Building the Book: A Full-Hardware Nasdaq Itch Ticker Plant on Solarflare's AoE FPGA Board

Miles Sherman (ms4543@columbia.edu), Pranav Sood (ps2729@columbia.edu),
Kevin Wong (kw2500@columbia.edu), Art Iakovlev (ai2283@columbia.edu),
& Naman Parashar (np2437@columbia.edu)

May 31, 2013

1 Abstract

The Itch ticker plant receives market data from the Nasdaq stock exchange and builds a database of all open orders for given stock symbols. The system is implemented in hardware on Solarflare's AoE FPGA board and utilizes the ternary tree data structure to enable $O(logn)$ lookup times in hardware. In addition to its performance, this system is designed in a parameterized manner so as to allow for future optimization and customization.

2 Overview

The stock market has undergone a technological overhaul in the past few decades. Cutting edge software technologies have implanted themselves in this industry by enabling high speed trading and providing large firms with the statistical information and risk management solutions they need to be successful. However, as technology scales and latency requirements continues to become more stringent over time, the shortcomings of software have been exposed. The amount of data streaming through the Nasdaq at any given moment is enormous and running software on a general purpose CPU simply can no longer satisfy the demand for low latency. There has been extensive effort to optimize these softwares to handle the data but there is an intrinsic ceiling on performance as determined by the general purpose chips supporting their operation. As this ceiling approaches quickly, the industry has turned to custom hardware to help drive the market and satisfy the current demands for low latency.

There are a number of different standard communication protocols used to transmit market data. Most of these protocols provide extensive information regarding the current state of the book (the database of open orders). Some exchanges only provide a fixed number of price levels; however, Nasdaq, the second largest exchange by market cap, provides only the bare minimum amount of information regarding each market event. This is done in an effort to decrease the latency of data transmission and it is receiving party's responsibility to handle the data in an efficient manner. This creates direct competition between firms to build the fastest receiving system; a single micro-second can determine the difference between a huge win or bankruptcy.

From the sudden demand for custom hardware in the marketplace has come a number of financial hardware design startups as well as an onset of internal hardware development at the larger trading shops and banks. Initial designs from these companies have been somewhat simple, exploiting the intrinsic speed of custom hardware but without the level of complexity that is much more cumbersome to implement in hardware than software. With this project, we took a first step towards the hardware implementation of a large-scale market-data handler with the complexity of a typical software system. For this system, we chose to handle the Nasdaq Itch protocol specifically because of the limited amount of information transmitted in each message (see figure 5 for relevant message types and their associated data fields). While the messages corresponding to initial additions of orders to the database (this database is referred to as the L3-Book) contains a complete description of that order (including order number, stock symbol, price, quantity of lots, etc), following messages that affect that same order will not reference the stock symbol or other order information. Therefore, in order to perform operations on that order, it is necessary to lookup the order from the L3-book (using the 64-bit order number as a key) which can contain

millions of orders at times. For this reason, a strictly iterative search would be an unacceptable bottleneck in the system. To solve this problem, we implemented the entire L3 book as a ternary tree data structure in hardware on a state of the art FPGA (Altera's Stratix V on Solarflare's AoE board). Using the below mentioned algorithms, we have achieved very impressive values of tick-to-trade latency, values that are not feasible in software. Also, in an effort to compete with the re-programmable nature of software systems, we have parameterized our system so that the algorithm can be easily changed and re-implemented using the resources we have created. With the support of Solarflare, Altera, Nasdaq, and some others, we are bringing our design up to industry standard quality and plan to open source the finished product by the end of 2013.

Below is discussed the general design of the hardware system from the ground up. In addition, current achievements are mentioned and goals for the near future are set.

3 Structure of the Ticker Plant System

The individual modules making up the Itch ticker plant system are discussed below. Much time was spent developing the algorithms by which the system would be implemented. To understand the functionality of our ticker plant, it is important to first understand our underlying methods. The top level block diagram of the system can be seen in figure 1.

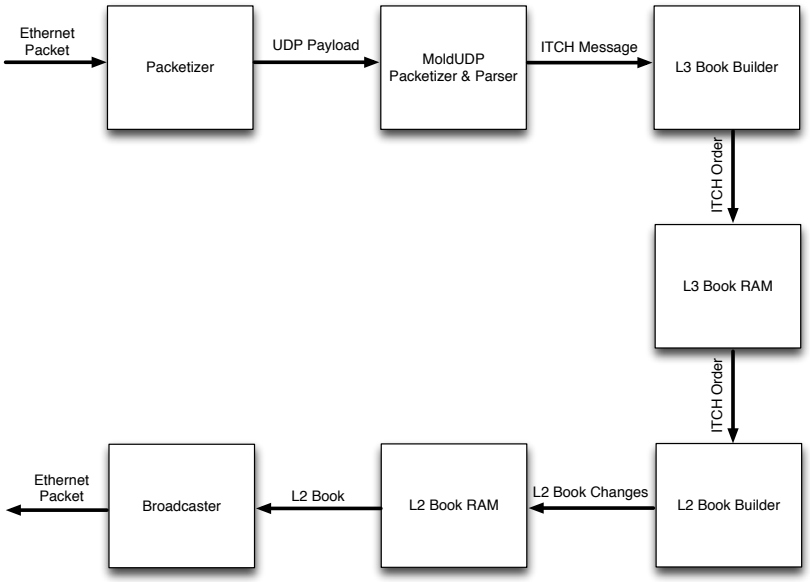

Figure 1: Top Level Block Diagram of the Nasdaq Itch Ticker Plant

3.1 UDP Packetizer

The packetizer is the first module in our design. The payload data inside each packet is encapsulated inside Ethernet, IP and UDP headers. The packetizer essentially strips off these header fields and passes on the payload data to the next stage, which is the MoldUDP Packetizer. This has been implemented using a state machine which begins when the EN and data_valid_in signals (all signals reference figure 2) are high and it receives a start_of_packet signal. It continues sequentially from one state to the other, removing unwanted header data in each stage. When it reaches the payload of the packet, it sends out a data_valid_high signal to the the next

