A Low-Latency FPGA-based Infrastructure for High Frequency Trading Systems

Andrew Boutros, Brett Grady, and Mustafa Abbas
Department of Electrical and Computer Engineering
University of Toronto
Emails: {andrew.boutros, bgrady, mustafas.abbas}@mail.utoronto.ca

I. Introduction

Modern financial exchanges are increasingly dominated by automated, algorithmic trading of securities. A central exchange broadcasts trading activities to trading machines over a network, which can then issue ”orders” based on market conditions. A large portion of these automated trades are based on identifying and exploiting the market spread which is the transient differences between top asking price and bidding price at very high speeds. These systems are referred to as High Frequency Trading (HFT) systems [1].

Due to the importance of low latency in arbitrage, HFT traders and hardware vendors have been in an arms race to lower total round-trip latency, also commonly referred to as *time to trade*. Typical high-end microprocessor systems reach a floor latency of a few microseconds [2]. On the other hand, FPGA-based systems can achieve far lower latencies reaching 1μs including with less jitter [3]. This latency advantage is critical for all algorithms but particularly arbitrage-based trading strategies, and justifies the relative complexity of designing a hardware FPGA-accelerator for such applications versus a software-based microprocessor systems.

One of the aspects of HFT that hardware based solutions excel at is rapid, jitter free encoding/decoding of the financial data streams. Typical financial exchanges broadcast multiplex financial updates along one Ethernet connection at typical line rates of 10Gb/s [3]. These are usually compressed in a domain specific format to save on bandwidth; a prominent example is FAST (FIX Adapted for STreaming), which is an adaptation of FIX (Financial Information Exchange) [4]. Many HFT hardware firms provide FPGA IP to filter high speed network traffic, decode/encode data, and perform preprocessing [3]. This allows a trading firm to customize the FPGA with their proprietary algorithms without having to implement an entire system.

Our project design is a complete system that provides an abstract view of the market state, offloading the network stack, financial protocol decoding/encoding and order book keeping to latency optimized hardware. As the system receives orders over Ethernet, the order book module sends updates to the custom application layer if the best bid or ask changes. Outgoing orders are encoded into FAST format and sent through the network. A simple proof-of-concept trading algorithm is used downstream of the order book for demonstration which can be replaced by the traders' proprietary algorithms. Overall round-trip latency is measured as 563ns using on-board monitoring.

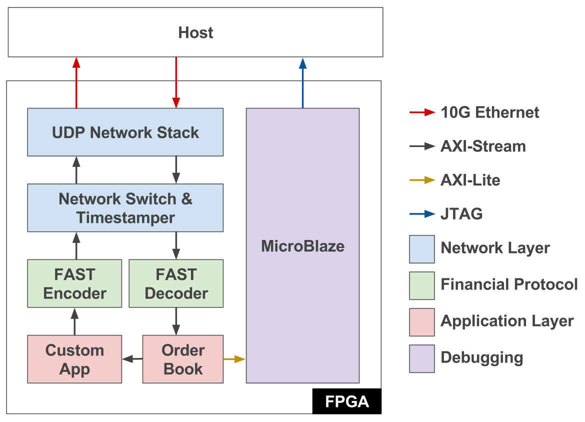

Fig. 1. HFT System Block Diagram

II. Current Status

In this project, we developed a complete prototype of an FPGA-based HFT system as depicted in Fig. 1. The development of this project was two-fold: the host side acting as the financial exchange and the hardware-accelerated HFT system on the FPGA. The host uses a python script to send market orders to the FPGA over the network. The orders would stream in from the network to be decoded into meaningful fields that are sent to the order book module. The order book keeping block then sends the current best bid and ask to the application layer, which could then issue an order according to the trading algorithm to be encoded and sent to the exchange.

A. Implemented Design Overview

On the FPGA side, the Network layer of the system was implemented using an off-the-shelf UDP Network stack from Xilinx which was used in [5] along with a network switch and a timestamper that attaches to each of the received packets a 64-bit time-stamp indicating its arrival time which is used to measure the round-trip latency of the system.

The FAST decoder/encoder is used to change the financial messages sent over the network into a meaningful data structure to be used in the order book keeping block. This data structure is composed of four mandatory fields in the current system which are: order ID which is a unique ID specified by the exchange for each order, order size in hundreds of units, the unit price in dollars, and the order type whether it is a bid or ask.

