Amanda Ke
Allan Ko
6.111 Final Report
January 22, 2017

LASERNET

Table of Contents

1. Introduction 1

2. System Overview 2
- 2.1. Software Architecture 2
- 2.2. Hardware Components 4

3. Detailed Specifications 5
- 3.1. Protocol Layer 5
  - State machine (mainfsm.v) 5
  - Packet construction (makepacket.v) 10
  - Packet verification (receivepacket.v) 12
- 3.2. Physical Layer 13
  - Laser modulation (serial_tx.v) 15
  - Photodiode input (serial_rx.v) 15
- 3.3. User Interface 17
  - Keyboard input (keyboardinput.v) 17
  - VGA monitor display (screenlayout.v) 20

4. Development Notes 21

5. Conclusion 23

6. Appendix 23
- 6.1. Video 23
- 6.2. Verilog 23

1. Introduction

Free-space optical (FSO) communication systems transmit wireless data using visible-frequency electromagnetic waves propagating in free space. FSO systems using lasers as the communication medium are of particular interest, as they theoretically enable point-to-point long-range communication links with data rates comparable to (if not better than) those achievable by radio broadcast or fiber optic cables, without the infrastructural cost of wired connections. Laser-based communication systems therefore have a wide range of theoretical applications, such as providing network connectivity in emergency situations or improving the data rates of deep space transmissions.

Inspired by laser communications research underway at NASA and Facebook's Connectivity Lab, this report details the development of LASERNET, an FSO communication system implemented with FPGAs and off-the-shelf lasers. LASERNET robustly transmits data from one FPGA to another over a laser link by using a simplified version of Transmission Control Protocol (TCP). LASERNET also includes a basic user interface for entering new text messages for transmission, and for displaying messages received from the other FPGA.

Another networking aspect we wanted to explore with this project was Software-Defined Networking, or SDN. The goal of SDN is to programmatically control, change, and manage network behavior dynamically via open interfaces and abstraction of lower-level functionality. Traditional networks have static architectures, which makes adapting to the dynamic needs of modern computing environments (such as data centers) difficult. Writing a communications protocol in Verilog gives access to the forwarding plane to a router over the network, allowing a network administrator more control over the network without having to change individual switches.

Both nodes in our system can transmit and receive data, so they operate with the same software and hardware. We describe the general software architecture, hardware components, and block diagram of the system in Section 2. We provide more detailed specifications of LASERNET's submodules in Section 3. Finally in Section 4, we reflect on the development process and suggest possible enhancements to our system.

2. System Overview

2.1. Software Architecture

The system is managed by a finite state machine that establishes TCP-like connections, tracks which packets have been acknowledged, retransmits packets when necessary, processes data for user input/output, and closes the TCP connection when the transmission is complete. The TCP connection transmits packets with a go-back-N automatic repeat request protocol, ensuring that data arrives complete and in order. Our implementation is detailed in Section 3.1.

The main state machine interfaces with submodules for message transmission and reception. To transmit messages, we implement submodules for constructing TCP packet headers and modulating the laser. To receive messages, we implement submodules for detecting the incoming signal with a photodiode and verifying the incoming packets' TCP checksums. For a simple user interface and application layer, we implement submodules for inputting new messages on a keyboard, and for displaying the incoming and outgoing messages on a monitor. The entire system is clocked on a single 65 Mhz clock domain for simplicity, as XVGA output (1024 by 768 resolution at 60hz) requires a 65 Mhz clock. The 65 Mhz clock signal is generated with Vivado's clock wizard.

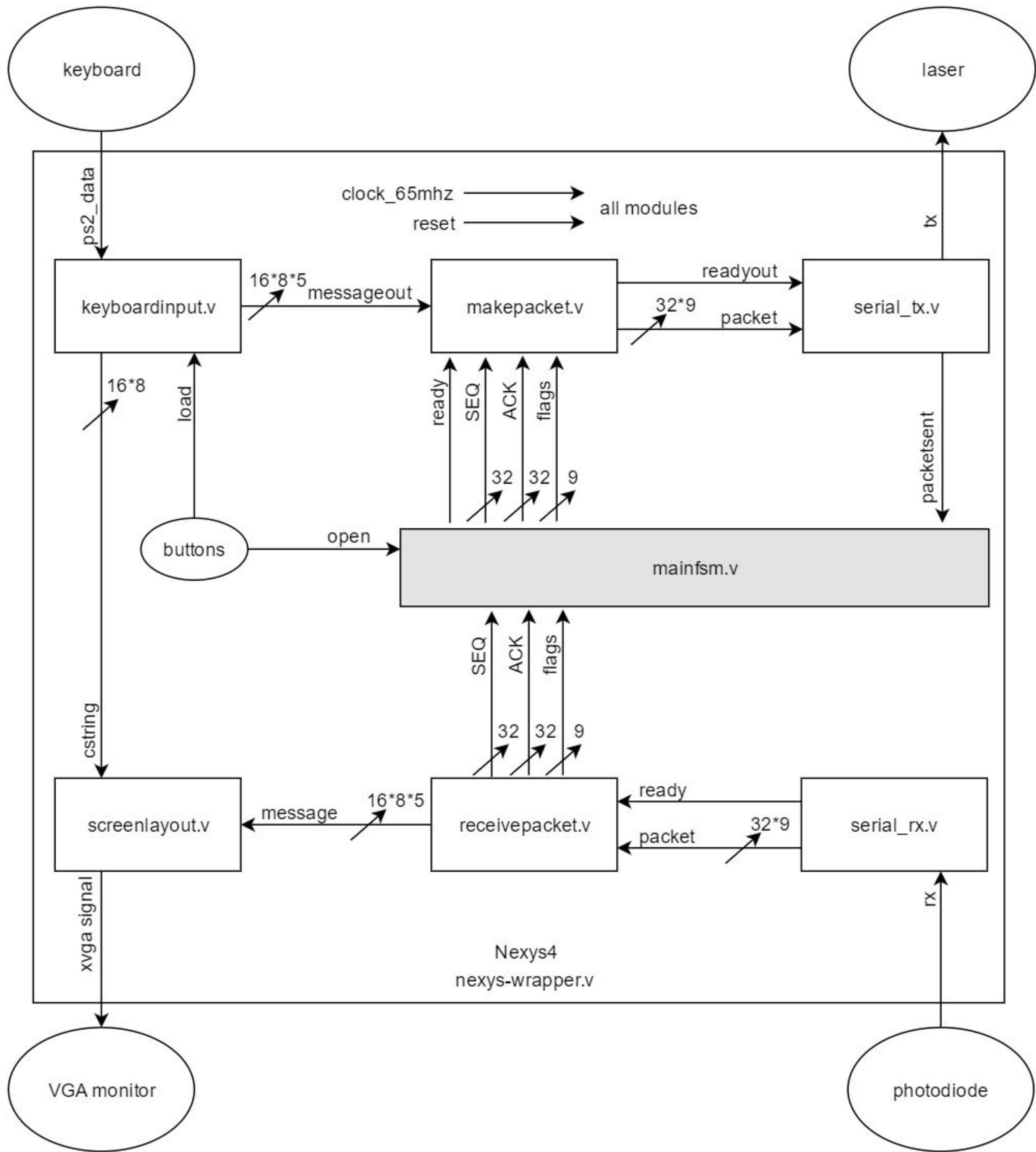

Figure 2.1.1 - Block diagram

The block diagram of the system is shown in Figure 2.1.1. It includes all the major wires and buses between submodules (represented by rectangles) as well as interfaces to hardware peripherals (represented by ovals). Notice also the small oval included within the Nexys4 block, representing hardware buttons on the Nexys4 board. Logic for message input and transmission flows from left to right on the top half of the diagram,

from the keyboard to the laser output; logic for message reception flows from right to left on the bottom half of the diagram, from the photodiode to the VGA monitor. Full specifications for each submodule are described in Section 3.

2.2. Hardware Components

Each node of LASERNET is implemented on the Digilent Nexys4 DDR development board, powered by the Xilinx Artix-7 FPGA. The laser data link uses an off-the shelf 650nm laser for transmission, a 650nm photodiode for reception, and an external 5V power supply. A basic user interface is provided through a USB keyboard and a VGA monitor.

The transmitting laser used was a 5mW Red 650nm laser dot diode, commonly found in laser pens and pointers. The digital input from JA[0] is fed into an inverting comparator (LM311). The output of the comparator is tied to the negative lead of the laser while the positive lead is connected to the 5V power supply (Figure 2.2.1).

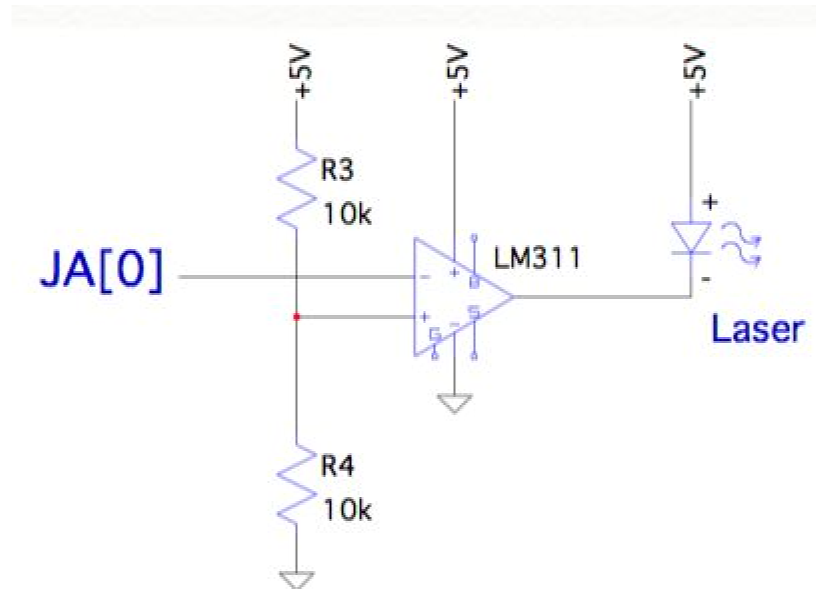

Figure 2.2.1 - Circuit diagram for laser transmitter.

3. Detailed Specifications

3.1. Protocol Layer

Developed by Allan

State machine (mainfsm.v)

The data link is supervised by *mainfsm.v*, the state machine which coordinates all the other modules of the system. The primary function of this module is to coordinate the three stages of the TCP connection: (1) the three-way handshake to initiate a connection, (2) data transmission with a go-back-N automatic repeat request protocol to detect dropped packets, and (3) the four-way handshake to terminate the connection. The state machine monitors the flags, sequence number, and acknowledgement number from the header of the most recent packet received through *receivepacket.v*. It uses that information to determine if a new packet needs to be transmitted, and if so, which packet to transmit.

The module has the following inputs and outputs:

input clk: system clock
input reset: system reset
input open: button for initiating a TCP handshake (BTNC in our implementation)
input packetsent: wired to *serial_tx.v*, goes high when a packet finishes transmitting
input [31:0] ISN: initial SEQ number. Set to 0 in our implementation.
input [31:0] SNmax: number of data packets to send. Set to 5 in our implementation.
input [15:0] window: window size for go-back-N. Set to 3 in our implementation.

input readyin: input ready line from *receivepacket.v*. Unused in our implementation.
input [31:0] ACKin: ACK number of last packet from *receivepacket.v*
input [31:0] SEQin: SEQ number of last packet from *receivepacket.v*
input [8:0] flagsin: flag bits from last packet from *receivepacket.v*

output readyout: output ready to *makepacket.v*. Goes high to trigger new transmission.
output [31:0] ACKout: ACK number of next packet, for *makepacket.v*
output [31:0] SEQout: SEQ number of next packet, for *makepacket.v*
output [8:0] flagsout: flag bits for next packet, for *makepacket.v*

output [3:0] statedisplay: the current state of the state machine. Useful for debugging.

The official TCP specification is described by RFC 793, which sketches the general state diagram of a TCP connection on page 23. The implementation we use for LASERNET has eight states, summarized in Figure 3.1.1.

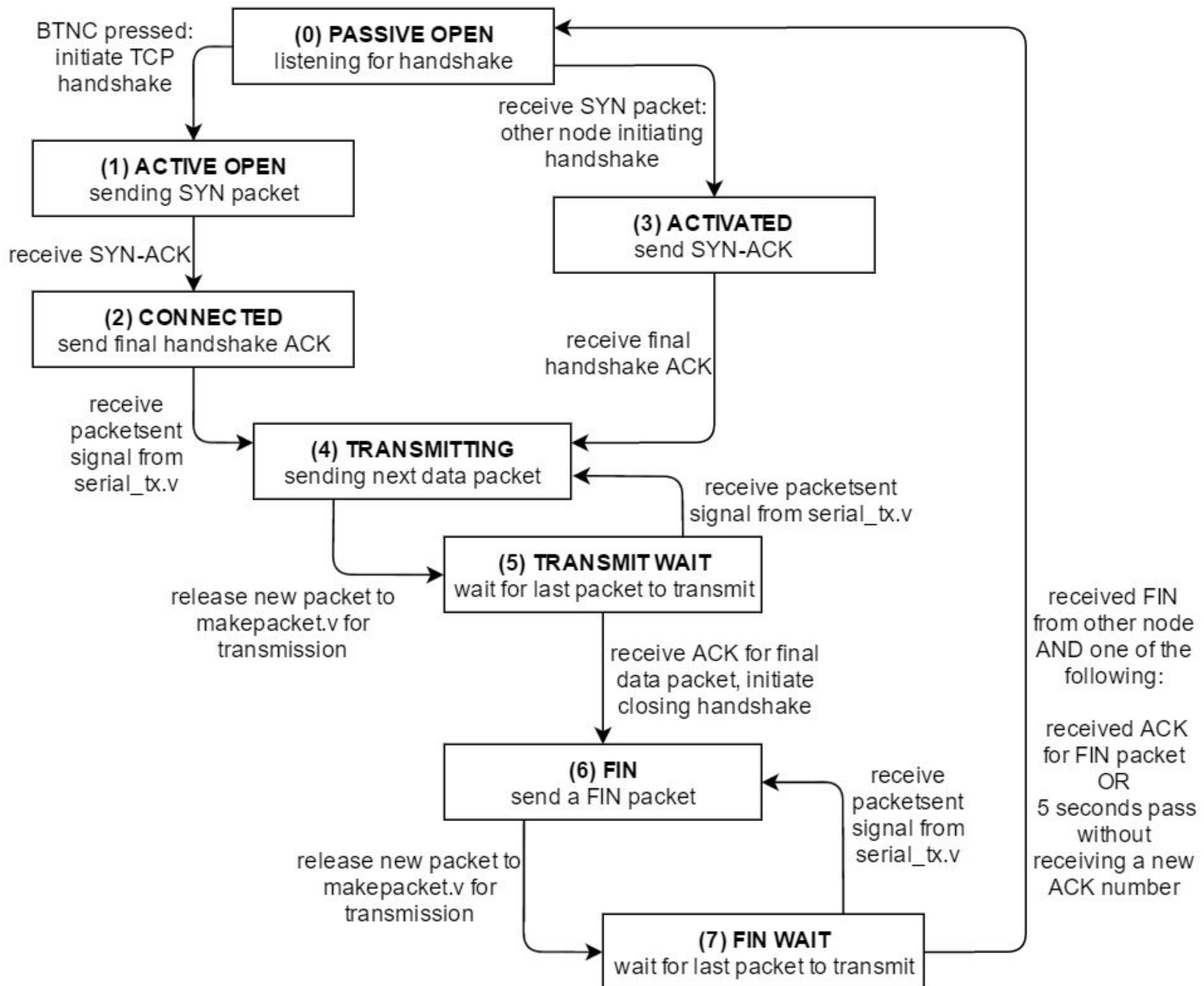

Figure 3.1.1 - State transition diagram for TCP connection

State 0: PASSIVE OPEN is the default state of the system at power-on. The node is passively listening for incoming packets from the other node. If the center button (BTNC) is pressed by the user, the FPGA will transmit a SYN packet, attempting to initiate a TCP handshake and open a connection with the other node (State 1). If the FPGA detects a SYN packet, then the other node is attempting to open a TCP connection, and the FPGA will send a SYN-ACK packet to continue the handshake (State 3).

