

FPGA: HFT Exchange Ethernet Packets Handling and FPGA Setup Process Documentation

Abstract: This project explores the implementation of a high-frequency trading (HFT) strategy on a Xilinx Kintex-7 FPGA using the DSLX High-Level Synthesis toolchain. Our team designed and synthesized a finite state machine (FSM) capable of parsing Ethernet packets, performing risk checks, and generating trading orders in real time. The system handles packet formatting, UDP payload management, and state transitions entirely in hardware to maximize parallelism and speed. We document the full development process, from configuring Vivado on student systems to integrating DSLX-generated Verilog into FPGA hardware. Challenges related to tool setup, module interfacing, and RTL integration were addressed through extensive debugging and custom wrapper modules. This work serves as a foundation for future projects aiming to interface FPGAs directly with trading systems and exchanges.

Brendan McGinn, Nick Palma, TJ Weber

CSE40438: High-Frequency Trading Technologies

Professor Matthew Belcher

April 30, 2025

1 Project Proposal / Overview

For our final project, our group chose to take on the challenge of implementing a trading strategy for the class trading program using the provided FPGA. As part of their High-Level Synthesis course, Brendan and Nick gained access to the newly available Xilinx tools installed on the student machine *student04*. This allowed us to leverage the Vivado software from that class to streamline development for our High-Frequency Trading project.

1.1 Project Goals

In the High-Level Synthesis course, Professor Morrison's work with DSLX tools aligned closely with the topics we were exploring in High-Frequency Trading, giving our group a unique opportunity to get a head start on the learning curve involved in setting up an FPGA-based trading system. This led us to define two primary goals for our final project:

- ❖ Implement a high-frequency trading strategy on an FPGA board using the DSLX toolchain: effectively writing parsing code and synthesizing to Verilog.
- ❖ Document the setup and usage of the Xilinx tools: supporting both educational use and troubleshooting by future students.

1.2 Documentation

The deliverables included in this paper are as follows:

- ❖ A verified walkthrough of Professor Cong Hao's HLS labs (developed for her HLS class at Georgia Tech) in order to give a basis for transferring strategies from higher-level code to the low-level code like Verilog needed for the FPGA. This document will include our step-by-step development and implementation process, while also identifying potential

issues, and offering clarifications to help Notre Dame students avoid similar obstacles if they intend to follow our footsteps in implementing a trading strategy on the FPGA.

- ❖ A detailed guide on implementing an HFT strategy on the Kintex-7 board. This will cover how to use Vivado on a remote machine to deploy changes to an FPGA board connected to a local machine. The goal is to enable students to replicate the setup and implementation process for their own designs.

2 Trading System FSM Overview:

On the Xilinx Kintex-7 board, our group will be implementing a basic high frequency trading system. The system will have four main tasks:

- ❖ Read and format incoming ethernet messages to be acceptable to the user module.
- ❖ Parse the incoming packets from the exchange, registering information from the order.
- ❖ Fill in the data section of the order, potentially by receiving information from UDP packets sent by the software side of the trading system (already built) or potentially simply sending certain data from parsing, such as sequence number.
- ❖ Evaluate risk conditions and decide whether to send the order or cancel the send.

To receive the maximum benefit from utilizing an FPGA, all four of the processes must occur in parallel. It also must comply with the outline of an ethernet protocol as shown in Figure 2.1.

Preamble	Start frame delimiter (SFD)	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32-bit CRC)	Interpacket gap (IPG)
7	1	6	6	(4)	2	42–1500 ^[c]	4	12
(not part of the frame)		← 64–1522 octets →						(not part of the frame)

Figure 2.1 Ethernet Protocol

3 Implementation Process of the FSM

3.1 Parsing the Packet

There are 6 states in this particular step. The first state is IDLE, where the FPGA will spend most of its time. There is 1 input into the IDLE state: an 64-bit data word. If that data word matches the start of the Ethernet preamble and the Start of Frame (0x555555555555555d), the system will move into DELAY state. Otherwise it will continue to loop in the IDLE state.