The order book keeping is implemented using a heap-like structure in which the bid book is a 4096-entry max heap that keeps track of the highest buying price in the market and the ask book is another 4096-entry min heap that keeps track of the lowest selling price in the market. The book keeping process is considered an essential pre-processing step for almost all financial trading algorithms that mostly focus only on the current best bid and ask.

In order to measure the round-trip latency of the system, we implemented a simple application layer that sends an order once the buying or selling price hits a specified threshold. This block adds negligible latency and allows us to accurately measure the latency of our system keeping in mind that it can be easily substituted by the trader's custom proprietary financial algorithm which is beyond the scope of this project.

Since the system only reacts when the best bid or ask price hits the specified threshold, it was necessary to be able to monitor the state of the system. Therefore, we used a MicroBlaze and a few simple modules to act as a system heartbeat monitor by reading the current best bid and best ask order IDs from the order book keeping block over an AXI-Lite interface and printing them to the console over the JTAG connection.

On the host side, we implemented a python script that sends and receives network packets to and from the FPGA and performs a software emulation and visualization of the current state of the order book. This was very useful for monitoring and debugging the system during the integration phase of the project and also verifying correct functionality by comparing it to the system state monitored by the MicroBlaze.

B. *Future Improvements*

There is room for improving the current system by adding more features and capabilities, some of which are quite straight-forward while others are somewhat complicated. One of the simple enhancements to the system is supporting trading more than one type of securities. This could be done by adding multiple order book keeping blocks with multiplexing and de-multiplexing logic between them and the FAST decoder/encoder. Another simple enhancement is to support more than one FAST decoding/encoding template, however, this requires knowledge of the used templates in different exchanges which is, most of the time, propriety to the exchange and its registered members and would require paying fees to get access to it. One more advanced enhancement is the ability to off-load the less-priority contents of the order book when it is full or almost full to off-chip DDR memory and restore them later when there is enough space available in the order book. This will require adding a DDR Memory Controller block to the system as well as extra logic to keep track of the order book capacity and determine when to perform this off-loading.

Generally, we believe that the ultimate goal is to make FPGAs an accessible platform for trading algorithms' software developers. Therefore, future work can incorporate the designed infrastructure into a software flow that given a C/C++ financial trading algorithm, automatically generates an IP core for it and produces a complete FPGA platform.

III. Initial Architectural Design

The project specification evolved from a simple idea: to apply HLS techniques and FPGAs to financial trading applications. The first phase the project was understanding the prior work in HFT and basics of computerized financial exchanges so that we could determine a viable project scope. The scope of the project was then narrowed down to building back-end infrastructure that gives an abstract view of the market. This would allow multiple trading algorithms to be developed without needing to redesign the low level interface. The initial specs of the system can be briefly summarized in the following six points:

- Implement a custom low-latency UDP/IP layer that does not contain any features that are unnecessary for the HFT system targeting to minimize the round-trip latency of the system as much as possible.
- Support a real-life financial protocol as much as possible and since there is little standardization across financial exchanges in this regard, we chose to implement the FAST financial protocol for decoding and encoding as it has superior publicly available documentation and is being used actively in prominent exchanges, such as the NYSE and Nasdaq [4].
- Implement an order book which maintains a sorted list of 1024 bid and ask orders. After considering several hardware architectures, like the ones presented in [6], we chose to implement an up-down heap-based structure as it scales well to large numbers of entries and does not require binning inputs into categories.
- Implement a very simple application layer to be able to close the cycle and measure the round-trip latency of the system since implementing a complex trading algorithm is out of the scope for this project.
- Include a MicroBlaze soft-processor to measure the system's round-trip latency and communicate with the host over PCIe as shown in Fig. 2.
- Target a round-trip latency of 1μs and 200 MHz clock frequency.

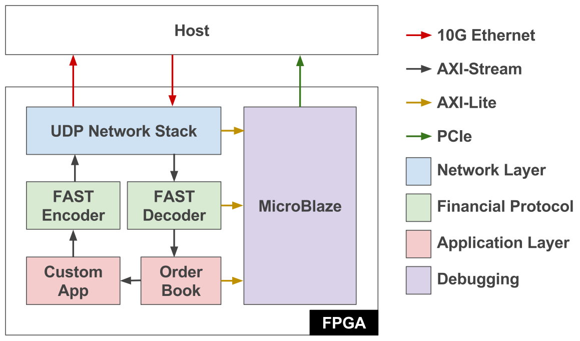

Fig. 2. Original System Specification

IV. Specification Evolution and Final Architectural Design

Upon receiving feedback about the initial system specs explained in the previous section and as we went further into the implementation of the system, some of the planned specs were modified as follows:

