes("FPGA RX INIT","ascii"), "big")
PC_TO_FPGA = bytes("PC to FPGA RX CONNECTED","ascii")
RX = bytes("RX","ascii")
SERVER_TO_PC = int.from_bytes(bytes("Server to PC","ascii"), "big")
STOP_MESSAGE = int.from_bytes(bytes("STOP","ascii"), "big")

print("Starting RX")
# Step 1: Get message from the FPGA
# print("Waiting for FPGA...")
```

```
while True:
    data, addr = sock.recvfrom(DATA_SIZE)
    if(int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == FPGA_TO_PC):
        break

# Step 2: Send message to Server
# print("Sending to Server...")
for i in range(RESEND + 1):
    sock.sendto(RX, (RX_SERVER_IP,RX_SERVER_PORT))

# Step 3: Get message from Server
# print("Waiting for Server...")
while True:
    data, addr = sock.recvfrom(DATA_SIZE)
    if(int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == SERVER_TO_PC):
        break

# Step 4: Send message to FPGA
# print("Sending to FPGA...")
for i in range(RESEND + 1):
    sock.sendto(PC_TO_FPGA, (FPGA_IP,FPGA_PORT))

# # Step 5 & Step 6:
# # Connection is now established so go into an infinite loop
# # stop if you get the STOP message
print("RX connected")
while True:
    data = sock.recv(DATA_SIZE)
    sock.sendto(data, (FPGA_IP,FPGA_PORT))
    if(int.from_bytes(data,"big") & 0xFFFFFFFF == STOP_MESSAGE):
        for j in range(RESEND + 1):
            sock.sendto(data, (FPGA_IP,FPGA_PORT))
        break
    elif (int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == FPGA_TO_PC):
        for j in range(RESEND + 1):
            sock.sendto(PC_TO_FPGA, (FPGA_IP,FPGA_PORT))
print("RX Closed")
sock.close()
```

// transmission.py

```
import socket
import time

"""
Uses WIFI on Mac OSX Big Sur 11.0.1
Establishes a connection from the FPGA to the Server
Using the PC as a middle man
1. Get a message from the FPGA
2. Send message to the Server
3. Get message from Server
4. Send a message to the FPGA
5. Go in to a loop where FPGA -> PC -> SERVER
6. End the connection if you get a STOP message
Note: Current Uncommented configuration is using a WIFI for the FPGA and
connecting an Ethernet to the MacBook
"""
RESEND = 2
DATA_SIZE = 548

# PC IP address and port
TX_PC_IP = ""
TX_PC_PORT = 1024

# FPGA IP address and port.
# FPGA_IP = "224.0.0.246" # WIFI
FPGA_IP = '169.254.255.255' #Ethernet
FPGA_PORT = 5001

# Server IP address and port
TX_SERVER_IP = "45.79.176.240"
TX_SERVER_PORT = 5010
# TX_SERVER_PORT = 5004 # Communicate with self

# Verification Codes
FPGA_TO_PC = int.from_bytes(bytes("FPGA TX INIT","ascii"), "big")
PC_TO_FPGA = bytes("PC to FPGA TX CONNECTED","ascii")
TX = bytes("TX","ascii")
```

```
STOP_MESSAGE = bytes("STOP","ascii")
SERVER_TO_PC = int.from_bytes(bytes("Server to PC","ascii"), "big")
STOP_MESSAGE = int.from_bytes(bytes("STOP","ascii"), "big")

# Setup Socket
sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.bind((TX_PC_IP,TX_PC_PORT))
sock.setblocking(True)

print("Starting TX")
# Step 1: Get message from the FPGA
# print("Waiting for FPGA...")
while True:
   data, addr = sock.recvfrom(DATA_SIZE)
   if(int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == FPGA_TO_PC):
       break

# Step 2: Send message to Server
# print("Sending to Server...")
for i in range(RESEND + 1):
   sock.sendto(TX, (TX_SERVER_IP,TX_SERVER_PORT))

# Step 3: Get message from Server
# print("Waiting from Server...")
while True:
   data, addr = sock.recvfrom(DATA_SIZE)
   if(int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == SERVER_TO_PC):
       break

# Step 4: Send message to FPGA
# print("Sending to FPGA...")
for i in range(RESEND + 1):
   sock.sendto(PC_TO_FPGA, (FPGA_IP,FPGA_PORT))

# Step 5 & Step 6:
# Connection is now established so go into an infinite loop
# stop if you get the STOP message
```

```python
print("TX connected")
while True:
    data = sock.recv(DATA_SIZE)
    sock.sendto(data, (TX_SERVER_IP,TX_SERVER_PORT))
    if(int.from_bytes(data,"big") & 0xFFFFFFFF == STOP_MESSAGE):
        for j in range(RESEND + 1):
            sock.sendto(data, (TX_SERVER_IP,TX_SERVER_PORT))
        break
    elif (int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == FPGA_TO_PC):
        for j in range(RESEND + 1):
            sock.sendto(PC_TO_FPGA, (FPGA_IP,FPGA_PORT))
print("TX Closed")
sock.close()
```

16.2 Controller

// reception.py

```python
import socket
import time

"""
Uses WIFI on Mac OSX Catalina Version 10.15.7
Establishes a connection from the FPGA to the Server
Using the PC as a middle man
1. Get a message from the FPGA
2. Send message to the Server
3. Get message from Server
4. Send a message to the FPGA
5. Go in to a loop where Server -> PC -> FPGA
6. End the connection if you get a STOP message
"""
RESEND = 1
DATA_SIZE = 548
# PC IP address and port
RX_PC_IP = ""
RX_PC_PORT = 5003