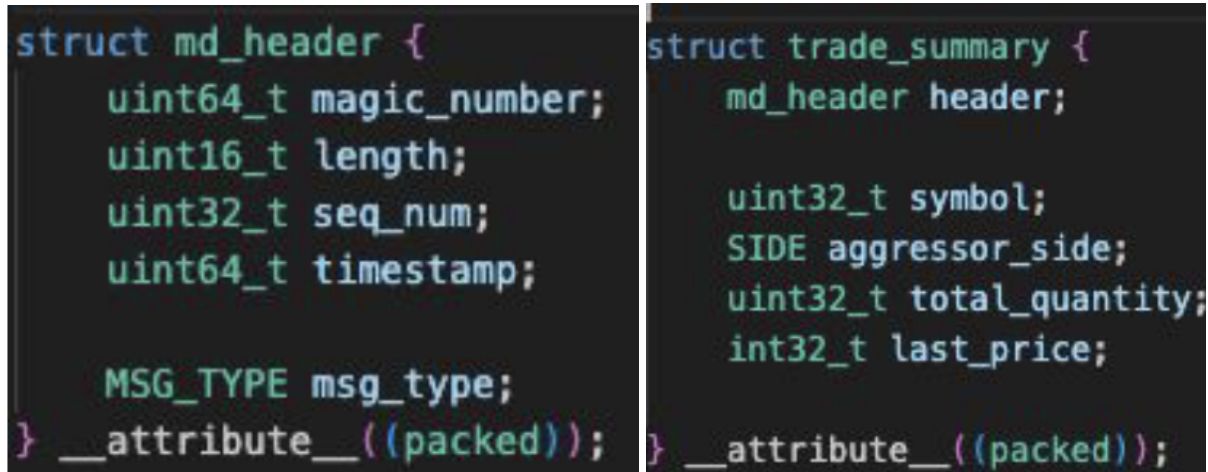
The DELAY state is used to begin to set the flip flop registers used to gather key information from the incoming packet, namely MAC source and destination. It will save these later so they may be used later to create the outgoing Ethernet packet. In order to ensure all of the information is properly gathered, there is a count variable iterated through each clock cycle. Once it reaches its 3rd clock cycle, the system moves into the PARSE1 state.

The PARSE1 state continues to read while saving the MAC destination and source to the proper registers. It is reading the IP and UDP addresses into the flip flop while it does this. This is a quicker state, and only takes a single “system” clock cycle, which is 8 clock cycles from the board.

Next, there is DELAY2, which operates very similarly to DELAY. It uses a counter to continue to read in information, now about the UDP payload, which has the read information. While reading in the payload, it will save the IP header and UDP header, as some of that information will be needed in the build.

Once the counter iterates twice, the system moves to the PARSE2 state. This state gets certain important elements, such as trade type, quantity, and price, to be used later in risk management. It also saves sequence number and other important information that is needed to

send a complete packet to the exchange. During this time, it also receives the Frame Check Sum into a flip flop register. It is essential that each piece of information from the payload (Figure 3.1) is correctly parsed to get an accurate response from the system.



```
struct md_header {
    uint64_t magic_number;
    uint16_t length;
    uint32_t seq_num;
    uint64_t timestamp;

    MSG_TYPE msg_type;
} __attribute__((packed));

struct trade_summary {
    md_header header;

    uint32_t symbol;
    SIDE aggressor_side;
    uint32_t total_quantity;
    int32_t last_price;
} __attribute__((packed));
```

Figure 3.1 Incoming UDP Payload

The final step necessary to the parsing of information is the RISK check. This step ensures that the information received from the ethernet packet is valid through an FCS number, which will set the “frame” variable. If this is not set to “true”, the system will go into the CLEAR state, where all registers are reset and the state returns to IDLE.