- Since network acceleration on FPGAs is already a mature topic and that "reinventing the wheel" here would lead to minimal gains, we decided on using already available IP cores from Xilinx to implement the UDP/IP layer.
- We followed the FAST specification for fields as closely as possible, including its unusual Base-10 floating point price format. Due to a lack of publicly available templates, we created our own template based on [7] as explained in Table I. We optimized our decoder/encoder for this specialized template (with pre-determined fields) for performance reasons; we assume commercial reconfigurable systems would take a similar approach.
- We figured out that using a MicroBlaze soft-processor to measure the system round-trip latency might result in inaccurate measurements due to the added overhead of communication between the MicroBlaze and other blocks, therefore we add a hardware-based mechanism for measuring the round-trip latency which will be explained in details later in this section. In this case the MicroBlaze would be used to monitor the state of the system by reading the current best bid and ask order IDs and printing them to the console over JTAG.
- We were advised by an industry expert that a capacity of 1024 for the order book might not be adequate, so we extended the order book to hold 4096 bids and 4096 asks. We also figured out that to implement a fully functional order book we need to add more capabilities than just a priority queue such as: arbitrary order deletion and multiple orders deletion which added a lot of complexity to the design.

For the rest of this section, we are going to give a detailed description of the architecture of the different blocks in the final implemented system which are: the network layer, the FAST decoder/encoder and the order book.

Field	Byte Offset	Size (Bytes)	Field Value
presence map	0	1	7b unsigned
template id	1	1	7b unsigned
price (exponent)	2	1	7b signed
price (mantissa)	3	1-5	7b-34b signed
order quantity	4-8	1-2	8b signed
order ID	9-13	1-5	32b unsigned
order type	14	1	2b unsigned

TABLE I. FAST Protocol Template Specification

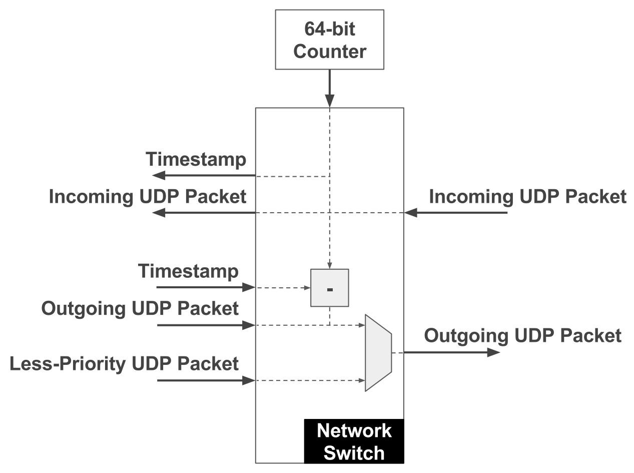

Fig. 3. Block Diagram of Network Switch

A. Network Layer

Financial Exchanges are typically built on top of commodity networking hardware and protocols. The physical layer is typically one of the various Ethernet standards (i.e. IEEE 802.3ae-2002); the transport and internet layers are typically UDP/IP [2], [3]. UDP is preferred over other alternatives because it is *connectionless* and broadcast orientated. Thus, a UDP stream can be broadcast to many different destinations easily, as there is no handshaking or retransmission.

The FAST protocol is defined on top of the UDP protocol. Every FAST message is guaranteed to be entirely contained within a single UDP packet, with more than one message allowed within the same UDP packet. For the purposes of our work we assumed one FAST message per UDP packet.

Our system leverages existing Ethernet and UDP/IP cores from Xilinx to implement the network stack. We built a simple custom network switch on top of this for additional system monitoring and priority based multiplexing of the channel.

The network switch fulfills two functions. The first of these, is to tag incoming packets with a timestamp taken from a simple 64-bit binary counter such that each increment of the counter is equivalent to a clock cycle. This timestamp is simply forwarded through the other stages of the system and is later received and compared against the counter value at the outgoing port on the network switch to measure the round-trip latency. These latency values are appended to the outgoing UDP packets to allow benchmarking the system during testing.

The switch also implements a *priority-based switching scheme*, allowing multiple connections to share the network stack. If the high priority trading hardware stream is non-empty, it is always preferred to the other connections. This enables the use of the low-priority port for any other purpose as for example sending monitoring information over the network without slowing down the outgoing trades by more than one clock cycle. We do not use this feature in our prototype.