State 1: ACTIVE OPEN gets triggered when BTNC is depressed to initiate a TCP handshake. The FPGA sends a SYN packet with an initial sequence number (ISN), and waits to receive a SYN-ACK packet from the other node. The SYN-ACK packet should have an acknowledgement number of ISN+1 (by convention, a packet numbered with a sequence number N is acknowledged by an acknowledgement number N+1). Once the state machine detects a SYN-ACK packet, it transitions to State 2 to complete the TCP handshake. Notice that if no SYN-ACK packet is received, the system hangs and the connection attempt fails. In our implementation, we use ISN = 0 for both systems.

State 2: CONNECTED is entered from State 1 when a SYN-ACK packet is received. The FPGA sends an ACK packet to acknowledge the other node's SYN. Once the packet is sent (indicated by the packetsent signal from the laser modulator *serial_tx.v* going high for one cycle) the state machine enters State 4 and begins transmitting data.

State 3: ACTIVATED is entered from State 0 if the other node is the one initiating the TCP handshake. The state machine sends a SYN-ACK in response, indicating its ISN of 0 and acknowledging the received SYN packet with an acknowledgement number of ACK = 1. It then waits to receive the last acknowledgement packet of the handshake, which should have an ACK of 1. Once that acknowledgement is received, the system enters State 4 and begins transmitting data.

State 4: TRANSMITTING is the state in which a new packet is transmitted. Upon entry, the state machine samples the current values of ACKin, SEQin, which represent the ACK and SEQ numbers from the header of last received packet. Based on the ACK and SEQ, the state machine decides what packet it should direct *makepacket.v* to construct, according to the go-back-N protocol. In our implementation we set the go-back-N window to N = 3.

According to go-back-N, if the last acknowledged packet was A, the node will repeatedly transmit every packet up to A + N. For instance, say our FPGA receives a packet with an acknowledgment number of 3 (which means that packet number 2 has been received by the other node). If we use a go-back-N protocol with a window size of 3, the FPGA will transmit packet 3, then 4, then 5, then 3 again, and so on, until it receives a new acknowledgment number. Notice how this means that not every packet needs to be individually acknowledged for the window to update. If the FPGA receives a new acknowledgement number of A = 6 (perhaps because acknowledgments 4 and 5 got dropped in transmission), it will immediately start transmitting packets 6, 7, and 8. This works because on the receiving side, packets are only accepted and acknowledged f they are received in order (see *receivepackets.v*).

If the window of transmission A + N is greater than the maximum sequence number of data available for transmission (call it SNmax), the protocol will simply transmit every available packet from A + 1 to SNmax.

While in State 4, the state machine calculates the outgoing ACK and SEQ numbers based on the current values of ACKin and SEQin. The outgoing SEQ is decided according to the go-back-N protocol described above, and the outgoing ACK is simply SEQin + 1. At the end of the clock cycle, the readyout signal goes high and *makepacket.v* constructs the appropriate packet and header for ACKout and SEQout. The state machine then immediately transitions to State 5.

Upon entry into State 4, the FPGA also checks the incoming flags to see if the FIN flag has gone high, indicating that the other node has sent a FIN packet. If it has, the FPGA

toggles an internal wire FINreceived from 0 to 1. This is important for the closing handshake terminating the connection.

State 5: TRANSMIT WAIT is immediately entered after a new packet is released to *makepacket.v*. Adding this state allows us to regulate the state machine, preventing it from trying to send a new packet every clock cycle (which would overload the laser modulator). Instead, the state machine waits for the packetsent signal from *serial_tx.v* to go high, indicating that a packet has finished transmitting. When the packetsent signal is detected, the state machine reenters State 4 to calculate and transmit the next packet. If the last available data packet has been transmitted, the state machine enters State 6 to initiate the four-way handshake for terminating the TCP connection.

State 6: FIN is entered when all the available data has been transmitted and acknowledged. However, this doesn't necessarily mean that the other node has finished transmitting data; so, this node will still have to continue transmitting packet acknowledgments as it continues receiving data. So, State 6 operates almost identically to State 4. Upon entry, it checks the current ACKin, SEQin, and flagsin, and updates the outgoing ACK (ACKout) to SEQin + 1. The outgoing SEQ (SEQout) stays constant at SNmax + 1, since we are now transmitting the FIN control packet repeatedly. After a packet is released to *makepacket.v*, the state machine immediately switches to State 7 to wait for the packet to send. The exception to this is if upon entry, state machine detects that the FIN flag has been received, in which case the FPGA releases a final acknowledgment packet and reenters State 0, closing the connection.

During this state, the state machine also checks if the ACK number has changed since the last time it was in state 6. If the ACK number has changed, it resets a five-second timeout counter to 0.

State 7: FIN WAIT is entered every time a packet is released from State 6. Like in State 5, it simply exists to regulate outgoing packets and give *serial_tx.v* time to fully transmit the previous packet before a new packet is released. When packetsent goes high, indicating that the last packet has finished transmitting, the state machine reenters State 6 to transmit a new FIN packet. The exception to this is if the five-second timeout counter reaches five seconds (or 325 000 000 clock cycles at 65 Mhz), indicating that five seconds have passed without receiving a new ACK. If the timeout counter completes, the state machine times out and returns to State 0.

In our implementation, we send SNmax = 5 data packets with a go-back-N protocol window of 3 and an initial sequence number of 0. SEQ = 0 corresponds to the initial TCP handshake (States 1 through 3). SEQ = 1 through SEQ = SNmax = 5 correspond to the five packets of data. SEQ = SNmax + 1 = 6 corresponds to the FIN packet.

Packet construction (makepacket.v)

This submodule is directed by *mainfsm.v* to construct outgoing packets. When the ready line goes high, it constructs a header, retrieves the corresponding data if necessary, calculates the TCP checksum, and forwards the complete packet to *serial_tx.v*.

This module has the following inputs and outputs:

input clk: system clock
input reset: system reset
input [31:0] ISN: initial sequence number. Set to 0 in our implementation.

input readyin: pulled high by *mainfsm.v* to trigger a nex packet transmission
input [15:0] window: window size for go-back-N. Set to 3 in our implementation.
input [31:0] seq: SEQ number of next packet transmission from *mainfsm.v*
input [31:0] ack: ACK number of next packet transmission from *mainfsm.v*
input [8:0] flags: flag bits of next packet transmission from *mainfsm.v*
input [16*8*5 - 1:0] message: full outgoing message (five 128-bit blocks of data)

output [32*9 - 1:0] packet: full outgoing packet (nine 32-bit octets)
output readyout: goes high to signal *serial_tx.v* that a new packet is ready to send

Every time the readyin line goes high, this module assembles a TCP header based on the current values of inputs seq, ack, flags, and window. If the flags indicate that this is a data packet (that is, the ACK flag is high and the SYN and FIN flags are low), then the module loads the 128-bit message block corresponding to the current sequence number into the four data rows of the packet. In other words, if seq = 1, the first 128 bits of message are loaded into the data rows; if seq = 2, the second 128 bits; and so on. If this is a control packet (SYN or FIN flags are high), the module leaves the four data rows of the packet empty, and transmits all zeros for data. Finally, the module calculates the TCP checksum, completing the packet and pulling readyout high for one cycle to indicate to *serial_tx.v* that a new packet is ready for transmission.

Each packet of data contains 16 ASCII characters of 8 bits each, or 128 bits of data. Adding in the five header rows (5*32 = 160 bits), each packet has a total of 288 bits. Each FPGA transmits five packets to relay an 80-character text message (or 16*8*5 = 640 bits) to the other FPGA. This is obviously inefficient; more than half of the data being transmitted represents packet headers and not data. But our implementation is sufficient to demonstrate a robust data link, and it is straightforward to modify this system to transmit more than four data rows per packet (as well as more than five

packets overall). The general form of the packet is summarized in Figure 3.1.2 and follows the general form of the TCP header specified in RFC 793 on page 15.

<table>
<tr><th colspan="2">source port (16 bits)</th><th>destination port (16 bits)</th></tr>
<tr><td colspan="3">sequence number (32 bits)</td></tr>
<tr><td colspan="3">acknowledgment number (32 bits)</td></tr>
<tr><td>zeroes</td><td>flags (9 bits)</td><td>go-back-N window size (16 bits)</td></tr>
<tr><td colspan="2">checksum (16 bits)</td><td>zeroes</td></tr>
<tr><td colspan="3">data (32 bits)</td></tr>
<tr><td colspan="3">data (32 bits)</td></tr>
<tr><td colspan="3">data (32 bits)</td></tr>
<tr><td colspan="3">data (32 bits)</td></tr>
</table>

Figure 3.1.2 - TCP packet. Each row corresponds to one 32-bit octet.

The source and destination port are used by TCP to specify which application on a node is making the TCP connection. This is usually used by more sophisticated computers that run multiple applications, all of which may wish to access a network. In our application, there is only one "application" on each node, so the source and destination port numbers are set to 0.

The sequence number and acknowledgement number are 32 bits each and correspond to the SEQ and ACK number of the current packet being sent. In our implementation we only have five data packets, so SEQ only goes up to 6 and ACK only goes up to 7. However, we include the full 32 bits as an available resource should we ever want to modify the system to transmit longer data files.

The nine flag bits are used for a number of functions. Some of them are are used for connection management, packet prioritization, or congestion management features; we leave these equal to 0 for all our packets. The three flag bits we do use are the ACK, SYN, and FIN flags (corresponding to bits 0, 1, and 4, indexing from the LSB). The SYN flag is used during the TCP handshake to indicate an attempt to open a new connection. The ACK flag indicates that the acknowledgment field should not be ignored; that is, that this packet acknowledges a previous one. The ACK flag is high for every packet after the initial SYN packet. For the SYN-ACK packet in the TCP handshake, both the SYN and ACK flags are high. Finally, the FIN flag goes high for packets in the closing handshake.

The go-back-N window size specifies what window size to use for the go-back-N protocol. In our implementation we just set the window size to 3 on both nodes; however, the full 16 bits are available in case fancier implementations are called for in the future. For instance, a more sophisticated version of LASERNET might use a different window size for each direction of data transmission, or dynamically scale the window size to improve the transmission efficiency (larger window sizes are generally more desirable for communication links with higher propagation delay, so that many packets can be transmitted without being individually acknowledged).

Finally, the 16-bit checksum is where the TCP checksum is bookkept. The TCP checksum is the complement of the one's complement sum of all 16-bit "words" in the packet. In other words, every 16-bit block in the segment is added using an end-around carry to produce a 16-bit "sum", and the checksum is the bitwise NOT of that sum. On the receiving end, an incoming packet is deemed valid if the one's complement sum of all 16-bit words in the packet (including the checksum) sums to a 16-bit zero (see also *receivepacket.v*).

Note that we have removed a substantial number of features supported by a full implementation of TCP. We don't use most of the flag bits, and we have also removed the data offset value (usually at the start of the fourth octet) and the urgent pointer (which typically comes after the checksum). We have also left out the optional header fields typically added after fifth octet. These features are optionally supported by TCP and are not critical to the core functionality of the protocol. Although they're unimplemented, we reserve empty bits for them so that the packet header stays faithful to the TCP specification.

Packet verification (receivepacket.v)

The packet verification module processes incoming packets received through the photodiode by *serial_rx.v*. It runs a small state machine that verifies that the TCP checksum is intact, and updates the primary state machine and the VGA display with new acknowledgments and new data. The inputs and outputs are as follows:

input clk: system clock
input reset: system reset
input ready: ready line pulled high by *serial_rx.v* if new packet has been received
input [31:0] ISN: the initial sequence number of the incoming packets. We set it to zero.
input [32*9 - 1:0] packet: the new packet received by *serial_rx.v*

output [31:0] seq: the sequence number of the last intact packet received in order
output [31:0] ack: the acknowledgement number of the last intact packet received
output [8:0] flags: the flag bits of the last intact packet received
output [16*8*5 - 1:0] message: buffer for received message (five 128-bit blocks).

When ready goes high, the system loads the new packet from *serial_rx.v* into an internal bus. Before the data or the SEQ and ACK numbers can be released to other modules, *receivepacket.v* checks two conditions. First, is the packet accurate? This is checked by verifying the TCP checksum. The module performs a one's complement sum of all 16-bit words in the packet. If it comes out to zero, the packet is intact and the state machine continues processing the packet. Otherwise, the packet has been corrupted in transmission. The module discards the received packet and returns to the idle state without updating any outputs.