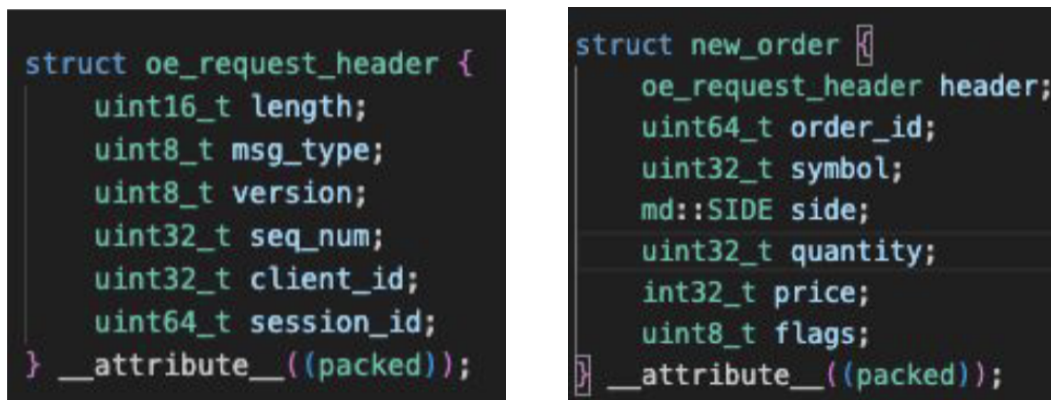
3.2 Building the Send

The START state, which begins the send process, has two main functions. The first is to ensure that the information received aligns with a “trade” type. For the simplicity of the build, the system is only trading after it receives a notification that a transaction has occurred. Any other circumstance leads to the system going into the CLEAR state. If a “trade” value is found, the Ethernet preamble and SOF are sent.

After the preamble and SOF are sent, the MAC destination and source are next. In the MAC state, the previously saved destination and source addresses are gathered from their registers, swapped to mark the send coming from the board, and sent out. Both are only 6 bytes, so the Ethernet type is sent alongside them to complete the 64-bit words.

Once the MAC and type are pushed, the UDP payload begins. The payload begins with the UDP and IP headers. Using the data previously kept in registers, the HEADERS state begins the process of building the UDP payload with both the IP and UDP headers.

Lastly, the payload is built to match what the exchange is expecting, as shown in Figure 3.2. For the original build, the same price and quantity received is sent back in the order. Otherwise, though, sequence number and order ID are updated to match what the current values should be. This is the final piece of the build before the FCS.



```
struct oe_request_header {
    uint16_t length;
    uint8_t msg_type;
    uint8_t version;
    uint32_t seq_num;
    uint32_t client_id;
    uint64_t session_id;
} __attribute__((packed));

struct new_order {
    oe_request_header header;
    uint64_t order_id;
    uint32_t symbol;
    md::SIDE side;
    uint32_t quantity;
    int32_t price;
    uint8_t flags;
} __attribute__((packed));
```

Figure 3.2: Outgoing UDP payload

3.3 Wrapper to Build Incoming and Outgoing Bytes

The FSM expects an incoming 64-bit word on each clock cycle to correctly parse, build and send packets. It works in these increments, as each of the flip flop registers hold all 64-bits at a time while the different states parse them. However, the input to the user module from the FPGA is only one byte per clock cycle. Furthermore, the pins on the FPGA only receive a half-byte at any given clock cycle. This created some slight timing issues, but 2 separate wrappers were implemented to fix them.

First, within the FSM, the state only changes every 8 clock cycles. To ensure this, there is a 4-bit value that must be set to 7 before any parse or build can occur. Every time the value iterates, the flip flop registers shift by 2, allowing space for the newly received byte to enter. After 8 cycles, the data has completely flopped, and is ready to navigate through the system again. A similar process is used on the outgoing messages. A 64-bit register takes the top byte and sends it, while shifting the other values forward. This way an outgoing byte is sent on every clock cycle in the correct order.

The second wrapper occurs outside of the user module. Instead, it works with the FPGA pins. The FPGA is reduced GMII (RGMII), which only takes one half-byte at a time. However, a module was implemented that allows the pin to work at 2x the clock cycle of the rest of the board. This module will collect 2 half-bytes and combine them on the FPGA clock cycle. This allowed the user module to bypass the limited input from the pins.

3.4 DSLX and High Level Synthesis Usage

The entire finite state machine was created using DSLX. DSLX is a software based RTL that allows for processes to be easily designed and synthesized into working Verilog. There are three main tasks for the High Level Synthesis project to work.

First, the actual FSM needs to be written in DSLX. The process and components of that piece are described in sections 3.1, 3.2, and 3.3. To write success systems, the inputs need to be passed and described in each state, which becomes an iterative process. The goal is to limit the amount passed if possible, so the original FSM only had the state, the three flip flop registers, an outgoing 64-bit word, and the counter necessary to some states. As the machine grew, certain elements were added, such as the delay counter, the order ID, the frame check, and the array of saved registers. Meanwhile, other unchanging factors, such as Ethernet Type and message type, were placed as predefined values. This way, they can be accessed when needed for the send without sending it through each state.

Second, the procedure needs to be built. This is where the transition for each piece of data is defined. For instance, this is where the flip flop registers are shifted to align with the correct information needed by the datapath. In the procedure, the initial “INIT” values are set, which for this system sets all variables to 0. Lastly, the procedure sets the input and output channels. These channels are one way, FIFO paths that connect the user module to the overall FPGA. They are set on a certain “tok” token, which occurs on every clock cycle.