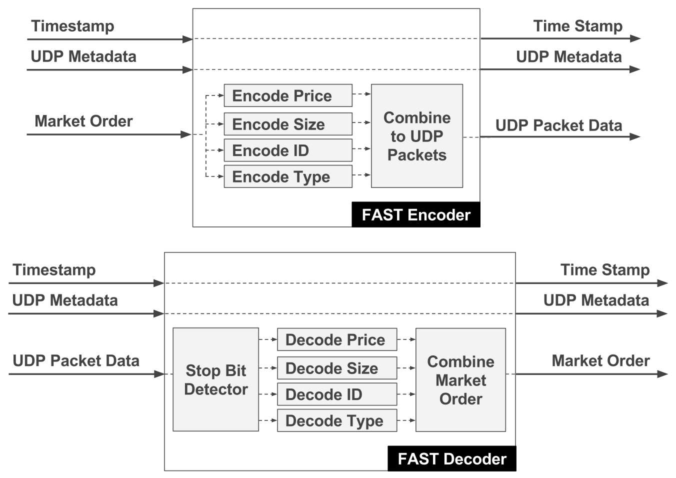

Fig. 4. Block Diagram of FAST Decoder/Encoder

B. FAST Decoder/Encoder

The FAST Protocol is a data compression algorithm developed for low-latency data transfer of market data from stock exchanges to market traders. The encoded data corresponds to multiple field values that define the market order such as the price, the size of an order, its type, etc. The data is decoded based on a template that defines what the fields in the message are and where they will occur.

There are however some challenges in the way a FAST message is organized that make it difficult to decode. For example, the FAST message does not have a fixed size. Thus, depending on the number and types of fields present in the message, the message can span any number of bytes. Also, the fields of a message do not have a fixed size either and can span from one to many bytes.

To detect the start and end of a field in a FAST message the most significant bit of each byte in the incoming network packet is reserved as a stop bit. If it is high, then that is the last byte in this field, otherwise it means that the byte is still part of the same field. Our implemented FAST decoder, as seen in Fig. 4, works by taking in a variable number of 64-bit chunks to which represent a single fast message. Then, it determines where the offset of each field present in the message is. Finally, individual decoders are used to decode all the fields in parallel providing a scalable architecture similar to that in [4]. Finally, the FAST Decoder would output a market order with the fields noted in Table I to be sent to the order book keeping block.

As for the encoder, shown in Fig. 4, it takes an incoming market order from the application layer and encodes all of its variables into the proper FAST-compliant data-types in the minimum number of bytes possible and place the stop bits at the end of each byte for each encoded variable. It then combines all of the encoded fields and send them to the Network Layer as a UDP packet in one or multiple 64-bit data chunks.

C. Order Book

The order book we implemented is based on a heap-like structure with an efficient algorithm for insertion and deletion of nodes presented in [8]. A min (max) heap is a binary tree where each node is guaranteed to have a value less (more) than its two children. In our case, each node of the heap does not represent a numerical value but rather a complete data structure containing the order price, size, ID and type such that the ordering is performed according to the price.

To maintain a complete order book, it is required to have two different heap structures: a min heap where the top of the heap is the order with least price and a max heap where the top of the heap is the order with the highest price for the ask and bid books respectively. Unlike a conventional priority queue, an order book requires to add more features than just inserting a node or deleting the top node such as: arbitrary delete, modify top and multiple-node delete.

1) Arbitrary Delete: The ability to delete any arbitrary node from the heap is necessary in case an order is timed out or is withdrawn from the market. A simple way to implement this is to go through all the nodes in the heap until finding the required node and removing it or marking it as invalid. This is performed in $O(N)$ where N is the size of the heap which is practically not affordable in such low-latency applications as HFT. Also, having a hash map that links the order ID to its current position in the heap would be inefficient in terms of on-chip memory utilization. Therefore, we decided to implement this feature by storing all the incoming arbitrary delete orders in another heap structure that basically keeps track of the to-be-deleted node with highest priority and each time the top of the original heap is changed, if it matches the top to-be-deleted order, they are both removed.

2) Modify Top: If a market order of size S is received, it means that the best S units are to be removed from the order book. However if S is less than the size of the current best bid/ask S_{best}, this requires not to remove the best bid/ask order but only to modify its size to $S_{best} - S$ which is implemented by adding extra logic to check the size of the incoming market order compared to that of the top order.

3) Multiple-Node Delete: This feature is necessary if the size of the received market order S is more than the size of the current best bid/ask S_{best}. This means that the incoming market order will lead to the removal of more than one order from the order book. This is performed iteratively by subtracting the size of the deleted best order from the required size of the market order until it reaches zero.