Second, is the packet in order? According to the go-back-N protocol, data packets are only accepted and acknowledged if they are received in order; that is, packet 3 is only accepted if packet 2 was the last packet accepted. The module tracks the sequence number of the last accepted packet with an internal bus, highestSNreceived. If the new packet has a sequence number of exactly highestSNreceived + 1, the packet is in order. The outgoing sequence number and message buffer are updated accordingly (if the packet accepted is sequence number 2, then the second 128-bit block of the message bus is updated with the four octets of data from the new packet). Otherwise, the outgoing sequence number and message buffer remain unchanged, and the new data is discarded.

However, packets that arrive with out-of-order *data* still carry relevant information in the ACK and flags field. Therefore, the ACK and flags outputs are updated with the acknowledgement number and flag bits of the incoming packet, provided it is intact, even if the sequence number of the new packet is out of order.

Notice that there is no ready line to *mainfsm.v* to notify it that a new packet has been accepted. This is because *receivepacket.v* operates asynchronously from *mainfsm.v*, checking new packets as they arrive and updating its outputs with the most recent information. The main state machine will check the outputs of *receivepacket.v* as necessary to decide which state transitions are required (and what information to send in the next outgoing packet).

3.2. Physical Layer

Developed by Amanda

When designing how nodes would physically interact, one must first decide whether to use serial or parallel type communication protocol. I chose to implement an asynchronous serial protocol, based on the existing RS-232 protocol.

Because there is no shared clocked between transmitter and receiver, the protocol uses a start bit, stop bit, and a fixed baud rate and data length. Below is a diagrammatic oscilloscope trace for a sample transmission of a packet of 8 bits. Notice, the least Significant bit (LSB) is sent first and the most significant bit (MSB) is sent last before the

stop bit. When the channel is idle (the transmitter is not sending anything), the laser is off.

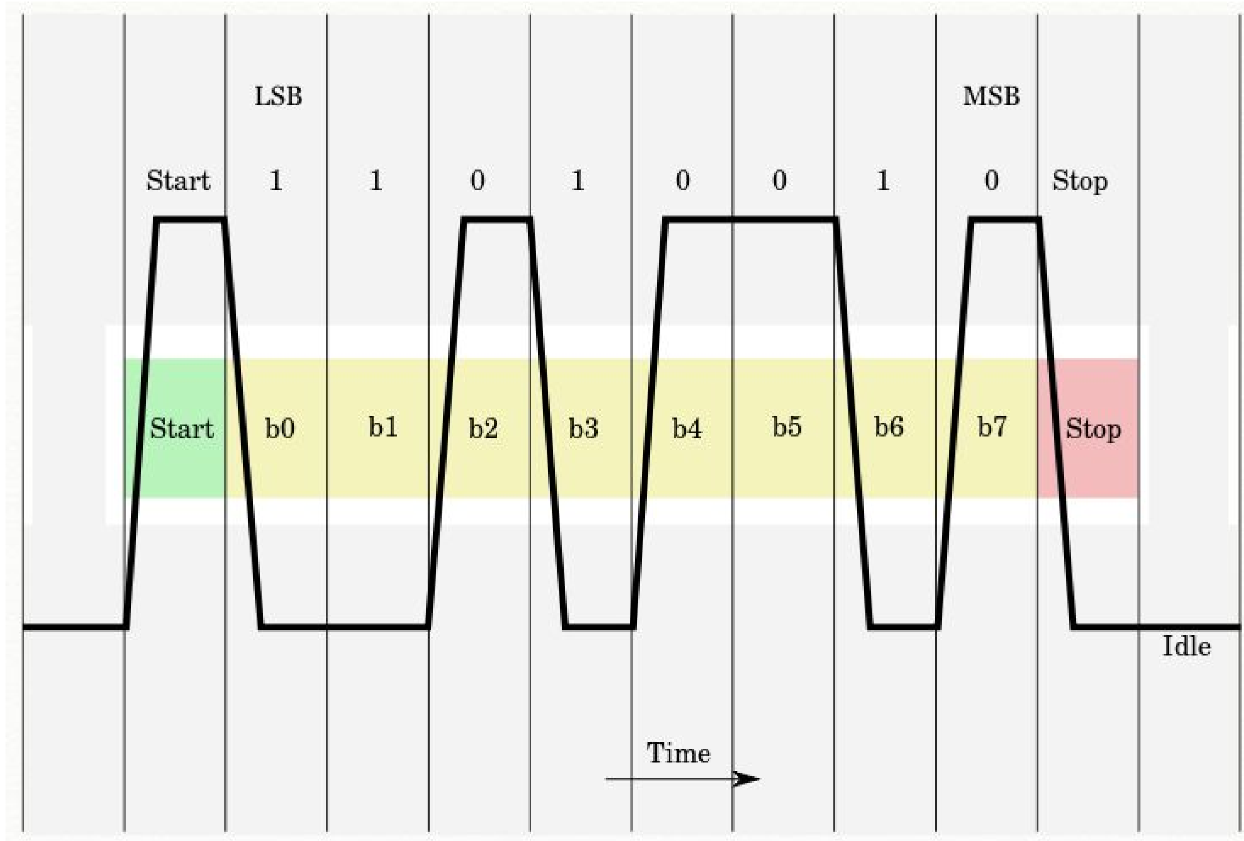

Figure 3.2.1 - Diagrammatic oscilloscope trace for a data length of 8 bits.

The baud rate is 1200. This rate was chosen because it was slow enough to see incoming transmissions appear on the VGA display one by one. With a faster baud rate, the messages appear almost instantaneously; which, in our opinion looks somewhat boring. The parameter CLK_PER_BIT sets the baud rate for the transmitter. For a certain baud rate, to calculate what to set CLK_PER_BIT, use this formula:

CLK_PER_BIT = clock frequency/baud rate

CLK_PER_BIT sets the number of clock cycles a high bit or low bit is transmitted for. For a digital ‘1’, the laser is on for CLK_PER_BIT cycles, and for a digital ‘0’, the laser is off for CLK_PER_BIT cycles.

Laser modulation (serial_tx.v)

This module controls the laser output, basically whether the laser is on or off.

This module has the following inputs, outputs, and parameters:

parameter PKT_LENGTH: number of bits in a packet (we use 9*32 = 288, or 9 octets)

input clk: system clock
input rst: system reset
input [PKT_LENGTH-1:0] data: data to be transmitted
input new_data: high when new data is ready to be transmitted

output tx: the output pin JA[0] is assigned to this wire, is high when the laser is on and low when the laser is off
output busy: high when laser is the midst of transmitting a packet, low when not transmitting
output done: high for one clock cycle after transmission of data is done, low otherwise

When the input new_data is high, the state machine in the transmitter module transitions to the START_BIT state, and the output bit busy goes high. After CLK_PER_BIT cycles, the state machine transitions to the DATA state. In this state, the laser is on or off corresponding to a 1 or 0, for CLK_PER_BIT clock cycles. After transmitting PKT_LENGTH-1 data bits, the state machines transitions to the STOP_BIT state. The stop bit is a 0. Finally, after finishing transmitting, the output wire busy goes low and the output wire done goes high for 1 clock cycle.

Photodiode input (serial_rx.v)

This module handles the received laser input from the photodiode receiver.

This module has the following inputs, outputs, and parameters:

parameter PKT_LENGTH: number of bits in a packet (we use 9*32 = 288, or 9 octets)

input clk: system clock
input rst: system reset

input rx: received input from input pin JA[1], the output of the receiver is fed into here

output [PKT_LENGTH-1:0] data: decoded data
output new_data:h high when full packet is received and decoded, low otherwise

When the receiver detects activity on the channel - input pin JA[1] is high, the state machine in the receiver transitions to the WAIT_HALF state. This receiver stays in this state for CLK_PER_BIT/2 clock cycles; this is to ensure that the receiver samples near the center of the data bit. After receiving PKT_LENGTH-1 data bits, the receiver transitions into the WAIT_HIGH state. Once the STOP_BIT is detected (JA[1] = 0), the receiver returns to the IDLE state. While receiving a transmission, the value output bus data is whatever the reg data_q is at that time. Only when a transmission is received fully is when the output bit new_data is high for one clock cycle.

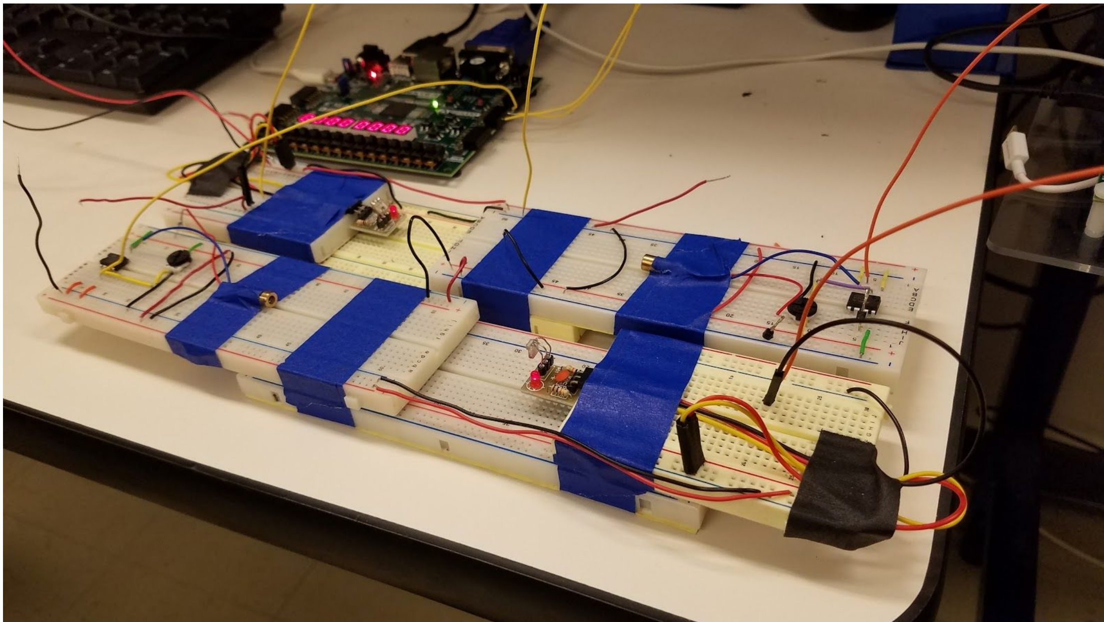

Figure 3.2.2 - Lasers and photodiodes. The laser transmitter and photodiode for the FPGA in the background are on the left side of the breadboard assembly. The photodiode and laser for the other FPGA (not shown) are on the right.

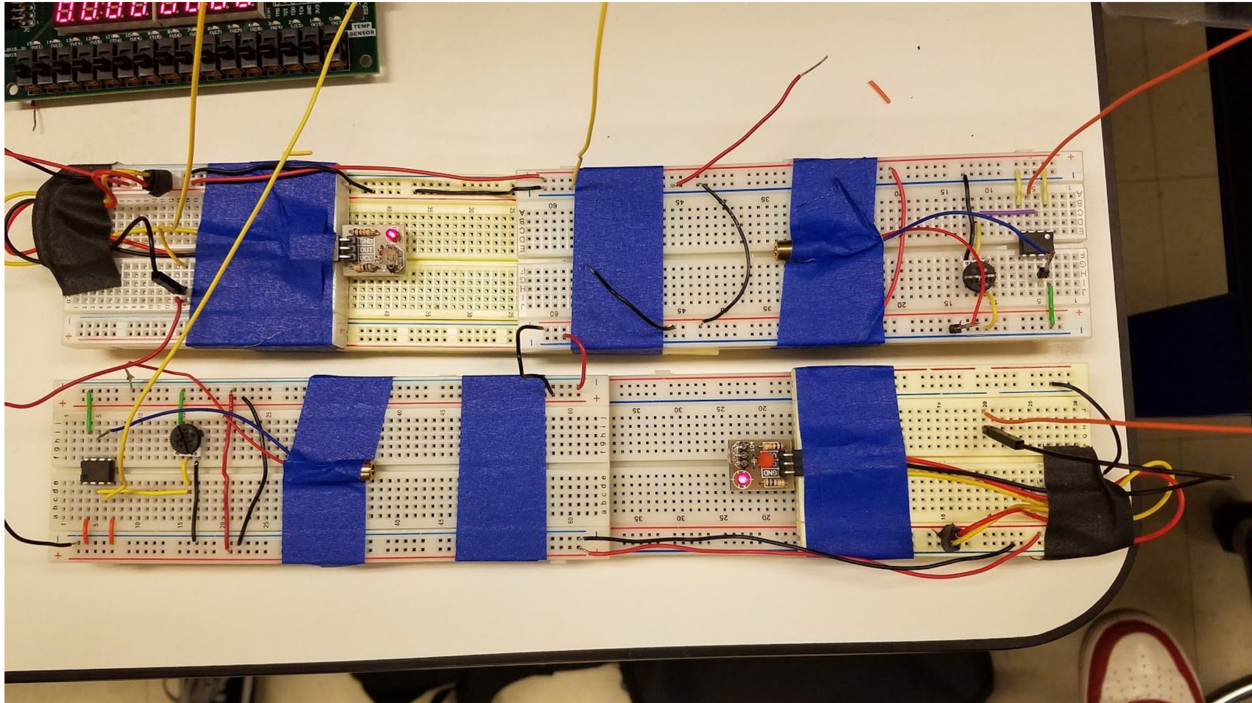

Figure 3.2.3 - Lasers and photodiodes, top view.

3.3. User Interface

Developed by Amanda

Keyboard input (keyboardinput.v)

These modules handle user input from the USB keyboard attached to the Nexys 4 DDR USB hub. When using the Nexys 4 DDR FPGA for any projects involving the USB host connector, lines 219 under the USB-RS232 Interface controlling pin D4 and lines 225 and 226 controlling package pins F4 and B2, respectively, under the USB HID, are not commented out.

The module keyboardexport has the following inputs and outputs:

input clock_65mhz: system clock
input reset: system reset
input ps2_clock,
input ps2_data,
input save: when save goes high, last sixteen characters get added to memory

output reg [16*8 - 1:0] cstring: last sixteen characters typed

output [16*8*5 - 1:0] messageout: 16*8 by 5 block of memory for holding outgoing message

The module keyboardexport outputs the last sixteen typed characters to the outgoing message block. The outgoing message block memory also lives in this module. This module receives the last sixteen characters from the keyboard by calling the ps2_ascii_input module.