The third and final step of the High Level Synthesis involves creating test benches, with a design shown in Figure 3.3. The overall process of this is creating further procedures that have predetermined input values. Those predetermined inputs must match the output from the FSM system. To determine whether they match, there is a terminator boolean that is set through an

“assert_eq” statement. While that terminator is marked “true”, the process continues to run until the process is completed. If not, it will throw an error and interrupt compilation, allowing the user to fix the issue. Each assert statement needs to be manually inputted, which resulted in over 1200+ lines of code for just test benches in this system.

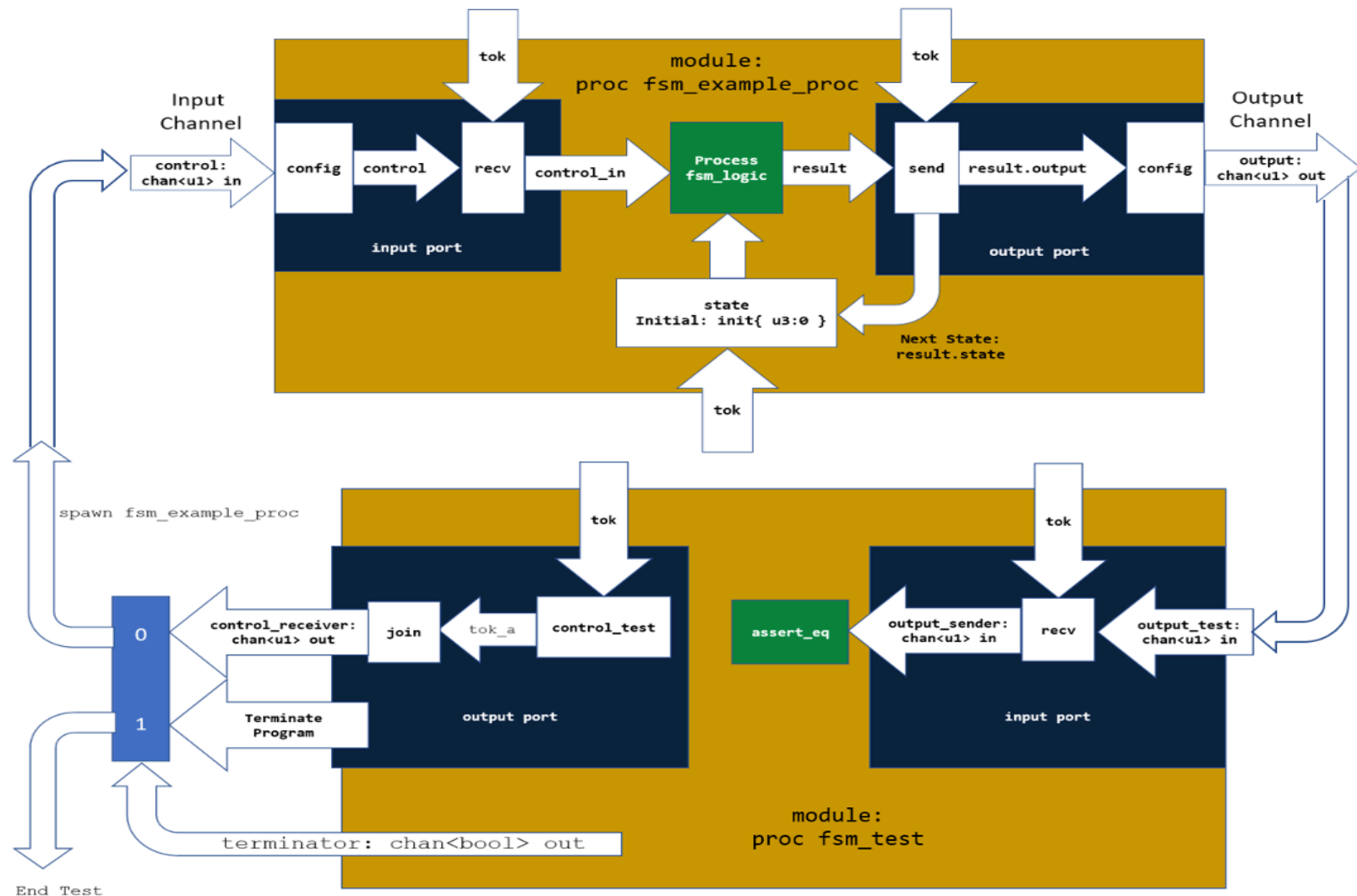


Figure 3.3: Testbench Design

4 Challenges and Lessons Learned

We faced a myriad of challenges during this project, but they can mostly be broken down into 2 categories: FPGA boot up and RTL integration. Beyond that, our issues were slight FSM creation errors that were expected and easily debugged. One example of this was the previously mentioned mismatch between input from the board and input to the user module. Simply creating a wrapper solved this issue.