The order book is implemented and tested to support those features as well as any combinations of them that can result from a complex scenario of received orders. As shown in Fig. 5, the order book takes an input stream of incoming decoded orders along with their time stamps. It passes the time stamp to the output stream and outputs the current top bid and ask orders as streams to the custom application layer. It also has two AXI-Lite ports for the MicroBlaze to read the current bid and ask order IDs for monitoring and debugging the system.

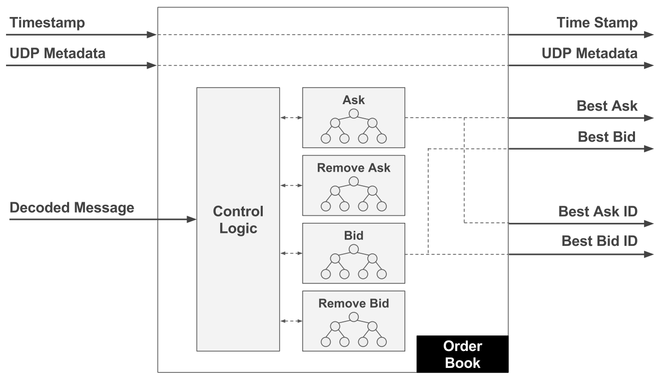

Fig. 5. Block Diagram of Order Book

V. Methodology

We developed all the blocks of our FPGA-based low-latency HFT system using High-level synthesis (HLS). HLS tools allow a hardware designer to develop IP cores at a higher abstraction level than using Hardware Description Languages like Verilog or VHDL which are at the Register Transfer Level (RTL) abstraction level.

A. Design Environment

The HLS tool we used was Vivado HLS which allows us to develop our system blocks using SystemC, C, or C++. The tool synthesizes these high level languages into RTL. Compared to writing RTL, writing and verifying an algorithm using a high level language is much simpler; thus, shortens the the hardware design cycle significantly. With that being said, in order to develop low-latency and minimal-area designs some code changes as well as significant directives must be correctly placed in code. This means that the designer must be aware of the hardware they wish to synthesis even when writing code in a high-level language. The IP cores generated were then integrated together using Vivado IP Integrator which allows us to visually connect wires between modules as well as to external inputs and outputs of the FPGA.

We also used Bitbucket which is a git-based version control to maintain an on-line version of our git repository for sharing amongst the team members. The repository on Bitbucket contained three main folders of source code: network layer source code, financial protocol source code, and application layer source code. Each of the folders corresponds to one or more IP core to be generated by Vivado HLS; moreover each team member was responsible only for designing one of the three sections and had complete control over the source code. This avoided any file accessing problems. At the same time allowed for every member to view other members' source code ensuring that the interface ports are properly matched.

During the integration phase, we copied our repository to Savi server space. Two members of the team would usually integrate at one time using the same PC to avoid conflict with multiple users accessing files. Using the server space we where able to access the FPGA board we used for development.

The board we used for development was the Xilinx Kintex Ultrascale FPGA, more specifically the xcku115-flva1517-2-e. The board contains 663,360 look-up tables (LUT), 1,326,720 flip-flops (FF), 5,520 Digital signal processing units (DSPs), and 4,320 block-ram (BRAMs).

B. Partitioning

We split the project into three parts: the network layer, the financial protocol decoder/encoder, and the application layer. Each team member was responsible for creating the IP cores for one of these parts. The project was divided into these three parts because each one one these parts could be implemented and tested separately then integrated together at a later stage.

C. Simulation, Verification, Testing

The individual testing process for the different system blocks was performed as follows:

Network Layer: The UDP module was linked together with a simple Loop-back module. The purpose of the simple loop back module was to send back the same network packet coming in to the FPGA. This was tested on the board with real network packets being send to and read from the FPGA.

FAST Protocol Layer: The FAST decoder and encoder were linked together back to back such that every decoded message was encoded again. A test module was implemented to send a fixed number of FAST messages to the decoder. After the message is passed through the decoder and looped back into the encoder the final encoded message was then compared with the original message for verification.

Application Layer: A test module was created for sending a stream of decoded market data into the order book. This same test module would take as an input the produced Top Bid and the Top Ask from the order book and verify the results.

After each of the layers were tested separately we incrementally integrated them together. We started by the integration of the Network Layer and the FAST protocol Layer. We preformed the integration by replacing the Simple Loop-back module used for testing with the FAST IP core. The flow of the system would start with the UDP module writing the encoded packets to the FAST decoder. Then the decoder would directly send the packet to the FAST encoder. The re-encoded packets are then sent back out through the UDP module and compared with the original packet on the host PC.