# FPGA IP address and port.
```

```
FPGA_IP = '169.254.255.255' #Ethernet
FPGA_PORT = 5001

# Server IP address and port
RX_SERVER_IP = "45.79.176.240"
RX_SERVER_PORT = 5010

# Setup Socket
sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.bind((RX_PC_IP,RX_PC_PORT))
sock.setblocking(True)

# Verification Codes
FPGA_TO_PC = int.from_bytes(bytes("FPGA RX INIT","ascii"), "big")
PC_TO_FPGA = bytes("PC to FPGA RX CONNECTED","ascii")
RX = bytes("RX","ascii")
SERVER_TO_PC = int.from_bytes(bytes("Server to PC","ascii"), "big")
STOP_MESSAGE = int.from_bytes(bytes("STOP","ascii"), "big")

print("Starting RX")
# Step 1: Get message from the FPGA
print("Waiting for FPGA...")
while True:
   data, addr = sock.recvfrom(DATA_SIZE)
   if(int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == FPGA_TO_PC):
       print(addr)
       break

# Step 2: Send message to Server
print("Sending to Server...")
for i in range(RESEND + 1):
   sock.sendto(RX, (RX_SERVER_IP,RX_SERVER_PORT))

# Step 3: Get message from Server
print("Waiting from Server...")
while True:
   data, addr = sock.recvfrom(DATA_SIZE)
```

```
    if(int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == SERVER_TO_PC):
        print(addr)
        break

# Step 4: Send message to FPGA
print("Sending to FPGA...")
for i in range(RESEND + 1):
    sock.sendto(PC_TO_FPGA, (FPGA_IP,FPGA_PORT))

# Step 5 & Step 6:
# Connection is now established so go into an infinite loop
# stop if you get the STOP message
while True:
    data = sock.recv(DATA_SIZE)
    sock.sendto(data, (FPGA_IP,FPGA_PORT))
    if(int.from_bytes(data,"big") & 0xFFFFFFFF == STOP_MESSAGE):
        break
    elif (int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == FPGA_TO_PC):
        print("Sending to FPGA...")
        for j in range(RESEND + 1):
            sock.sendto(PC_TO_FPGA, (FPGA_IP,FPGA_PORT))
print("RX Closed")
sock.close()
```

// transmission.py

```
import socket
import time

"""
Uses WIFI on Mac OSX Catalina Version 10.15.7
Establishes a connection from the FPGA to the Server
Using the PC as a middle man
1. Get a message from the FPGA
2. Send message to the Server
3. Get message from Server
4. Send a message to the FPGA
5. Go in to a loop where FPGA -> PC -> SERVER
```

```
6. End the connection if you get a STOP message
"""
RESEND = 1
DATA_SIZE = 548

# PC IP address and port
TX_PC_IP = ""
TX_PC_PORT = 1024

# FPGA IP address and port.
FPGA_IP = '169.254.255.255' #Ethernet
FPGA_PORT = 5001

# Server IP address and port
TX_SERVER_IP = "45.79.176.240"
TX_SERVER_PORT = 5004
# TX_SERVER_PORT = 5010 # Communicate with self

# Verification Codes
FPGA_TO_PC = int.from_bytes(bytes("FPGA TX INIT","ascii"), "big")
PC_TO_FPGA = bytes("PC to FPGA TX CONNECTED","ascii")
TX = bytes("TX","ascii")
STOP_MESSAGE = bytes("STOP","ascii")
SERVER_TO_PC = int.from_bytes(bytes("Server to PC","ascii"), "big")
STOP_MESSAGE = int.from_bytes(bytes("STOP","ascii"), "big")

# Setup Socket
sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.bind((TX_PC_IP,TX_PC_PORT))
sock.setblocking(True)

print("Starting TX")
# Step 1: Get message from the FPGA
print("Waiting for FPGA...")
while True:
    data, addr = sock.recvfrom(DATA_SIZE)
    if(int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == FPGA_TO_PC):
        print(addr)
        break

# Step 2: Send message to Server
print("Sending to Server...")
for i in range(RESEND + 1):
    sock.sendto(TX, (TX_SERVER_IP,TX_SERVER_PORT))

# Step 3: Get message from Server
print("Waiting from Server...")
while True:
    data, addr = sock.recvfrom(DATA_SIZE)
    if(int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == SERVER_TO_PC):
        print(addr)
        break

# Step 4: Send message to FPGA
print("Sending to FPGA...")
for i in range(RESEND + 1):
    sock.sendto(PC_TO_FPGA, (FPGA_IP,FPGA_PORT))

# Step 5 & Step 6:
# Connection is now established so go into an infinite loop
# stop if you get the STOP message
while True:
    data = sock.recv(DATA_SIZE)
    sock.sendto(data, (TX_SERVER_IP,TX_SERVER_PORT))
    if(int.from_bytes(data,"big") & 0xFFFFFFFF == STOP_MESSAGE):
        break
    elif (int.from_bytes(data, "big") & 0xFFFFFFFFFFFFFFFFFFFFFFFF == FPGA_TO_PC):
        print("Sending to FPGA...")
        for j in range(RESEND + 1):
            sock.sendto(PC_TO_FPGA, (FPGA_IP,FPGA_PORT))
print("TX Closed")
sock.close()
```