The module ps2_ascii_input has the following input and outputs:

input clock_65mhz: system clock
input reset: system reset
input clock: PS/2 clock
input data: PS/2 data

output [7:0] ascii: ascii code (1 character)
output ascii_ready: ascii ready (one clock cycle active high)

This module generates ASCII codes from keyboard scan codes. A scancode is an 8 bit number is assigned to each key on the keyboard. On the figure below, keyboard scan codes are matched to their keyboard keys on a QWERTY keyboard. This module receives the latest keyboard scan code from the ps2 module. It matches the scan code to an ASCII character using a very long case statement. Not all keyboard scan codes are matched to an ASCII character; for example, the scan code for the button F2 is not matched to an ASCII character. In cases where a user does press the keys with scan codes not matched to an ASCII character, the case statements defaults to a ‘#’.

ESC 76 | F1 05 | F2 06 | F3 04 | F4 0C | F5 03 | F6 0B | F7 83 | F8 0A | F9 01 | F10 09 | F11 78 | F12 07

` ~ 0E | 1 ! 16 | 2 @ 1E | 3 # 26 | 4 $ 25 | 5 % 2E | 6 ^ 36 | 7 & 3D | 8 * 3E | 9 (46 | 0) 45 | - _ 4E | = + 55 | BackSpace ← 66

TAB 0D | Q 15 | W 1D | E 24 | R 2D | T 2C | Y 35 | U 3C | I 43 | O 44 | P 4D | [{ 54 |] } 5B | \ | 5D

Caps Lock 58 | A 1C | S 1B | D 23 | F 2B | G 34 | H 33 | J 3B | K 42 | L 4B | ; : 4C | ' " 52 | Enter ↵ 5A

Shift 12 | Z 1Z | X 22 | C 21 | V 2A | B 32 | N 31 | M 3A | , < 41 | > . 49 | / ? 4A | ⇧ Shift 59

Ctrl 14 | Alt 11 | Space 29 | Alt E0 11 | Ctrl E0 14

Figure 3.3.1 - Keyboard scan codes matched to keyboard keys

The module ps2 has the following input and outputs:

input clock: system clock
input reset: system reset
input ps2c: ps2 clock
input ps2d: ps2 data
input fifo_rd: fifo read request (active high)

output [7:0] fifo_data: fifo data output
output fifo_empty: fifo empty (active high)
output fifo_overflow: fifo overflow - too much kbd input

This module synchronously works with the system clock. The keyboard can send data to the host only when both the ps2d and clock lines are high (or idle). Because the host is the bus master, the keyboard must check to see whether the host is sending data before driving the bus. To facilitate this, the clock line is used as a “clear to send” signal. If the host drives the clock line low, the keyboard must not send any data until the clock is released. The keyboard sends data to the host in 11-bit words that contain a ‘0’ start bit, followed by 8-bits of scan code (LSB first), followed by an odd parity bit, and terminated with a ‘1’ stop bit. The keyboard generates 11 clock transitions (at 20 to 30 KHz) when the data is sent, and data is valid on the falling edge of the clock. This module has built-in first-in, first-out (FIFO) buffer, and its output to the ps2_ascii_input is whatever the read pointer (rptr) is pointing to. Once a scancode is read, the read pointer increments.

VGA monitor display (screenlayout.v)

The VGA monitor display is controlled by the module screenlayout.v . This module sets the layout of the the user interface, defining the coordinates on the screen where the incoming text, outgoing text, and keyboard input are displayed.

This module has the following inputs and outputs:

input clock_65mhz: system clock
input [10:0] hcount: horizontal index of current pixel (0..1023)
input [9:0] vcount: vertical index of current pixel (0..767)
input [16*8*5 - 1:0] messageout: outgoing message to display
input [16*8*5 - 1:0] messagein: incoming message to display
input [16*8 - 1:0] keyboard: last 16 characters entered on keyboard

output [2:0] pixels: pixels to be displayed

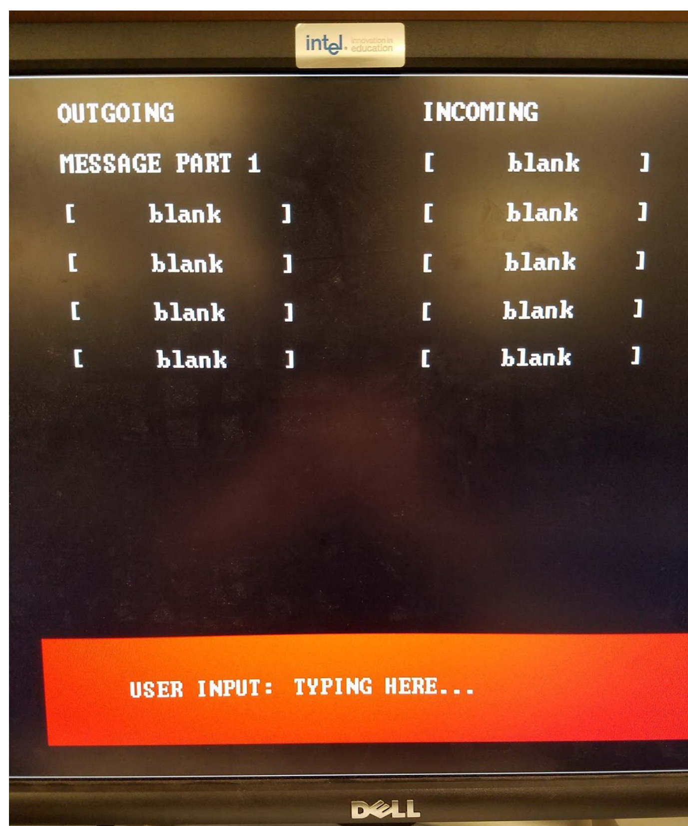

Figure 3.3.2 - VGA monitor display

Within the screenlayout.v module, the module char_string_display is called. This module converts the string of characters to display and output the string's pixels, and controls the font, size, and color of the displayed string.

This module has the following inputs, outputs, and parameters:

parameter NCHAR = 8: number of 8-bit characters in cstring
parameter NCHAR_BITS = 3: number of bits in NCHAR

input vclock: 65MHz clock
input [10:0] hcount: horizontal index of current pixel (0..1023)
input [9:0] vcount: vertical index of current pixel (0..767)
input [NCHAR*8-1:0] cstring: character string to display
input [10:0] cx: x-coordinate of the upper left corner to display string at
input [9:0] cy: y-coordinate of the upper left corner to display string at

output [2:0] pixel: pixels to be displayed

How this module works is that for each character in cstring, the module performs a lookup for which exact pixels in the 8 by 12 block to display for that character in the font rom. If hcount and vcount are in the character's pixel area, the characters pixels are generated and displayed. If one wanted to change the font of the characters, font_rom.mif would have to be modified.

4. Development Notes

When we initially proposed LASERNET, we anticipated that the basic mechanism of transmitting packets over a laser link with TCP would be fairly straightforward. We ambitiously estimated that it would only take two weeks of work to complete, giving us ample time to build interesting applications and fancier features. The name LASERNET hints at our original stretch goal: to build a wireless mesh network of FPGAs, all exchanging packets with each other with TCP/IP. We also wanted to implement a connectionless protocol like UDP, which would let us build showier applications like video streaming or "phone calls" in the style of VoIP.

However, when we started building, we quickly realized that simply implementing TCP would be more complex than we originally anticipated. The block diagram we originally proposed would not actually perform the features we needed, and we had to iterate

multiple times before deciding how the submodules would actually divide tasks to bookkeep TCP and go-back-N. Our lack of intuition when it comes to Verilog design also added delays. We realized that the five labs we completed earlier had been relatively straightforward by comparison; in the labs, the Verilog architecture had already been specified and we simply needed to implement the modules. Starting with a blank page and trying to design an entire system from the ground up was a colossal task and gave us a new appreciation for how well-designed the labs had been.

While building, we found it useful to take a very incremental development approach. As soon as we had a module specified, we built the most minimal version possible, using the switches and displays on the Nexys board to simulate inputs and outputs. The finite state machine which currently coordinates our entire system started as a standalone module that changed states based on a button press, displaying the current state number on the seven-segment display. The process of implementing the TCP handshake and the go-back-N protocol was made much simpler when we realized we could simulate incoming packet headers with the Nexys switches. By displaying the incoming and outgoing SEQ/ACK numbers on the seven-segment display, we could confirm that the state machine was correctly stepping through states and bookkeeping go-back-N before adding new features or incorporating peripheral modules.

Taking an incremental approach allowed us to iterate on our block diagram without wasting too much work. We only implemented a new submodule if we felt confident that its dependencies were fully functioning, so we realized right away if our original design was faulty and updated our architecture accordingly. We also made an effort to rigorously document our code as we went, which made integration easier down the line. As we built up a functioning codebase, it became easier and easier to develop new submodules and build on the existing architecture (perhaps a testament to the maxim that “the hardest part is getting started”).

Because we were debugging as we went, code integration went smoother than we expected. We were very surprised to find that the transmission/reception modules integrated successfully with the finite state machine on our first attempt. We were able to exchange three or four packets before the system timed out, and with some tweaks to the state machine, LASERNET was working smoothly within the next hour.

However, we also suspect that we started integrating much later than we should have. Coordinating our efforts was difficult because we were both used to working on projects independently. We didn’t feel comfortable integrating our code until we were both confident our submodules were working correctly. It’s possible that if we had integrated

earlier and debugged the entire system together (instead of debugging our own pieces individually) we would have completed a working system earlier.

Although we didn't achieve all the goals we set for ourselves, we're proud of the system that we've built. We believe it's well-designed and neatly organized, and it performs all the basic functions of a communications link. We demonstrate in our video (Section 6.1) how the system successfully transmits error-free messages, and successfully recovers from dropped packets (which we simulate by blocking the laser with a finger). Along the way, we learned a ton about what FPGAs and digital logic can accomplish. We were also excited to work with protocols and systems that interest us. Allan learned a ton about how TCP works and gained familiarity with go-back-N, while Amanda learned about the unique challenges of modulating and receiving data with lasers. Since we both have interests in wireless communication and networks, we appreciated the opportunity to work together (and learn together) while designing LASERNET, our tiny tribute to the future of the internet.

5. Conclusion

Out of enthusiasm for the potential that lasers have for high-speed wireless infrastructures, we have implemented a simple laser communication system for passing text messages between two FPGAs. We implement TCP (with checksums) for robustness against transmission errors and dropped packets. The current implementation exchanges five packets and uses go-back-N with a sliding window of 3, but these parameters can be easily modified to accommodate larger file transmissions or communication links with higher propagation delays. We have also left placeholder bits in the TCP header for optional TCP features not implemented here, such as packet prioritization, urgent pointers, or window scaling; this serves as scaffolding for any future developers who may wish to build on our basic design.

6. Appendix

6.1. Video

A video demonstration of LASERNET in action is available here.

6.2. Verilog

Complete Verilog code for LASERNET is available on github here as a Vivado project folder, including a fully-built bitstream file for loading onto the FPGA. We include the text of the primary modules below, for reference.

nexys-wrapper.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
//
// Create Date:
// Design Name:
// Module Name: labkit
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module labkit(
   input CLK100MHZ,
   input[15:0] SW,
   input BTNC, BTNU, BTNL, BTNR, BTND,
   input PS2_CLK, PS2_DATA,
   output UART_RXD_OUT,
   output[3:0] VGA_R,
   output[3:0] VGA_B,
   output[3:0] VGA_G,
   output VGA_HS,
   output VGA_VS,
   inout [7:0] JA,
   output LED16_B, LED16_G, LED16_R,
   output LED17_B, LED17_G, LED17_R,
   output[15:0] LED,
   output[7:0] SEG,  // segments A-G (0-6), DP (7)
   output[7:0] AN    // Display 0-7
   );

////////////////////////// SETUP ////////////////////////////////////////////////

/////////// CLOCKS

// create 65mhz clock for xvga
wire clock_65mhz;
wire locked;
clk_wiz_0 clock65(.clk_in1(CLK100MHZ), .clk_out_65mhz(clock_65mhz),
            .reset(1'b0),.locked(locked));

// create system clock
wire clocksys;
assign clocksys = clock_65mhz; // just clock everything at 65mhz
```

```
//////////// SWITCHES, BUTTONS, LEDS, DISPLAYS

//  instantiate 7-segment display
wire [31:0] to_display;
wire [6:0] segments;
display_8hex display(.clk(clocksys),.data(to_display), .seg(segments), .strobe(AN));
assign SEG[6:0] = segments;
assign SEG[7] = 1'b1;

// SW[15] for system reset
wire reset;
synchronize s1(.clock(clocksys),
        .in(SW[15]),.out(reset));

// SW[13] for toggling monitor display
wire screenmode;
synchronize s2(.clock(clocksys),
        .in(SW[13]),.out(screenmode));

// BTNC for opening a TCP connection
wire openTCP;
debounce db1(.reset(reset),.clock(clocksys),
       .noisy(BTNC),.clean(openTCP));

// BTNL for saving keyboard data to outgoing message block
wire save;
debounce db2(.reset(reset),.clock(clocksys),
       .noisy(BTNL),.clean(save));


//////////////////// INSTANTIATE AND WIRE UP MODULES ////////////////////////

////////////////////////// MAIN STATE MACHINE

wire packetsent;            // goes high for a cycle when a packet is finished sending

wire incomingready;         // incoming packet is ready
wire [31:0] incomingACK;    // incoming acknowledgment number
wire [31:0] incomingSEQ;    // incoming sequence number
wire [8:0] incomingflags;   // incoming TCP flags

wire outgoingready;         // outgoing packet is ready
wire [31:0] outgoingACK;    // outgoing acknowledgment number
wire [31:0] outgoingSEQ;    // outgoing sequence number
wire [8:0] outgoingflags;   // outgoing TCP flags

wire [3:0] state;           // to display the current state

wire [31:0] ISN;            // first sequence number to use
assign ISN = 32'd0;         // PARAMETER

wire [31:0] SNmax;          // largest sequence number available in data; equal to # of packets in message
assign SNmax = 32'h5;       // PARAMETER

wire [15:0] windowsize;     // window size for go-back-n protocol
assign windowsize = 16'd2;  // PARAMETER
```

```
mainfsm statemachine(.clk(clocksys), .reset(reset), .open(openTCP), .packetsent(packetsent),
            .ISN(ISN), .SNmax(SNmax),
            .window(windowsize),
            .readyin(incomingready), .ACKin(incomingACK), .SEQin(incomingSEQ), .flagsin(incomingflags),
            .readyout(outgoingready), .ACKout(outgoingACK), .SEQout(outgoingSEQ), .flagsout(outgoingflags),
            .statedisplay(state));