4.1 FPGA Boot-Up Challenges

FPGA boot up was a difficult task, as this board was new to the Notre Dame environment. Our first order of business was finding a computer to run Vivado, which was needed for the Kintex-7 board. The original idea was to use a PC set aside for us in the HFTT class office. However, when trying to install Vivado, some errors occurred that caused us to need to reboot the Ubuntu OS. This process was trying, but eventually successful. Still, due to partition and license errors, we were unable to get Vivado working on that PC. The next idea was to install Vivado on Nick's computer. Again, this was unsuccessful as Vivado was far too large. The solution to this was to download Vivado Labs, but that requires a bitstream to already be generated. Lastly, Professor Morrison allowed us to use *student04*, which recently had Vivado put on it, and the license he has for it. This helped us immensely, and led to us being able to access Vivado after about a month's delay.

After this success, we still had a couple issues with getting the FPGA set up. First, we needed to be able to program the Kintex-7 board. *Student04* was being accessed virtually by us, so we could not physically attach the board to program it. The work around for this was to generate bitstreams through Vivado tools on the virtual machine, download that to Nick's computer, and then use Vivado Lab to upload the bitstream. The second issue was configuring

Vivado on *Student04*. Since it was recently installed, there were several PATH and license issues that needed to be resolved. Ultimately, we used Professor Hao's labs to ensure the installation had worked properly. The process of debugging the Vivado set up and learning how to program the board took us about two weeks. Both of these issues being resolved permitted us to begin using some demos from the board's manufacturer. The first step was to get flashing LEDs. Once that was done, we could shift to integrating the synthesized RTL on the board.

4.2 RTL Integration and Debugging Challenges

The integration of the DSLX generated Verilog and the board's preset modules was a very large challenge. One issue we found was with the board's controller and UDP modules. For instance, the UDP modules only allowed for 32-bit payload, which is significantly smaller than the payload needed to send an order. With more time, we most likely would have built our own with larger capabilities, but for the time being we are only sending the sequence number when parsed. Another issue is with the handling of ARP requests. To validate the ethernet connection, there needs to be an ARP request in the beginning to establish the MAC addresses. Then, there are periodic ARP requests to keep the connection alive. For a significant amount of, the controller would ignore the ARP requests when the user module was being implemented. This meant the connection was never formed, and therefore nothing could be read or sent. To fix this, we put the output of the user module to dummy wires, which we can store and send when no ARP requests are being detected.

On the Verilog side of integration, we had some issues with getting input into the module, and difficulty troubleshooting. The generated Verilog is quite compact and very difficult to understand. This becomes an issue when trying to figure out the issue with the input. Professor Morrison was able to help us find the issue with the input, which had to do with correctly setting

the flags to allow input and output to flow. With the written DSLX, there were no flags for i/o ready and valid, but when generated these flags were created. Originally, we were not setting them, which created the issue. Once those were fixed, we had some slight timing debugging to do, but we were able to manage these. We used some dummy FSMs, with unique inputs to find issues, such as not correctly getting the ethernet preamble and SOF. By setting flags and using some debug tests, we completed the integration.

We used trial and error to fix a lot of the problems we faced. With FPGAs, all of the simulations could work perfectly, but the actual implementation goes horribly wrong. With having to first set up Vivado, then do our integration, we encountered a lot of errors to be debugged. The first of two main takeaways we have would be to do a more intensive inventory at the beginning. Some issues could have been avoided if we were more familiar with the board, aware of Vivado being set up on *Student04*, and more familiar with the generated Verilog. If we had been more thorough, we likely would have saved some time and been able to do more. However, parts of that still would not have been easy, as the manual and comments from the manufacturer for the board came in Chinese. A second key lesson is not getting too far ahead of ourselves. After we could get a couple demos on the board, we immediately tried to put in our entire user module. We likely should have tried to implement only certain parts, to make troubleshooting easier. By the end, we created modules to do similar tasks to what our user module does to see what was working and fix what was not. If we had started with that, we likely would have finished sooner. Overall, with an FPGA project, we expected many challenges and to find solutions. Being able to work through these issues is great practice for working with FPGAs in the future.