Finally to integrate the Application Layer the wires linking the FAST encoder and FAST decoder back-to-back were instead used to connect to the Order Book module and the Application layer. Correct inputs and outputs between the FAST Layer and the Application Layer were tested in simulation before placing in the final design with the Network layer in hardware. Finally a full system test with 1000 random orders was done to fully test the system functionality comparing the software implantation on the host to the system state printed by the MicroBlaze on the FPGA.

VI. Contributions

We wanted to assure that all three team members acquire the experience of both development of IP cores using Vivado HLS as well as system integration using Vivado IP Integrator. Therefore, each member was responsible for developing the IP cores of one of the system layers: Network Layer, Financial Decoding and Application Layer. Then each two members worked together on the integration tasks: integrating the network layer with the financial decoding, integrating the two application layer cores to the rest of the system and integrating the MicroBlaze for debugging and monitoring to the streaming system. The list of each member's contributions is as follows:

Brett: My contributions were in two main areas; the *server-side software framework* and the *hardware network infrastructure*.

Vivado HLS:

- Developed a hardware dual port UDP switch w/ priority based arbitration between ports. The network switch timestamped all incoming traffic, so that outgoing packets could be tagged with a latency number for benchmarking. Final estimated latency of this block is 1 cycle.

Vivado IP Integrator:

- Initial integration and test of Xilinx UDP/IP hardware stack IP, including developing simple loopback modules.
- Setup and debug Ethernet interface issues between the FPGA and the server.
- Integration of the FAST decoder/encoder with the Network layer and testing the robustness of both blocks with Mustafa.

Software Framework:

- Wrote all server-side software including Python implementations of the order book model and FAST encoder/decoder, as well as network code and a graphical order-book viewer.

Mustafa: My main contributions to the project were in creating the *Financial Protocol Layer*.

Vivado HLS:

- Designed and developed an HLS implantation of the FAST Decoder and FAST Encoder. Verified the design in IP integrator simulation.
- Optimized the FAST Layer to run with the as low latency as possible with a 5-ns clock. Final results were 9 cycles for the FAST Decoder and 0 cycles for the FAST Encoder.
- Modified the fast Decoder to send a request to the proper network port as well as await its response to initialize the Network interface. After this the decoder can then start accepting network packets and will never go back to the initialization state again.

Vivado IP Integrator:

- Integrated the FAST Layer with the Network Layer. Additionally helped ensure that the software and hardware implementation matched and followed correct FAST protocol specifications with Brett.
- Integrated the FAST Layer with the Application Layer with Andrew. First in simulation, then in hardware.

Andrew: My contributions to the project were in the *Application Layer* and the *debugging MicroBlaze*.

Vivado HLS:

- Designed the order book IP core based on a highly optimized heap-like structure. This core supports all the features required for the HFT order book keeping application that can be stripped down to be used as a hardware priority queue for any other streaming application. Final estimated latency is 53 cycles for basic insert and delete operations including complete re-ordering of the heap structure but the top entry is written to the output stream once ready.
- Implemented a very basic application layer IP core to close the cycle and be able to measure the round-trip latency of the system. This block sends a decision once the best buy (sell) price becomes less (more) than a specified threshold. Final estimated latency of this block is 1 cycle.
- Implemented several adapter IP cores that were used in testing and trial designs such as: BRAM to AXI-Stream, AXI-Stream to BRAM and AXI-Lite to AXI-Stream adapters.

Vivado IP Integrator:

- Integrated, simulated and tested the order book and simple threshold blocks together.
- Integrated the FAST decoding/encoding layer with the two blocks of the application layer with Mustafa.
- Incorporated the MicroBlaze soft-processor to the final system design to read and print to the console the top bid and ask order IDs to compare to the software visualization on the host.
- Modified the final system design to use different clocks since the Ethernet IP core and the Network layer runs at 156 MHz (64-bits at 10Gb/s) and the rest of the system runs at 200 MHz.