//// memory blocks for holding incoming and outgoing messages
//// each packet is 16 characters, can store/display 5 packets at a time

wire [16*8*5 - 1 : 0] outgoing; // outgoing message
wire [16*8*5 - 1 : 0] incoming; // incoming message

////////////////////////// KEYBOARD INPUT

wire [16*8 - 1 : 0] currentkeyboard;

keyboardexport kbdexport1(.clock_65mhz(clock_65mhz), .reset(reset),
              .ps2_clock(PS2_CLK), .ps2_data(PS2_DATA),
              .cstring(currentkeyboard),
              .save(save), .messageout(outgoing)
              );

//////////////////////// PACKET GENERATOR AND CHECKSUM

wire [32*9 - 1 : 0] outgoingpacket;
wire readytotransmit;

makepacket packetgen(.clk(clocksys), .reset(reset),
            .ISN(ISN), .readyin(outgoingready),
            .window(windowsize),
            .seq(outgoingSEQ),.ack(outgoingACK), .flags(outgoingflags),
            .message(outgoing),
            .packet(outgoingpacket),.readyout(readytotransmit));

/////////////////////// PACKET RECEIVER AND CHECKSUM
wire [32*9 - 1 :0] incomingpacket;  // incoming packet received from photodiode
wire readyfromlaserinput;        // goes high when a new packet is fully received from photodiode

receivepacket packetrcv(.clk(clocksys), .reset(reset),
              .ready(readyfromlaserinput),
              .ISN(ISN),
              .packet(incomingpacket),
              .seq(incomingSEQ),.ack(incomingACK),.flags(incomingflags),
              .message(incoming));

/////////////////////// LASER OUTPUT AND PHOTODIODE INPUT


wire laser_out;
wire busy;
wire photodiode_in;

assign JA[0] = laser_out;
assign photodiode_in = JA[1];
```

```
serial_tx stx(.clk(clocksys),
       .rst(reset),
       .data(outgoingpacket),        // outgoing data packet
       .new_data(readytotransmit),   // goes high when new packet available to transmit
       .tx(laser_out),               // laser signal output to JA[0]
       .busy(busy),                  // high when transmitting a message, low when not
       .done(packetsent));           // high for one cycle after completing transmission
defparam stx.CLK_PER_BIT = 54166; //1350 for 4800 baud rate, 54166 for 1200 baud rate
defparam stx.PKT_LENGTH = 32*9;

serial_rx srx(.clk(clocksys),
       .rst(reset),
       .rx(photodiode_in),               // incoming photodiode signal on JA[1]
       .data(incomingpacket),            // incoming data packet
       .new_data(readyfromlaserinput));  // high for one cycle when new packet available
defparam srx.CLK_PER_BIT = 54166; //1350 for 4800 baud rate, 54166 for 1200 baud rate
defparam srx.PKT_LENGTH = 32*9;


//////////////////////// workspace

// display some important numbers to seven-segment display and LEDs
assign to_display = {incomingACK[3:0], incomingSEQ[3:0], outgoingACK[3:0], outgoingSEQ[3:0], 12'h000, state};

assign LED[2:0] = {outgoingflags[4], outgoingflags[1], outgoingflags[0]};
assign LED[5:3] = {incomingflags[4], incomingflags[1], incomingflags[0]};

assign LED[12] = busy;              // LED constant when laser transmitting
assign LED[13] = laser_out;         // LED flashes when laser transmitting
assign LED[14] = photodiode_in;     // LED flashes when laser receiving

//// these lines are for debugging the state machine and simulating incoming packets
// assign packetsent = SW[14]; // simulate a packet being sent
// assign incomingflags = {4'b0000, SW[2], 2'b00, SW[1], SW[0]}; // simulate incoming ack, syn, fin
// assign incomingACK = {28'd0, SW[11:8]};
// assign incomingSEQ = {28'd0, SW[7:4]};



////////////////////////  XVGA DISPLAY ////////////////////////////////////

//  generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync,vsync,blank;
xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);

wire [2:0] sample_pixels;
wire dis;
screenlayout face( .clock_65mhz(clock_65mhz),
          .hcount(hcount), .vcount(vcount),
          .display(dis),
          .messageout(outgoing), .messagein(incoming),
          .keyboard(currentkeyboard),
          .pixels(sample_pixels) );

//  red text box for displaying user input
```

```
wire [23:0] paddle_pixel;
blob #(.WIDTH(800),.HEIGHT(128),.COLOR(24'hFF_00_00))   // red!
     paddle1(.x(11'd100),.y(10'd600),.hcount(hcount),.vcount(vcount),
             .pixel(paddle_pixel));

//  white border around screen
wire [2:0] white_outline_pixels;
assign white_outline_pixels = (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 7 : 0;


/////// output to vga

// screenmode is a switch that selects what to show on the screen

reg [2:0] rgb;
reg b,hs,vs;
always @(posedge clock_65mhz) begin
  hs <= hsync;
  vs <= vsync;
  b <= blank;
  if (screenmode == 1'b1) begin
    // 1 pixel outline of visible area (white)
    rgb <= white_outline_pixels;
  end
  else begin
     // default: text
   rgb <=  sample_pixels |
           white_outline_pixels ;
  end
end

assign VGA_R = {4{rgb[2]}} | paddle_pixel[23:16];
assign VGA_G = {4{rgb[1]}} | paddle_pixel[15:8];
assign VGA_B = {4{rgb[0]}} | paddle_pixel[7:0];

assign VGA_HS = ~hs;
assign VGA_VS = ~vs;


endmodule
```

---

mainfsm.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: keam / allanko for 6.111 fa2016 - lasernet
// Engineer:
//
// Create Date:
// Design Name:
// Module Name: mainfsm
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
```

```
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

// operates controller FSM and tracks go-back-n protocol

module mainfsm(
    input clk,
    input reset,
    input open,
    input packetsent,  // high for one cycle after a packet is transmitted
    input [31:0] ISN,                          // initial sequence number to use
    input [31:0] SNmax,                        // largest address of data to send; ie the address of the last data packet
    input [15:0] window, // size of go-back-n window
    input readyin,
    input [31:0] ACKin,
    input [31:0] SEQin,
    input [8:0] flagsin,
    output reg readyout,                       // high for one cycle when you want to send a new packet
    output reg [31:0] ACKout,
    output reg [31:0] SEQout,
    output [8:0] flagsout,
    output reg [3:0] statedisplay // figure out how many bits you need for this
    );

    // read incoming flags
    wire flagsinACK ;
    assign flagsinACK = flagsin[4];

    wire flagsinSYN;
    assign flagsinSYN = flagsin[1];

    wire flagsinFIN;
    assign flagsinFIN = flagsin[0];

    reg FINreceived; // bookkeeps when you've received a FIN

    // wires for outgoing flags
    reg flagsoutACK;
    reg flagsoutSYN;
    reg flagsoutFIN;

    assign flagsout = {4'b0000, flagsoutACK, 2'b00, flagsoutSYN, flagsoutFIN};

    ////////////////// ESTABLISH STATES ////////////////////////////////////
    reg [3:0] state, nextstate;

    parameter S_PASSIVE_OPEN                   = 4'h0; // idle

    // opening a TCP connection

    parameter S_ACTIVE_OPEN                            = 4'h1; // (initiating) - sending SYN
    parameter S_CONNECTED                      = 4'h2; // (initiating) received SYN-ACK - sending ACK
```

```
parameter S_ACTIVATED                   = 4'h3; // (listening) - received SYN - sending SYN-ACK

// transmitting packets

parameter S_TRANSMITTING                = 4'h4; // making the next packet to transmit
parameter S_TRANSMIT_WAIT               = 4'h5; // waiting for packet to get transmitted

// closing the TCP connection

parameter S_FIN                 = 4'h6; // no more data - transmitting FIN flags
parameter S_FIN_WAIT                    = 4'h7; // waiting for fin packet to get transmitted


//////////////////////////////////////////////////////////////////////////////

// tracking important numbers
reg [31:0] SN;                  // sequence number relative to ISN (SEQout = ISN + SN)
reg [31:0] lastACK;         // last acknowledgment received from other node
reg [31:0] nextACK;         // next acknowledgment to send to other node

// counter for closing connection at the end of FIN_WAIT
reg [29:0] finwaitcounter = 0;
parameter [29:0] FINWAITMAX = 30'd325_000_000; // equivalent to 5 seconds at 65 mhz

// state behavior
always @(*) begin
        case(state)
        S_PASSIVE_OPEN : begin

                statedisplay = S_PASSIVE_OPEN;

                flagsoutSYN = 1'b0;
                flagsoutACK = 1'b0;
                flagsoutFIN = 1'b0;
                ACKout = 32'd0;
                SEQout = ISN + SN;

                nextstate = open ? S_ACTIVE_OPEN :
                                        (flagsinSYN & !flagsinACK) ? S_ACTIVATED :
                                        S_PASSIVE_OPEN;

        end

        S_ACTIVE_OPEN : begin

          statedisplay = S_ACTIVE_OPEN;

                flagsoutSYN = 1'b1;
                flagsoutACK = 1'b0;
                flagsoutFIN = 1'b0;
                ACKout = 32'd0;
                SEQout = ISN + SN;

                nextstate = (flagsinSYN & flagsinACK & (ACKin == (ISN + 1))) ? S_CONNECTED :
                                        S_ACTIVE_OPEN;
```

```
end

S_CONNECTED : begin

        statedisplay = S_CONNECTED;

        flagsoutSYN = 1'b0;
        flagsoutACK = 1'b1;
        flagsoutFIN = 1'b0;
        ACKout = nextACK;
        SEQout = ISN + SN;

        nextstate = packetsent ? S_TRANSMITTING :
                                 S_CONNECTED;

end

S_ACTIVATED : begin

        statedisplay = S_ACTIVATED;

        flagsoutSYN = 1'b1;
        flagsoutACK = 1'b1;
        flagsoutFIN = 1'b0;
        ACKout = nextACK;
        SEQout = ISN + SN;

        nextstate = (!flagsinSYN & flagsinACK & (ACKin == (ISN + 1))) ? S_TRANSMITTING :
                                 S_ACTIVATED;

end

S_TRANSMITTING : begin

        statedisplay = S_TRANSMITTING;

        flagsoutSYN = 1'b0;
        flagsoutACK = 1'b1;
        flagsoutFIN = 1'b0;
        ACKout = nextACK;
        SEQout = ISN + SN;

        nextstate = S_TRANSMIT_WAIT;

end

S_TRANSMIT_WAIT : begin

        statedisplay = S_TRANSMIT_WAIT;

        flagsoutSYN = 1'b0;
        flagsoutACK = 1'b1;
        flagsoutFIN = 1'b0;
        ACKout = nextACK;
        SEQout = ISN + SN;

        nextstate = (lastACK == ISN + SNmax + 32'd1) ? S_FIN :
```

```
                                                        packetsent ? S_TRANSMITTING :
                                                        S_TRANSMIT_WAIT;

        end

        S_FIN : begin

                statedisplay = S_FIN;

                flagsoutSYN = 1'b0;
                flagsoutACK = 1'b1;
                flagsoutFIN = 1'b1;
                ACKout = nextACK;
                SEQout = ISN + SN;
                nextstate = ((lastACK == ISN + SNmax + 32'd2) & FINreceived ) ? S_PASSIVE_OPEN :
                                                        S_FIN_WAIT;

        end

        S_FIN_WAIT : begin

                statedisplay = S_FIN_WAIT;

                flagsoutSYN = 1'b0;
                flagsoutACK = 1'b1;
                flagsoutFIN = 1'b1;
                ACKout = nextACK;
                SEQout = ISN + SN;

                nextstate = packetsent ? S_FIN :
            finwaitcounter == FINWAITMAX ? S_PASSIVE_OPEN :
                                                        S_FIN_WAIT;
        end

        default : nextstate = S_PASSIVE_OPEN;

        endcase
end


// clocked state updates
always @(posedge clk) begin
        state <= reset ? S_PASSIVE_OPEN : nextstate;

        case(nextstate)

        S_PASSIVE_OPEN : begin

                nextACK <= 32'd0;
                SN <= 32'd0;
                lastACK <= 32'd0;
                readyout <= 1'b0;
                FINreceived <= 1'b0;
      finwaitcounter <= 20'd0;

        end
```

```
    S_ACTIVE_OPEN : begin

                nextACK <= 32'd0;
                SN <= 32'd0;
                lastACK <= 32'd0;
                readyout <= (nextstate != state) ? 1'b1 : 1'b0;
                FINreceived <= 1'b0;
finwaitcounter <= 20'd0;

    end

    S_CONNECTED : begin

                nextACK <= (nextstate != state) ? SEQin + 32'd1 : // nextACK sampled upon entry
                                                    nextACK;
                SN <= 32'd0;
                lastACK <= (nextstate != state) ? ACKin : // lastACK sampled upon entry
                                                    lastACK;

                readyout <= (nextstate != state) ? 1'b1 : 1'b0;

                FINreceived <= 1'b0;
finwaitcounter <= 20'd0;

    end

    S_ACTIVATED : begin

                nextACK <= (nextstate != state) ? SEQin + 32'd1 : // nextACK sampled upon entry
                                                    nextACK;
                SN <= 32'd0;
                lastACK <= 32'd0;

                readyout <= (nextstate != state) ? 1'b1 : 1'b0;

                FINreceived <= 1'b0;
finwaitcounter <= 20'd0;

    end

    S_TRANSMITTING : begin

                nextACK <= (nextstate != state) ? SEQin + 32'd1 : // nextACK sampled upon entry
                                                    nextACK;


                SN <= ((nextstate != state) & (ISN + SN == ACKin + window)) ? ACKin - ISN :
                            ((nextstate != state) & (SN == SNmax)) ? ACKin - ISN :
                             (nextstate != state) ? SN + 1 :
                             SN;

                lastACK <= (nextstate != state) ? ACKin : // lastACK sampled upon entry
                                                    lastACK;

                readyout <= (nextstate != state) ? 1'b1 : 1'b0;

                FINreceived <= ((nextstate != state) & flagsinFIN) ? 1'b1 : FINreceived; // on entry, check if FIN received
```

```
        finwaitcounter <= 20'd0;

          end

          S_TRANSMIT_WAIT : begin

                    nextACK <= nextACK;
                    SN <= SN;
                    lastACK <= lastACK;
                    readyout <= 1'b0;
                    FINreceived <= FINreceived;
        finwaitcounter <= 20'd0;

          end

          S_FIN : begin
                    nextACK <= (nextstate != state) ? SEQin + 32'd1 :
                                                      nextACK;

                    SN <= SNmax + 32'd1;  // FIN packet has SN = 1 + maximum SNmax
                    lastACK <= (nextstate != state) ? ACKin :
                                                      lastACK;

                    readyout <= (nextstate != state) ? 1'b1 : 1'b0;
                    FINreceived <= ((nextstate != state) & flagsinFIN) ? 1'b1 : FINreceived; // on entry, check if FIN received
        finwaitcounter <= ACKin != lastACK ? 20'd0 : finwaitcounter; // reset finwait counter only if new ACK is different from old ACK

          end

          S_FIN_WAIT : begin

                    nextACK <= nextACK;
                    SN <= SN;
                    lastACK <= lastACK;
                    readyout <= 1'b0;
                    FINreceived <= FINreceived;
        finwaitcounter <= finwaitcounter + 1;

          end

          default : begin

                    nextACK <= 32'd0;
                    SN <= 32'd0;
                    lastACK <= 32'd0;
                    readyout <= 1'b0;
                    FINreceived <= 1'b0;

          end

          endcase

    end
```

```
endmodule
```