5 Professor Hao's Lab Instructions

This section is used to show how we debugged the labs given by Professor Hao. Following these steps will allow for the labs to be completed successfully.

5.1 Lab 1

The first problem we ran into following the lab steps was a bad reference to an include file `"/Vitis_HLS/2024.2/include"` which is a path that does not exist on the student machines. In the Makefile, we changed a few variables to reflect correct paths. The changes follow:

```
AUTOPILOT_ROOT := /escnfs/home/csesoft/xilinx/2024-vitis-vivado/Vitis_HLS/2024.2
```

```
IFLAG += -I "/escnfs/home/csesoft/xilinx/2024-vitis-vivado/Vitis/2024.2/include"
```

Then, further PATH updates were needed, so we added the following line to `~/.bashrc`:

```
PATH=$PATH/escnfs/home/csesoft/xilinx/2024-vitis-vivado/Vivado/2024.2/bin/:/escnfs/home/csesoft/xilinx/2024-vitis-vivado/Vitis_HLS/2024.2/bin/:/escnfs/home/csesoft/xilinx/2024-vitis-vivado/Vitis/2024.2/bin/:
```

After these changes, running the commands indicated in the readme allowed us to successfully run the lab and generate reports. This step took about ten minutes to run.

5.2 Lab 2

Attempting to run the makefile for Lab 2 results in various errors and warnings about missing files and unused parameters. These seem to be related to attempting to use an incorrect compiler. We had to create an edited version of the include file utilized by the lab 2 makefile that comments out certain lines. Specifically, we commented out all `#elif_has_warning` lines in each included file. After editing the file, we changed the makefile to reference our updated version:

```
IFLAG += -I "PATH TO PERSONAL/include"
```

Additionally, it seems that earlier student machines do not support identifying the compiler used. This means that student04/06 require commenting out many more lines in these include files, as opposed to student11 which was able to successfully run with our updated include file. Once these changes were made, we successfully ran the lab and generated reports using the same commands from the readme for the first lab.

5.3 Lab 3

Lab 3 did not require additional changes after fixing the issues with the first two labs. We were able to successfully run this lab and generate reports without additional debugging.

6 Process to use Kintex-7 board

The current implementation we have running on the Kintex board follows the general outline detailed below. A more comprehensive understanding of the verilog running on the board comes from the comments left in the actual verilog files of the project.

To begin, the board is constantly monitoring the data coming across on its ethernet connection. It receives this data in 4 bit words according to the rgmii standard. Similarly, the board also sends out data in 4 bits words according to the same standard. To make data processing easier, the verilog module gmii_to_rgmii buffers data both for input and output to create 8 bit words for the rest of our modules to use and split the 8 bit words that our other modules create to properly transmit out. This creates a need for different clock speeds within our design, as if only 4 bits are read every clock cycle 8 bit words cannot be read every cycle of that same clk. Fortunately this clock scaling is addressed by modules clk_wz_0 and IDELAYCTRL which were included in the PUZHI demo project which we used as the our starting block and have not modified in any way NOR SHOULD THESE MODULES BE MODIFIED IN THE FUTURE.

Once 8 bit words are generated, the packets are processed in parallel by the logic developed in modules udp and arp. The udp module is responsible for processing the reception and transmission of true udp packets while the arp module is responsible for the reception and transmission of arp request packets. Depending on which type of packet has actually been received data routing and output from both these blocks is filtered by the net_ctrl module's logic. While our group was not focused on the arp request processing, it is important to note that this section cannot be discounted. Without proper reception and response any signals intended for the board will no longer be sent due to an unverified connection. Given our time constraints we focused on the processing of true udp packets when developing our trading strategy, but a true fully functioning trading strategy would have to properly interact with these arp requests as well. Upon a valid udp packet reception, the udp_rx block will read in the packet 8 bits at a time parsing through and saving critical information as well as outputting the packet payload 64 bits at a time. For our project this block no longer plays a role, but has been left in for troubleshooting purposes and to show how to properly parse through the udp packet. Also receiving every 8 bit word is our trading strategy encompassed by the module e1less_module (strangely named but our most recent module under test in a long line of verilog modules generated by DLSX as we have been debugging the develop strategy). This demo currently waits in and idle state to receive an 8 bit trigger '0x5d' before stepping through the rest of the trading systems states outputting a number corresponding to its current state as it does so.