Resource	Utilization	Available	Utilization %
LUT	49,638	663,360	7.48
LUTRAM	2,148	29,3760	0.73
FF	32,718	1,326,720	2.47
BRAM	474	2,160	21.92
DSP	3	5,520	0.05

TABLE II. FPGA RESOURCE AREA UTILIZATION

Frequency	Round-trip Latency	Round-trip Runtime
156 MHz	88 cycles	563 ns

TABLE III. FPGA TIMING RESULTS

Module	Latency	Total Latency	Runtime
Network Transmit	13 cycles	-	85 ns
Network Receive	33 cycles	-	209 ns
Network Switch	12 cycles	12 cycles	76.9 ns
FAST Encode/ Decode	18 cycles	30 cycles	115.4 ns
Order Book	12 cycles	42 cycles	76.9 ns

TABLE IV. TIMING BREAKDOWN

System	FPGA	Decode/Encode	Latency
This work	Kintex Ultrascale	192.3 ns	563 ns
Ref. [9]	Virtex-4	-	2.6 us
Ref. [3]	Virtex-5	200 ns	1 us

TABLE V. COMPARISON TO OTHER WORKS

VII. DESIGN CHARACTERISTICS

A. *Resource Utilization*

Our final placement resource utilization on the Xilinx Kintex Ultrascale FPGA are reported in Table II. Since Kintex FPGA is large two-die FPGA, the overall resource utilization percentage of our design is minimal. However, our BRAM utilization does take a noticeable percentage of the overall design. This utilization is mostly due to the order book storage. The oder book is able to support 4096 market order entries so the BRAM utilization noted here is sufficient for storing a significant amount of market data without the need to scale. Additionally, almost half of the BRAM utilization came at the trade off for running the order book at lower latency. This was due to limitations in HLS array partitioning noted in the conclusion section, which made it difficult to lower the BRAM utilization without sacrificing latency. Therefore, we took advantage of the space available on the FPGA and traded off area for lower latency.

We ran our final design on a 156.2 MHz clock. The clock frequency for our design was determined by the network stack, which in turn uses 156.2 MHz to match the 10G Ethernet bit rate. Future work may include trying to operate the non-network related logic at a high clock frequency. We measured the round trip latency by sending 10,000 random test orders and averaging the number of cycles it takes for the packet to enter the Network Layer until it is sent back to the host through the Network Layer when triggering a response from the Application Layer. On average, we achieved a round trip latency of **42 cycles**. Adding the additional latency need for transmitting and receiving through the Ethernet as reported by [10] gives us a total or **88 cycles**. Therefore, the total round trip runtime is **563 ns** as summarized in Table III.

To better understand the latency due to the addition of different modules in our design we report the timing breakdown in Table IV. We note that the physical Ethernet port takes 85ns on transmit and 209ns on receive [10], we suspect that this latency is due to the streaming interface, a latency insensitive handshake protocol. This shows us that latency of the streaming interconnect is indeed significant. When adding the FAST Encoder/ Decoder we achieved a total latency 30 cycles for the system. Up to this point we are still able to run the system at very low latency. Interestingly enough adding the Order Book to the system only added 12 cycles of latency. This is minimal even though the Order Book is a significant contribution to the system.

In Table V, we compare the system round-trip latency achieved in this work to other FPGA-based High Frequency Trading systems introduced in literature. In [9], they implement multiple parallel FAST decoding streams that decodes incoming packets and passes the decoded data to a processor that performs the book keeping and any financial application in software. They report a total latency of 2.6 us measured from a packet transmit until receiving decoded data on the processor. In [3], the authors use a different financial exchange protocol called FIX. They report a latency of 200 ns for encoding and decoding the packets and a round-trip latency of 1 us excluding the application layer. When performing the same measurement on our system, we record a comparable latency of 192 ns for the encoder/decoder and a round-trip latency of around 563 ns including order book keeping.

B. *Where The Time Went*

Initially, we spent most of our time optimizing the IP cores to operate with as low-latency as possible. This was done though code modifications and applying HLS pragmas. However, before all optimizations were complete we spent some time to integrate our blocks using Vivado IP Integrator to see if we can get a functional system running on the FPGA. This ended up taking up a lot of our time as simulation results did not initially match up with the results seen coming from the FPGA. It was good idea that we attempted the integration early though since after this each of us was able to individually optimize their code and test its effect on the final system. Therefore, after the initial integration we had no major bottlenecks since no team member was waiting on another to be able to test their system.

VIII. PROBLEMS

We encountered issues with IP Integrator whilst integrating the FAST encoder/decoder into the design. We had functional operation during simulation, but ultimately unresponsive hardware when synthesized. We ended up simply modifying the loopback module incrementally with the encoder/decoder features. Despite much investigative work, we are still uncertain as to the cause of the issue, although we suspect it was related to the interface pragmas used.