---

keyboardinput.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// allanko and keam for 6.111 fall 2016 - lasernet
//////////////////////////////////////////////////////////////////////////////////

//// bumping the last sixteen characters to the outgoing message block //////////
//// outgoing message block memory lives in this module! ////////////////////////

module keyboardexport(
    input clock_65mhz,
    input reset,
    input ps2_clock,
    input ps2_data,
    input save,              // when save goes high, last sixteen characters get added to memory
    output reg [16*8 - 1:0] cstring,  // last sixteen characters typed
    output [16*8*5 - 1:0] messageout  // 16*8 by 5 block of memory for holding outgoing message
    );

    // reading in sixteen characters from the keyboard

    wire [7:0] ascii;
    wire       char_rdy;

    ps2_ascii_input kbd(clock_65mhz, reset,
                   ps2_clock, ps2_data,
                   ascii, char_rdy);

    reg [3:0] kbdin_count = 0;
    reg [7:0] last_ascii;

    always @(posedge clock_65mhz) begin

        kbdin_count <= reset ? 0 : char_rdy ? kbdin_count-1 : kbdin_count;
        last_ascii <= reset ? " " : char_rdy ? ascii : last_ascii;

        cstring[7:0] <= reset ? "" : (kbdin_count==0) ? last_ascii : cstring[7:0];
        cstring[7+'o10:'o10] <= reset ? "" : (kbdin_count==1) ? last_ascii: cstring[7+'o10:'o10];
        cstring[7+'o20:'o20] <= reset ? "" : (kbdin_count==2) ? last_ascii: cstring[7+'o20:'o20];
        cstring[7+'o30:'o30] <= reset ? "" : (kbdin_count==3) ? last_ascii: cstring[7+'o30:'o30];
        cstring[7+'o40:'o40] <= reset ? "" : (kbdin_count==4) ? last_ascii: cstring[7+'o40:'o40];
        cstring[7+'o50:'o50] <= reset ? "" : (kbdin_count==5) ? last_ascii: cstring[7+'o50:'o50];
        cstring[7+'o60:'o60] <= reset ? "" : (kbdin_count==6) ? last_ascii: cstring[7+'o60:'o60];
        cstring[7+'o70:'o70] <= reset ? "" : (kbdin_count==7) ? last_ascii: cstring[7+'o70:'o70];

        cstring[7+'o100:'o100] <= reset ? "" : (kbdin_count==8) ? last_ascii: cstring[7+'o100:'o100];
        cstring[7+'o110:'o110] <= reset ? "" : (kbdin_count==9) ? last_ascii: cstring[7+'o110:'o110];
        cstring[7+'o120:'o120] <= reset ? "" : (kbdin_count==10) ? last_ascii: cstring[7+'o120:'o120];
        cstring[7+'o130:'o130] <= reset ? "" : (kbdin_count==11) ? last_ascii: cstring[7+'o130:'o130];
        cstring[7+'o140:'o140] <= reset ? "" : (kbdin_count==12) ? last_ascii: cstring[7+'o140:'o140];
        cstring[7+'o150:'o150] <= reset ? "" : (kbdin_count==13) ? last_ascii: cstring[7+'o150:'o150];
        cstring[7+'o160:'o160] <= reset ? "" : (kbdin_count==14) ? last_ascii: cstring[7+'o160:'o160];
        cstring[7+'o170:'o170] <= reset ? "" : (kbdin_count==15) ? last_ascii: cstring[7+'o170:'o170];
```

```
end

// when we made this an indexed bus, the nonblocking assignments below didn't work properly
reg [16*8 - 1:0] messageoutarray4;
reg [16*8 - 1:0] messageoutarray3;
reg [16*8 - 1:0] messageoutarray2;
reg [16*8 - 1:0] messageoutarray1;
reg [16*8 - 1:0] messageoutarray0;

assign messageout = {messageoutarray4,
                messageoutarray3,
                messageoutarray2,
                messageoutarray1,
                messageoutarray0};

reg [2:0] messageout_index = 3'd0;
parameter [2:0] MAXINDEX = 3'd4;

// when save goes high, put last sixteen characters from keyboard into outgoing memory

parameter S_WAIT = 2'b00;
parameter S_SAVE = 2'b01;
parameter S_RESET = 2'b10;

reg [1:0] state = S_RESET;
reg [1:0] laststate;

always @(posedge clock_65mhz) begin
   laststate <= state;

   case(state)

      S_WAIT : begin

         state <= reset ? S_RESET :
                save ? S_SAVE :
                S_WAIT;

      end

      S_SAVE : begin

         if (state != laststate) begin
            if (messageout_index == 3'd0) messageoutarray0 <= cstring;
            else if (messageout_index == 3'd1) messageoutarray1 <= cstring;
            else if (messageout_index == 3'd2) messageoutarray2 <= cstring;
            else if (messageout_index == 3'd3) messageoutarray3 <= cstring;
            else messageoutarray4 <= cstring;
         end

         messageout_index <= (state != laststate) & (messageout_index == MAXINDEX) ? 3'd0 :
                     (state != laststate) ? messageout_index + 3'd1 :
                     messageout_index;

         state <= reset ? S_RESET :
                save ? S_SAVE :
```

```
                S_WAIT;

        end

        S_RESET : begin

            messageoutarray0 <= "[     blank    ]";
            messageoutarray1 <= "[     blank    ]";
            messageoutarray2 <= "[     blank    ]";
            messageoutarray3 <= "[     blank    ]";
            messageoutarray4 <= "[     blank    ]";

          messageout_index <= 3'd0;

          state <= reset ? S_RESET :
                S_WAIT;

        end

        default : begin

          state <= reset;

        end

      endcase

   end



endmodule



////////////////////// keyboard input //////////////////////////////////
module ps2_ascii_input(clock_27mhz, reset, clock, data, ascii, ascii_ready);

   // INPUTS:
   //
   //   clock_27mhz   - master clock
   //   reset         - active high
   //   clock         - ps2 interface clock
   //   data          - ps2 interface data
   //
   // OUTPUTS:
   //
   //   ascii         - 8 bit ascii code for current character
   //   ascii_ready   - one clock cycle pulse indicating new char received
   //
   //
   // Author: C. Terman / I. Chuang
   // Date: 24-Oct-05
   // module to generate ascii code for keyboard input
   // this is module works synchronously with the system clock

   input clock_27mhz;
```

```
input reset;            // Active high asynchronous reset
input clock;            // PS/2 clock
input data;             // PS/2 data
output [7:0] ascii;   // ascii code (1 character)
output ascii_ready;              // ascii ready (one clock_27mhz cycle active high)

reg [7:0]   ascii_val;          // internal combinatorial ascii decoded value
reg [7:0]   lastkey;            // last keycode
reg [7:0]   curkey;             // current keycode
reg [7:0]   ascii;              // ascii output (latched & synchronous)
reg             ascii_ready;    // synchronous one-cycle ready flag

// get keycodes

wire        fifo_rd;            // keyboard read request
wire [7:0]  fifo_data;          // keyboard data
wire        fifo_empty;         // flag: no keyboard data
wire        fifo_overflow;      // keyboard data overflow

ps2 myps2(reset, clock_27mhz, clock, data, fifo_rd, fifo_data,
              fifo_empty,fifo_overflow);

assign      fifo_rd = ~fifo_empty;          // continous read
reg             key_ready;

always @(posedge clock_27mhz)
  begin

          // get key if ready

          curkey <= ~fifo_empty ? fifo_data : curkey;
          lastkey <= ~fifo_empty ? curkey : lastkey;
          key_ready  <= ~fifo_empty;

          // raise ascii_ready for last key which was read

          ascii_ready <= key_ready & ~(curkey[7]|lastkey[7]);
          ascii <=  (key_ready & ~(curkey[7]|lastkey[7])) ? ascii_val : ascii;

  end

always @(curkey) begin //convert PS/2 keyboard make code ==> ascii code
  case (curkey)
    8'h1C: ascii_val = 8'h41;           //A
    8'h32: ascii_val = 8'h42;           //B
    8'h21: ascii_val = 8'h43;           //C
    8'h23: ascii_val = 8'h44;           //D
    8'h24: ascii_val = 8'h45;           //E
    8'h2B: ascii_val = 8'h46;           //F
    8'h34: ascii_val = 8'h47;           //G
    8'h33: ascii_val = 8'h48;           //H
    8'h43: ascii_val = 8'h49;           //I
    8'h3B: ascii_val = 8'h4A;           //J
    8'h42: ascii_val = 8'h4B;           //K
    8'h4B: ascii_val = 8'h4C;           //L
    8'h3A: ascii_val = 8'h4D;           //M
    8'h31: ascii_val = 8'h4E;           //N
```

```
      8'h44: ascii_val = 8'h4F;                  //O
      8'h4D: ascii_val = 8'h50;                  //P
      8'h15: ascii_val = 8'h51;                  //Q
      8'h2D: ascii_val = 8'h52;                  //R
      8'h1B: ascii_val = 8'h53;                  //S
      8'h2C: ascii_val = 8'h54;                  //T
      8'h3C: ascii_val = 8'h55;                  //U
      8'h2A: ascii_val = 8'h56;                  //V
      8'h1D: ascii_val = 8'h57;                  //W
      8'h22: ascii_val = 8'h58;                  //X
      8'h35: ascii_val = 8'h59;                  //Y
      8'h1A: ascii_val = 8'h5A;                  //Z

      8'h45: ascii_val = 8'h30;                  //0
      8'h16: ascii_val = 8'h31;                  //1
      8'h1E: ascii_val = 8'h32;                  //2
      8'h26: ascii_val = 8'h33;                  //3
      8'h25: ascii_val = 8'h34;                  //4
      8'h2E: ascii_val = 8'h35;                  //5
      8'h36: ascii_val = 8'h36;                  //6
      8'h3D: ascii_val = 8'h37;                  //7
      8'h3E: ascii_val = 8'h38;                  //8
      8'h46: ascii_val = 8'h39;                  //9

      8'h0E: ascii_val = 8'h60;                  // `
      8'h4E: ascii_val = 8'h2D;                  // -
      8'h55: ascii_val = 8'h3D;                  // =
      8'h5C: ascii_val = 8'h5C;                  // \
      8'h29: ascii_val = 8'h20;                  // (space)
      8'h54: ascii_val = 8'h5B;                  // [
      8'h5B: ascii_val = 8'h5D;                  // ]
      8'h4C: ascii_val = 8'h3B;                  // ;
      8'h52: ascii_val = 8'h27;                  // '
      8'h41: ascii_val = 8'h2C;                  // ,
      8'h49: ascii_val = 8'h2E;                  // .
      8'h4A: ascii_val = 8'h2F;                  // /

      8'h5A: ascii_val = 8'h0D;                  // enter (CR)
      8'h66: ascii_val = 8'h08;                  // backspace

      //  8'hF0: ascii_val = 8'hF0;                        // BREAK CODE

      default: ascii_val = 8'h23;                // #
    endcase
  end
endmodule // ps2toascii

////////////////////////////////////////////////////////////////////////////
// new synchronous ps2 keyboard driver, with built-in fifo, from Chris Terman

module ps2(reset, clock_27mhz, ps2c, ps2d, fifo_rd, fifo_data,
           fifo_empty,fifo_overflow);

  input clock_27mhz,reset;
  input ps2c;                                    // ps2 clock
  input ps2d;                                    // ps2 data
  input fifo_rd;                  // fifo read request (active high)
```

```
 output [7:0] fifo_data;          // fifo data output
 output   fifo_empty;             // fifo empty (active high)
 output   fifo_overflow;          // fifo overflow - too much kbd input

reg [3:0] count;      // count incoming data bits
reg [9:0] shift;      // accumulate incoming data bits

reg [7:0] fifo[7:0];   // 8 element data fifo
reg fifo_overflow;
reg [2:0] wptr,rptr;   // fifo write and read pointers

wire [2:0] wptr_inc = wptr + 1;

assign fifo_empty = (wptr == rptr);
assign fifo_data = fifo[rptr];

// synchronize PS2 clock to local clock and look for falling edge
reg [2:0] ps2c_sync;
always @ (posedge clock_27mhz) ps2c_sync <= {ps2c_sync[1:0],ps2c};
wire sample = ps2c_sync[2] & ~ps2c_sync[1];

always @ (posedge clock_27mhz) begin
  if (reset) begin
    count <= 0;
    wptr <= 0;
    rptr <= 0;
    fifo_overflow <= 0;
  end
  else if (sample) begin
        // order of arrival: 0,8 bits of data (LSB first),odd parity,1
        if (count==10) begin
          // just received what should be the stop bit
          if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
                        fifo[wptr] <= shift[8:1];
                        wptr <= wptr_inc;
                        fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
          end
          count <= 0;
              end else begin
          shift <= {ps2d,shift[9:1]};
          count <= count + 1;
              end
      end
  // bump read pointer if we're done with current value.
  // Read also resets the overflow indicator
  if (fifo_rd && !fifo_empty) begin
    rptr <= rptr + 1;
    fifo_overflow <= 0;
  end
end

endmodule
```

---

makepacket.v

```
`timescale 1ns / 1ps
```

```
////////////////////////////////////////////////////////////////////////////////
// Company: keam / allanko for 6.111 fa2016 - lasernet
// Engineer:
//
// Create Date:
 // Design Name:
// Module Name: makeheader
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