The outputs generated by e1less_module are stored in a large 2048 bit buffer through the module user_process which then waits to receive a flag from the udp_tx module requesting the data to be sent as the payload. Once this occurs, the user_process module will send udp_tx the data in the buffer 8 bits at a time until the entire payload has been transmitted. Udp_tx, as mentioned before, sends each 8 bit word to be transmitted to the gmii_to_rgmii module, which breaks it into the 4 bit words that can be sent by the board out across the ethernet.

This current running demo proves that our strategy can run through its intended states acting accordingly at each. However, it does not achieve the goal of transmitting the intended udp packet info, which would be the logical next step if the project continues. Currently, transmission is also dictated by the `udp_rx` module which expects to begin transmitting a udp packet back out to sender immediately after receiving an input creating timing issues with our strategy which processes slower than this resulting in seemingly odd output.

[illegible]

This pads the trigger word with enough trailing content so the reception of the packet takes long enough for the buffer to properly load in before `udp_tx` begins requesting a payload.

7 Conclusion and Next Steps

7.1 Summary of Achievements and Learning Outcomes

Creating an ethernet parsing system from an exchange on the FPGA Kintex-7 was a difficult, but worthwhile project. Through the process of creating the system, we were able to get hands-on experience both with using High Level Synthesis principles for creating a working, complex Finite State Machine and procedure, and with the process of setting up, debugging, and integrating a new FPGA. The efficiency of using DSLX rather than hand writing Verilog became instantly apparent through our first iterations, and even more important while creating test cases. Understanding the full start to completion of an FPGA project and board was a new experience for us that will be paramount in our future. This project gave us interdisciplinary knowledge in working with FPGAs, through the fields of High Level Synthesis and High Frequency Trading.

7.2 Immediate Future Goals

Given time, we see this project going even further. There are two main areas of expansion: interacting directly with the NIC and the exchange and interfacing with a trading system. Both of these projects would require that the user create their own controller and UDP modules for the Kintex-7 board (or find a working pre-made one) rather than using the demos, as the demos do not have the necessary complexity to handle the data packets from the exchange. Once that step is completed, the board will be able to handle actual exchange interfacing. From there, connecting with a pre-built trading system would be the next step.

7.3 System Integration Vision

The trading system and board would need to have a separate connection. Information from the trading exchange would come in through the same input channel. It would likely carry information about what prices to buy at, quantity to buy at, risk checks, etc. The board would hold this information in several registers, and would use the information to build new orders and to decide to send new orders. This would allow for a more sophisticated trading system that can update with market data the software trading system receives. To make this successful, there would need to be further parsing with the UDP payload, where the “magic number” and IP address are checked to see where the information is coming from and what actions to take. The trading system would build a “magic number” specific to each request, such as set trade price or hold all trades, and the board would then update those registers. The original parsing FSM would continue to work the same, only with more detailed strategies. There would also need to be a separate output solely for the trading system. This output channel would send a log of trades made as well as verification that values were updated when sent to the board. Creating interaction between the board and a software trading system is a viable goal that would allow for actual trading strategies to be implemented.

7.4 Long-Term Vision: NIC and Exchange Integration

The next possible goal for the future would be creating a connection with the NIC and exchange. The NIC (Network Interfacing Card) is the hardware that allows for the FPGA to interface with the switch for the exchange network. In order for this to work, we would create a new begin module that connects with the NIC port first. From there, we would need to send login information to the exchange and parse through the response for necessary information, such as client ID and session ID, for trading. Once found, they would be saved to registers to be used

later when building orders. The FPGA would also likely need to send this information to the software trading system. Furthermore, at the end of trading, the FPGA would have to close the connection gracefully after receiving a command from the software trading system, which is why this would be implemented last.

Interfacing with the software trading system and connecting to the exchange would allow for the Kintex-7 board to be fully used for trading with the HFTT exchange. The speed of the board would be exponentially faster than the software systems being used. The process of creating this system gave us invaluable experience with High Level Synthesis principles, evaluating with an ethernet packet from an exchange, debugging and setting up an FPGA from scratch, and the implementation of parallelism within an FPGA system.