We used 16-bit fixed-point representation for the prices in the order book. This gave us enough resolution to differentiate between two prices that are different by 1 cent for our application. However, it might cause precision loss when converting the decimal floating-point representation of the FAST protocol to the fixed-point representation in the order book but still be able to differentiate between two prices that are 1 cent apart. If higher resolution is required then wider fixed-point format could be used trading resources utilization for higher resolution.

When trying to add a MicroBlaze to the final system for debugging purposes to print the current top bid and top ask IDs to the console, we encountered a problem that the use of `xil_printf` function inside a loop did not build or generate the ELF file and complained that it can not fit into the local MicroBlaze instruction memory. However, it compiled perfectly fine when placing the `xil_printf` outside the loop. We did not find an obvious reason for that and the problem was resolved by increasing the size of the MicroBlaze's instruction memory during its block automation in Vivado IP integrator.

IX. Conclusion and Comments

Most of the low-latency systems tend to use RTL designs as it gives the designer a finer cycle-by-cycle control over the generated hardware. This project shows that, even for very tight latency constraints, HLS tools could be used to achieve results that are comparable to prior RTL-based work in a much less development time. Our project would likely have taken twice as long or more (four months instead of two) if we had opted for a purely HDL approach instead. It will require much more time and effort to write the modules in HDL for the FAST encoder/decoder and order book which are more software-oriented. For example, adding support for an additional exchange FAST message template is much easier to do by modifying C++ code rather than HDL. Generally, HLS technology seems mature enough for real projects, especially if the work is streaming based which is completely handled and supported by HLS libraries.

Another strong point for HLS is that the experts developing the financial trading algorithms most probably have no or very little background in RTL-based hardware design which makes it really hard for them to accelerate their algorithms on an FPGA. Therefore, developing the infrastructure in HLS makes it easier for them to exploit the power of HLS by developing their algorithms in software using C/C++, optimizing it for as small latency as possible and easily integrating it to our infrastructure.

On the other hand, one of the problems we encountered with HLS in the design of the order book block is being unable to do custom partitions of an array. Having a tree-like structure, it would have been useful to partition the order book array into variable-sized parts since the tree levels are of sizes 1, 2, 4, 8 and so on. Therefore, the only two options were to use separate arrays of different sizes for each level or to use a 2-D array of size equal to the number of levels multiplied by the size of the base level. We went for the second option although it consumes more resources since the first option would fairly complicate the code that uses index arithmetic linking between two levels.

If we were to repeat a similar project, we would focus more closely on interface specification upfront, to avoid the many small corrections we had to make to get consistency. Collaborating on a system design must involve everybody agreeing on a clear statement of goals and scope, and the sooner this happens the less reworking needs to be done.

Course Comments:

In terms of the course contents, it would have been interesting to get introduced to how HLS tools work in more details instead of just how to use them. We think that understanding how the tool was developed and how it works will definitely help us understand how it behaves in specific situations. Also, it would have been useful to get some hands-on experience about the clock-domain crossing we studied in the lectures by implementing a multi-clock design in one of the course's assignments. In terms of the project, it would be beneficial if the initial assignments could somehow be tied to developing a small, simplified parts of the overall final project. This might aid in completing projects by the end of the course, as well as reveal potential problems earlier in the course.

References

[1] W. Barker *et al.*, "The growth of high-frequency trading: Implications for financial stability," *Bank of Canada-Financial Systems Review*, 2011.

[2] H. Subramoni *et al.*, "Streaming, low-latency communication in on-line trading systems," in *IEEE International Symposium on Parallel Distributed Processing (IPDPSW)*, pp. 1–8, 2010.

[3] J. W. Lockwood *et al.*, "A low-latency library in fpga hardware for high-frequency trading (hft)," in *IEEE Symposium on High-Performance Interconnects*, pp. 9–16, 2012.

[4] H. Li *et al.*, "Fast protocol decoding in parallel with fpga hardware," in *IEEE 17th International Conference on Computational Science and Engineering (CSE)*, pp. 1669–1673, 2014.

[5] D. Rozhko *et al.*, "Packet matching on fpgas using hmc memory: Towards one million rules," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 201–206, 2017.

[6] S.-W. Moon *et al.*, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1215–1227, 2000.

[7] D. Rosenberg, "Fast specification version 1.1," Dec 2006.

[8] R. V. Nageshwara and V. Kumar, "Concurrent access of priority queues," *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1657–1665, 1988.

[9] C. Leber *et al.*, "High frequency trading acceleration using fpgas," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pp. 317–322, 2011.

[10] Xilinx, "10 gigabit ethernet subsystem v3.0 product guide," pp. 14–15, November 2015.