// makes packets
// accepts user input of message into memory
// makes header, grabs corresponding block of input data, constructs a packet
// and calculates a checksum

// SNmax bookkept by messageinput module
//

module makepacket(
    input clk,
    input reset,
    input [31:0] ISN,
    input readyin,              // goes high to trigger generation of a new packet
    input [15:0] window,
    input [31:0] seq,
    input [31:0] ack,
    input [8:0] flags,
    input [16*8*5 - 1 : 0] message, // outgoing message data
    output reg [32*9 - 1:0] packet, // 9 octets - 4 header, 1 checksum, 4 data
    output reg readyout         // goes high when new packet is ready
    );


    wire [31:0] octet1;                         // source/dest port
    wire [31:0] octet2;                         // seq
    wire [31:0] octet3;                         // ack
    wire [31:0] octet4;                         // flags and window size
    wire [31:0] octet5;                         // checksum
    wire [31:0] octet6, octet7, octet8, octet9;     // data

    // HEADER: source, destination, seq, ack, flags. four octets.

    assign octet1 = 32'd0;
    assign octet2 = seq;
    assign octet3 = ack;
    assign octet4 = {7'd0, flags, window};
```

```
    // DATA: pick which packet to send based on current SN. four octets.

    wire [31:0] index;
    assign index = seq - ISN - 32'd1; // SN = 1 / index = 0 at first entry into S_TRANSMITTING

    wire [16*8 - 1 : 0] messagepart0, messagepart1, messagepart2, messagepart3, messagepart4; // breaking up the message into
packets
    wire [16*8 - 1 : 0] messagetosend;                          // message to send

    assign messagepart0 = message[16*8*1 - 1 : 0];
    assign messagepart1 = message[16*8*2 - 1 : 16*8*1];
    assign messagepart2 = message[16*8*3 - 1 : 16*8*2];
    assign messagepart3 = message[16*8*4 - 1 : 16*8*3];
    assign messagepart4 = message[16*8*5 - 1 : 16*8*4];

    assign messagetosend =  (index == 32'd0) ? messagepart0 :
                    (index == 32'd1) ? messagepart1 :
                    (index == 32'd2) ? messagepart2 :
                    (index == 32'd3) ? messagepart3 :
                    (index == 32'd4) ? messagepart4 :
                            128'd0;


    assign octet6 = messagetosend[32*4 - 1 : 32*3];
    assign octet7 = messagetosend[32*3 - 1 : 32*2];
    assign octet8 = messagetosend[32*2 - 1 : 32*1];
    assign octet9 = messagetosend[32*1 - 1 : 32*0];

    // CHECKSUM: complement of ones complement sum of every 16-bit word. 1 octet.

    wire [31:0] sum;
    wire [15:0] checksum;
    assign sum = octet1[31:16] + octet1[15:0] +
            octet2[31:16] + octet2[15:0] +
            octet3[31:16] + octet3[15:0] +
            octet4[31:16] + octet4[15:0] +
            octet6[31:16] + octet6[15:0] +  // skip octet 5 because that's where the checksum goes
            octet7[31:16] + octet7[15:0] +
            octet8[31:16] + octet8[15:0] +
            octet9[31:16] + octet9[15:0];

    assign checksum = (sum[31:16] + sum[15:0]) < sum[15:0] ?  ~(sum[31:16] + sum[15:0] + 16'b1) :
                                    ~(sum[31:16] + sum[15:0]);

    assign octet5 = {checksum, 16'd0};


    // STATE MACHINE

    parameter WAIT = 2'b0; // idle
    parameter MAKE = 2'b1; // make and transmit packet
    reg [1:0] state;

    always @(posedge clk) begin
      case(state)

        WAIT : begin
```

```
            readyout <= 1'b0;
            state <= !reset & readyin ? MAKE : WAIT;

        end

        MAKE : begin

            readyout <= 1'b1;
            packet <= {octet1, octet2, octet3, octet4, octet5, octet6, octet7, octet8, octet9};

            state <= WAIT;
        end

        default : state <= WAIT;

    endcase

  end

endmodule
```

---

serial_tx.v

```
module serial_tx
  ////////////////////////////////////////////////
  //
  // CLK_PER_BIT is used to set the baud rate
  // 65mhz/4800
  // baud Rates (bits/sec)
  //    4800      --   13541.667
  //    9600      --   67708.833
  //    14400     --   4513.888
  //    19200     --   3385.416
  //    57600     --   1128.472
  //    115200    --   564.236
  //    128000    --   507.8125
  //
  ////////////////////////////////////////////////

  #(parameter CLK_PER_BIT = 13540,
    parameter PKT_LENGTH = 32)
  (input clk,
  input rst,
  input [PKT_LENGTH-1:0] data,
  input new_data,
  output tx,
  output busy,
  output done
  );

  // clog2 is 'ceiling of log base 2' which gives you the number of bits needed to store a value
  parameter CTR_SIZE = $clog2(CLK_PER_BIT); //needs to be at least ceil(log2 of clocks cyless per bit)

  localparam STATE_SIZE = 2;
  localparam IDLE = 2'd0,
  START_BIT = 2'd1,
```

```
DATA = 2'd2,
STOP_BIT = 2'd3;

reg [CTR_SIZE-1:0] ctr_d, ctr_q;
reg [23:0] bit_ctr_d, bit_ctr_q;
reg [PKT_LENGTH-1:0] data_d, data_q;
reg [STATE_SIZE-1:0] state_d, state_q = IDLE;
reg tx_d, tx_q;
reg busy_d, busy_q;
reg done_d, done_q;

assign tx = tx_q;
assign busy = busy_q;
assign done = done_q;

always @(*) begin
ctr_d = ctr_q;
bit_ctr_d = bit_ctr_q;
data_d = data_q;
state_d = state_q;
busy_d = busy_q;
done_d = done_q;

case (state_q)
  IDLE: begin
      busy_d = 1'b0;
      tx_d = 1'b0;
      bit_ctr_d = 24'b0;
      ctr_d = 1'b0;
      done_d = 1'b0;
      if (new_data) begin
        data_d = data;
        state_d = START_BIT;
        busy_d = 1'b1;
    end
  end
  START_BIT: begin
    busy_d = 1'b1;
    ctr_d = ctr_q + 1'b1;
    tx_d = 1'b1;
    done_d = 1'b0;
    if (ctr_q == CLK_PER_BIT - 1) begin
      ctr_d = 1'b0;
      state_d = DATA;
    end
  end
  DATA: begin
    busy_d = 1'b1;
    tx_d = data_q[bit_ctr_q];
    ctr_d = ctr_q + 1'b1;
    done_d = 1'b0;
    if (ctr_q == CLK_PER_BIT - 1) begin
      ctr_d = 1'b0;
      bit_ctr_d = bit_ctr_q + 1'b1;
      if (bit_ctr_q == PKT_LENGTH-1) begin /////////
        state_d = STOP_BIT;
      end
```

```
      end
    end
    STOP_BIT: begin
      busy_d = 1'b1;
      tx_d = 1'b0;
      ctr_d = ctr_q + 1'b1;
      done_d = 1'b1;
      if (ctr_q == CLK_PER_BIT - 1) begin
        state_d = IDLE;
      end
    end
    default: begin
      state_d = IDLE;
    end
  endcase
  end

  always @(posedge clk) begin
      if (rst) begin
        state_q <= IDLE;
        tx_q <= 1'b0;

        data_q <= 0;
        bit_ctr_q <= 0;
        ctr_q <= 0;
        busy_q <= 0;
        done_q <= 0;

      end else begin
        state_q <= state_d;
        tx_q <= tx_d;

        data_q <= data_d;
        bit_ctr_q <= bit_ctr_d;
        ctr_q <= ctr_d;
        busy_q <= busy_d;
        done_q <= done_d;
      end


  end

endmodule
```

---

serial_rx.v

```
`timescale 1ns / 1ps

module serial_rx
  #(parameter CLK_PER_BIT = 50,
    parameter PKT_LENGTH = 32)
  (input clk,
  input rst,
  input rx,
  output [PKT_LENGTH-1:0] data,
  output new_data);
```

```
// clog2 is 'ceiling of log base 2' which gives you the number of bits needed to store a value
parameter CTR_SIZE = $clog2(CLK_PER_BIT);

localparam STATE_SIZE = 2;
localparam IDLE = 2'd0,
  WAIT_HALF = 2'd1,
  WAIT_FULL = 2'd2,
  WAIT_HIGH = 2'd3;

reg [CTR_SIZE-1:0] ctr_d, ctr_q;
reg [23:0] bit_ctr_d, bit_ctr_q;
reg [PKT_LENGTH-1:0] data_d, data_q;
reg new_data_d, new_data_q;
reg [STATE_SIZE-1:0] state_d, state_q = IDLE;
reg rx_d, rx_q;

assign new_data = new_data_q;
assign data = data_q;

always @(*) begin
  rx_d = rx;
  state_d = state_q;
  ctr_d = ctr_q;
  bit_ctr_d = bit_ctr_q;
  data_d = data_q;
  new_data_d = 1'b0;

  case (state_q)
    IDLE: begin
      bit_ctr_d = 24'b0;
      ctr_d = 1'b0;
      if (rx_q == 1'b1) begin
        state_d = WAIT_HALF;
      end
    end
    WAIT_HALF: begin
      ctr_d = ctr_q + 1'b1;
      if (ctr_q == (CLK_PER_BIT >> 1)) begin
        ctr_d = 1'b0;
        state_d = WAIT_FULL;
      end
    end
    WAIT_FULL: begin
      ctr_d = ctr_q + 1'b1;
      if (ctr_q == CLK_PER_BIT - 1) begin
        data_d = {rx_q, data_q[PKT_LENGTH-1:1]};
        bit_ctr_d = bit_ctr_q + 1'b1;
        ctr_d = 1'b0;
        if (bit_ctr_q == PKT_LENGTH-1) begin
          state_d = WAIT_HIGH;
          new_data_d = 1'b1; ////change later? idk

        end
      end
    end
    WAIT_HIGH: begin
      if (rx_q == 1'b0) begin
```

```
        state_d = IDLE;
      end
    end
    default: begin
      state_d = IDLE;
    end

  endcase

end

always @(posedge clk) begin
  if (rst) begin
    ctr_q <= 1'b0;
    bit_ctr_q <= 24'b0;
    new_data_q <= 1'b0;
    state_q <= IDLE;

    rx_q <= 0;
    data_q <= 0;

  end else begin
    ctr_q <= ctr_d;
    bit_ctr_q <= bit_ctr_d;
    new_data_q <= new_data_d;
    state_q <= state_d;

    rx_q <= rx_d;
    data_q <= data_d;
  end


end

endmodule
```

---

receivepacket.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: keam / allanko for 6.111 fa2016 - lasernet
// Engineer:
//
// Create Date:
// Design Name:
// Module Name: receivepacket.v
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
```

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

// checksumming received data and passing it to state machine / screen display as necessary
// block of memory containing received message lives here

module receivepacket(
        input clk,
        input reset,
        input ready,
        input ISN,
        input [32*9 - 1 : 0] packet,

        output reg [31:0] seq,
        output reg [31:0] ack,
        output reg [8:0] flags,
        output [16*8*5 - 1 : 0] message
        );

        wire [31:0] octet1;                     // source/dest port
  wire [31:0] octet2;                   // seq
  wire [31:0] octet3;                   // ack
  wire [31:0] octet4;                   // flags and window size
  wire [31:0] octet5;                   // checksum
  wire [31:0] octet6, octet7, octet8, octet9;     // data

  // break down the packet

  assign octet1 = packet[32*9 - 1 : 32*8];
  assign octet2 = packet[32*8 - 1 : 32*7];
  assign octet3 = packet[32*7 - 1 : 32*6];
  assign octet4 = packet[32*6 - 1 : 32*5];
  assign octet5 = packet[32*5 - 1 : 32*4];
  assign octet6 = packet[32*4 - 1 : 32*3];
  assign octet7 = packet[32*3 - 1 : 32*2];
  assign octet8 = packet[32*2 - 1 : 32*1];
  assign octet9 = packet[32*1 - 1 : 32*0];

  // CHECKSUM: complement of ones complement sum of every 16-bit word.
  // if checksum = 0, packet is intact

  wire [31:0] sum;
  wire [15:0] checksum;
  wire goodpacket;
  assign sum = octet1[31:16] + octet1[15:0] +
        octet2[31:16] + octet2[15:0] +
        octet3[31:16] + octet3[15:0] +
        octet4[31:16] + octet4[15:0] +
        octet5[31:16] + octet5[15:0] + // notice - including the checksum octet here!
        octet6[31:16] + octet6[15:0] +
        octet7[31:16] + octet7[15:0] +
        octet8[31:16] + octet8[15:0] +
        octet9[31:16] + octet9[15:0];

  assign checksum = (sum[31:16] + sum[15:0]) < sum[15:0] ?  ~(sum[31:16] + sum[15:0] + 16'b1) :
                                ~(sum[31:16] + sum[15:0]);
```

```
assign goodpacket = ~|checksum; // goodpacket = 1 only if checksum = 0

// REGISTERS FOR INCOMING MESSAGE

reg [16*8 - 1 : 0] messagepart1 = "[     blank    ]";
reg [16*8 - 1 : 0] messagepart2 = "[     blank    ]";
reg [16*8 - 1 : 0] messagepart3 = "[     blank    ]";
reg [16*8 - 1 : 0] messagepart4 = "[     blank    ]";
reg [16*8 - 1 : 0] messagepart5 = "[     blank    ]";


assign message = {messagepart5, messagepart4, messagepart3, messagepart2, messagepart1};

// STATE MACHINE FOR TRACKING PACKETS AND UPDATING OUTPUTS

parameter HOLD          = 2'b00; // hold current values
parameter UPDATE_OOO = 2'b01; // received packet out of order OR receiving control packets, update flags and ACK only
parameter UPDATE_ALL = 2'b10; // received packet in order, update flags, ACK, SEQ, and message
parameter RESET         = 2'b11; // reset state

reg [1:0] state, laststate;
reg [31:0] highestSNreceived;

wire [31:0] SNreceived;
assign SNreceived = octet2 - ISN;

wire [16*8 - 1 : 0] messagereceived;
assign messagereceived = {octet6, octet7, octet8, octet9};

always @(posedge clk) begin

        laststate <= state;

        case(state)
                HOLD : begin

                        highestSNreceived <= (!reset & ready & goodpacket & (SNreceived == highestSNreceived + 32'd1)) ?
octet2 : highestSNreceived;

                        state <= (!reset & ready & goodpacket & (SNreceived == highestSNreceived + 32'd1)) ?
UPDATE_ALL :
                                        (!reset & ready & goodpacket) ? UPDATE_OOO :
                                        reset ? RESET :
                                        HOLD;
                end

                UPDATE_OOO : begin

                        ack <= octet3;
                        flags <= octet4[24:16];

                        state <= HOLD;
                end

                UPDATE_ALL : begin

                        seq <= octet2;
```

```
                ack <= octet3;
                flags <= octet4[24:16];

                messagepart1 <= (SNreceived == 32'd1) ? messagereceived : messagepart1;
                messagepart2 <= (SNreceived == 32'd2) ? messagereceived : messagepart2;
                messagepart3 <= (SNreceived == 32'd3) ? messagereceived : messagepart3;
                messagepart4 <= (SNreceived == 32'd4) ? messagereceived : messagepart4;
                messagepart5 <= (SNreceived == 32'd5) ? messagereceived : messagepart5;

                state <= HOLD;
            end

            RESET : begin

                    messagepart1 <= "[     blank    ]";
                    messagepart2 <= "[     blank    ]";
                    messagepart3 <= "[     blank    ]";
                    messagepart4 <= "[     blank    ]";
                    messagepart5 <= "[     blank    ]";

                    seq <= 32'd0;
                    ack <= 32'd0;
                    flags <= 9'd0;

                    highestSNreceived <= 32'd0;

                    state <= HOLD;
            end

            default : state <= RESET;

        endcase
    end

endmodule
```

---

screenlayout.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:
// Design Name:
// Module Name: screenlayout
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
```

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

// lay out all the text on the screen


module screenlayout(
   input clock_65mhz,                  // 65MHz clock
   input [10:0] hcount,        // horizontal index of current pixel (0..1023)
   input [9:0]  vcount,        // vertical index of current pixel (0..767)
   input display,
   input [16*8*5 - 1:0] messageout,  // outgoing message to display
   input [16*8*5 - 1:0] messagein,   // incoming message to display
   input [16*8 - 1:0] keyboard,      // last 16 characters entered on keyboard
   output [2:0] pixels         // char display's pixel
   );

   // user 1 text start at ~107 (x-coord) ???????
   // user 2 text start at ~298 (x-coord) ????????


   // convert incoming message wires to arrays

   wire [16*8 - 1:0] messageoutarray[4:0];
   assign messageoutarray[0] = messageout[16*8*1 - 1 : 16*8*0];
   assign messageoutarray[1] = messageout[16*8*2 - 1 : 16*8*1];
   assign messageoutarray[2] = messageout[16*8*3 - 1 : 16*8*2];
   assign messageoutarray[3] = messageout[16*8*4 - 1 : 16*8*3];
   assign messageoutarray[4] = messageout[16*8*5 - 1 : 16*8*4];

   wire [16*8 - 1:0] messageinarray[4:0];
   assign messageinarray[0] = messagein[16*8*1 - 1 : 16*8*0];
   assign messageinarray[1] = messagein[16*8*2 - 1 : 16*8*1];
   assign messageinarray[2] = messagein[16*8*3 - 1 : 16*8*2];
   assign messageinarray[3] = messagein[16*8*4 - 1 : 16*8*3];
   assign messageinarray[4] = messagein[16*8*5 - 1 : 16*8*4];

   //////////  lay out all the text on the screen

   // headers: outgoing message on left side, incoming message on right side, user input at bottom


      wire [8*8-1:0] cstring_outgoing = "OUTGOING";
      wire [2:0]  cdpixel_outgoing;
      char_string_display cd_outgoing(clock_65mhz,hcount,vcount,
                              cdpixel_outgoing, cstring_outgoing,
                              11'd150, 10'd25);          // coordinates of string
      defparam cd_outgoing.NCHAR = 8;                    // number of characters in cstring
      defparam cd_outgoing.NCHAR_BITS = 4;                  // number of bits in NCHAR


      wire [8*8-1:0] cstring_incoming = "INCOMING";
      wire [2:0]  cdpixel_incoming;
      char_string_display cd_incoming(clock_65mhz,hcount,vcount,
                              cdpixel_incoming, cstring_incoming,
                              11'd550, 10'd25);
      defparam cd_incoming.NCHAR = 8;
```

```
    defparam cd_incoming.NCHAR_BITS = 4;


    wire [11*8-1:0] cstring_input = "USER INPUT:";
    wire [2:0]  cdpixel_input;
    char_string_display cd_input(clock_65mhz,hcount,vcount,
                          cdpixel_input, cstring_input,
                          11'd200, 10'd650);
    defparam cd_input.NCHAR = 11;
    defparam cd_input.NCHAR_BITS = 4;

////////////////////////////// outgoing message

    wire [16*8-1:0] cstring_out1 = messageoutarray[0];
    wire [2:0]  cdpixel_out1;
    char_string_display cd_out1(clock_65mhz,hcount,vcount,
                         cdpixel_out1, cstring_out1,
                         11'd150, 10'd25+1*50);
    defparam cd_out1.NCHAR = 16;
    defparam cd_out1.NCHAR_BITS = 5;


    wire [16*8-1:0] cstring_out2 = messageoutarray[1];
    wire [2:0]  cdpixel_out2;
    char_string_display cd_out2(clock_65mhz,hcount,vcount,
                         cdpixel_out2, cstring_out2,
                         11'd150, 10'd25+2*50);
    defparam cd_out2.NCHAR = 16;
    defparam cd_out2.NCHAR_BITS = 5;


    wire [16*8-1:0] cstring_out3 = messageoutarray[2];
    wire [2:0]  cdpixel_out3;
    char_string_display cd_out3(clock_65mhz,hcount,vcount,
                         cdpixel_out3, cstring_out3,
                         11'd150, 10'd25+3*50);
    defparam cd_out3.NCHAR = 16;
    defparam cd_out3.NCHAR_BITS = 5;


    wire [16*8-1:0] cstring_out4 = messageoutarray[3];
    wire [2:0]  cdpixel_out4;
    char_string_display cd_out4(clock_65mhz,hcount,vcount,
                        cdpixel_out4, cstring_out4,
                        11'd150, 10'd25+4*50);
    defparam cd_out4.NCHAR = 16;
    defparam cd_out4.NCHAR_BITS = 5;


    wire [16*8-1:0] cstring_out5 = messageoutarray[4];
    wire [2:0]  cdpixel_out5;
    char_string_display cd_out5(clock_65mhz,hcount,vcount,
                        cdpixel_out5, cstring_out5,
                        11'd150, 10'd25+5*50);
    defparam cd_out5.NCHAR = 16;
    defparam cd_out5.NCHAR_BITS = 5;
```

```
//////////////////////////// incoming message

   wire [16*8-1:0] cstring_in1 = messageinarray[0];
   wire [2:0]  cdpixel_in1;
   char_string_display cd_in1(clock_65mhz,hcount,vcount,
                        cdpixel_in1, cstring_in1,
                        11'd550, 10'd25+1*50);
   defparam cd_in1.NCHAR = 16;
   defparam cd_in1.NCHAR_BITS = 5;


   wire [16*8-1:0] cstring_in2 = messageinarray[1];
   wire [2:0]  cdpixel_in2;
   char_string_display cd_in2(clock_65mhz,hcount,vcount,
                        cdpixel_in2, cstring_in2,
                        11'd550, 10'd25+2*50);
   defparam cd_in2.NCHAR = 16;
   defparam cd_in2.NCHAR_BITS = 5;


   wire [16*8-1:0] cstring_in3 = messageinarray[2];
   wire [2:0]  cdpixel_in3;
   char_string_display cd_in3(clock_65mhz,hcount,vcount,
                        cdpixel_in3, cstring_in3,
                        11'd550, 10'd25+3*50);
   defparam cd_in3.NCHAR = 16;
   defparam cd_in3.NCHAR_BITS = 5;


   wire [16*8-1:0] cstring_in4 = messageinarray[3];
   wire [2:0]  cdpixel_in4;
   char_string_display cd_in4(clock_65mhz,hcount,vcount,
                        cdpixel_in4, cstring_in4,
                        11'd550, 10'd25+4*50);
   defparam cd_in4.NCHAR = 16;
   defparam cd_in4.NCHAR_BITS = 5;


   wire [16*8-1:0] cstring_in5 = messageinarray[4];
   wire [2:0]  cdpixel_in5;
   char_string_display cd_in5(clock_65mhz,hcount,vcount,
                        cdpixel_in5, cstring_in5,
                        11'd550, 10'd25+5*50);
   defparam cd_in5.NCHAR = 16;
   defparam cd_in5.NCHAR_BITS = 5;

//////////// current keyboard input

   // display current user input in the red box
   wire [16*8-1:0] cstring = keyboard;
   wire [2:0]  cdpixel;
   char_string_display cd(clock_65mhz,hcount,vcount,
                      cdpixel,cstring,
                      11'd383,10'd650);
   defparam    cd.NCHAR = 16;
   defparam    cd.NCHAR_BITS = 4;
```

```
   // display

   assign pixels = cdpixel_outgoing | cdpixel_incoming | cdpixel_input |
                 cdpixel_out1 | cdpixel_out2 | cdpixel_out3 | cdpixel_out4 | cdpixel_out5 |
                 cdpixel_in1 | cdpixel_in2 | cdpixel_in3 | cdpixel_in4 | cdpixel_in5 |
                 cdpixel
                 ;
endmodule
```

---

xvga.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 12/05/2016 09:26:42 PM
// Design Name:
// Module Name: xvga
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

// this generate XVGA display signals (1024 x 768 @ 60Hz)

module xvga(vclock,hcount,vcount,hsync,vsync,blank);
  input vclock;
  output [10:0] hcount;
  output [9:0] vcount;
  output   vsync;
  output   hsync;
  output   blank;

  reg        hsync,vsync,hblank,vblank,blank;
  reg [10:0]          hcount;    // pixel number on current line
  reg [9:0] vcount;     // line number

  // horizontal: 1344 pixels total
  // display 1024 pixels per line
  wire      hsyncon,hsyncoff,hreset,hblankon;
  assign    hblankon = (hcount == 1023);
  assign    hsyncon = (hcount == 1047);
  assign    hsyncoff = (hcount == 1183);
  assign    hreset = (hcount == 1343);

  // vertical: 806 lines total
  // display 768 lines
  wire      vsyncon,vsyncoff,vreset,vblankon;
```

```
  assign    vblankon = hreset & (vcount == 767);
  assign    vsyncon = hreset & (vcount == 776);
  assign    vsyncoff = hreset & (vcount == 782);
  assign    vreset = hreset & (vcount == 805);

  // sync and blanking
  wire      next_hblank,next_vblank;
  assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
  assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
  always @(posedge vclock) begin
     hcount <= hreset ? 0 : hcount + 1;
     hblank <= next_hblank;
     hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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

---

blob.v

```
module blob
  #(parameter WIDTH = 64,            // default width: 64 pixels
            HEIGHT = 64,           // default height: 64 pixels
            COLOR = 24'hFF_FF_FF)  // default color: white
  (input [10:0] x,hcount,
   input [9:0] y,vcount,
   output reg [23:0] pixel);

  always @ * begin
     if ((hcount >= x && hcount < (x+WIDTH)) &&
             (vcount >= y && vcount < (y+HEIGHT)))
            pixel = COLOR;
     else pixel = 0;
  end
endmodule
```

---

display_8hex.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:   g.p.hom
// Engineer:
//
// Create Date:    18:18:59 04/21/2013
// Module Name:    display_8hex

// Description:  Display 8 hex numbers on 7 segment display
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
```

```
//
////////////////////////////////////////////////////////////////////////////////
module display_8hex(
   input clk,                 // system clock
   input [31:0] data,         // 8 hex numbers, msb first
   output reg [6:0] seg,      // seven segment display output
   output reg [7:0] strobe    // digit strobe
   );

   localparam bits = 13;

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

   wire [6:0] segments[15:0]; // 16 7 bit memorys
   assign segments[0]  = 7'b100_0000;
   assign segments[1]  = 7'b111_1001;
   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 @(posedge clk) begin
     counter <= counter + 1;
     case (counter[bits:bits-2])
        3'b000: begin
                seg <= segments[data[31:28]];
                strobe <= 8'b0111_1111 ;
               end

        3'b001: begin
                seg <= segments[data[27:24]];
                strobe <= 8'b1011_1111 ;
               end

        3'b010: begin
                 seg <= segments[data[23:20]];
                 strobe <= 8'b1101_1111 ;
                end
        3'b011: begin
                seg <= segments[data[19:16]];
                strobe <= 8'b1110_1111;
               end
        3'b100: begin
                seg <= segments[data[15:12]];
                strobe <= 8'b1111_0111;
               end
```

```
        3'b101: begin
                seg <= segments[data[11:8]];
                strobe <= 8'b1111_1011;
               end

        3'b110: begin
                 seg <= segments[data[7:4]];
                 strobe <= 8'b1111_1101;
                end
        3'b111: begin
                seg <= segments[data[3:0]];
                strobe <= 8'b1111_1110;
               end



     endcase
    end

endmodule
```

---

debounce.v

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output


module debounce #(parameter DELAY=270000)   // .01 sec with a 27Mhz clock
                    (input reset, clock, noisy,
                     output reg clean);

  reg [18:0] count;
  reg new;

  always @(posedge clock)
    if (reset) begin
      count <= 0;
      new <= noisy;
      clean <= noisy;
    end

    else if (noisy != new) begin
      new <= noisy;
      count <= 0;
    end

    else if (count == DELAY)
      clean <= new;

    else
      count <= count+1;

endmodule
```

---

synchronize.v

```
// pulse synchronizer

module synchronize #(parameter NSYNC = 2)  // number of sync flops.  must be >= 2
                   (input clock,in,
                    output reg out);

  reg [NSYNC-2:0] sync;

  always @ (posedge clock) begin
              {out,sync} <= {sync[NSYNC-2:0],in};
  end

endmodule
```