module (MoldUDP), which is an indication that this is the data at the packetizer's output is a valid part of the payload and can be processed. Also, the packetizer sends out the length of the payload for each packet to the MoldUDP module. As soon as the payload is exhausted, the done_o and the end_of_packet signals go high, and the FSM returns to the start state where it waits for start_of_packet signal to go high again. In case an anomaly is detected in the middle of a packet, i.e. it is not IPv4, the packet is dropped and the FSM returns to start.

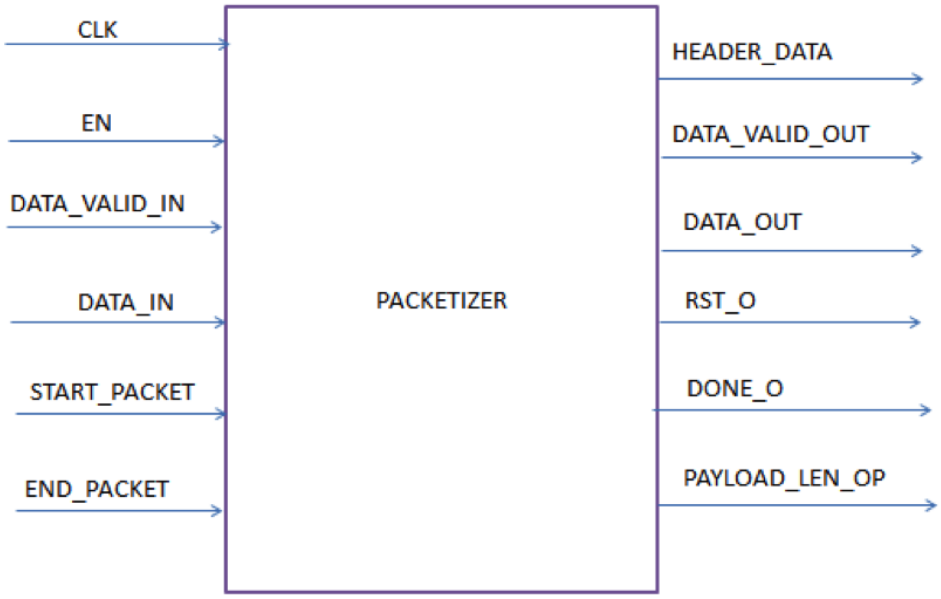

Figure 2: Top Level Interfaces of the Packetizer

Data comes into the packetizer at the rate of 8 bytes/clock cycle. If enable and valid_in signal are both high, and the start_of_packet signal goes high, the FSM begins to operate. The various states in the FSM are listed below and the state diagram can be seen in figure 3.

- start
- eth_src_mac
- eth_vlan (only if VLAN tagged, otherwise this state is bypassed)
- ip_hdr_s1
- ip_hdr_s2
- ip_hdr_opt
- udp_hdr_opt
- payload (stays in this state until the payload is exhausted)
- drop_packet (any anomaly in the packet causes the FSM to go here, it returns to start in the next cycle)
- others (this state handles any unforeseen situation, the FSM returns to start on the next cycle from this state as well.

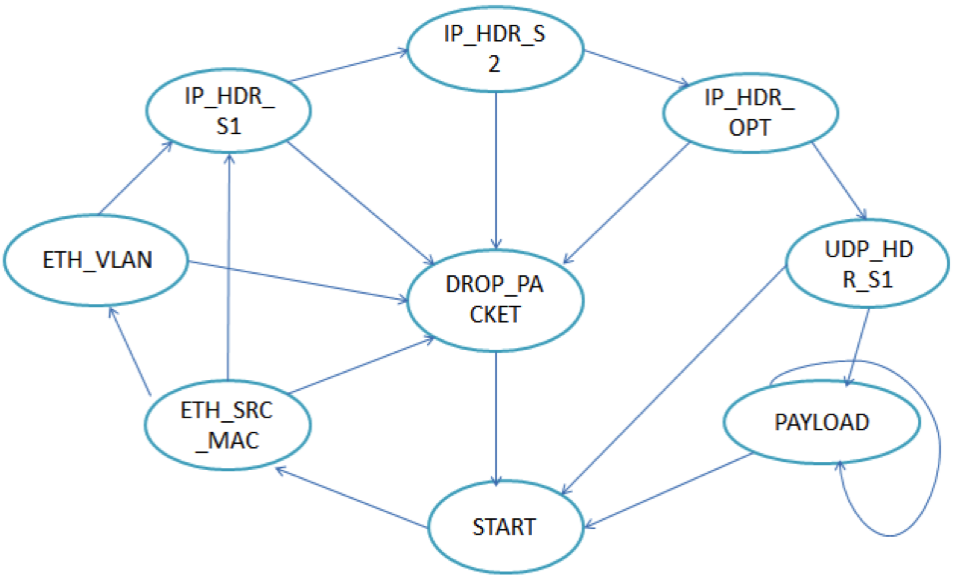

Figure 3: Finite State Machine Diagram of the Packetizer

As the input stage of the entire system, the final duty of the packetizer is to interface with the Avalon Streaming Bus of the Stratix V FPGA. This is the bus by which packets are received into the system. A very specific protocol must be obeyed for successful data transfer, the timing diagram for this protocol can be seen in figure 4.

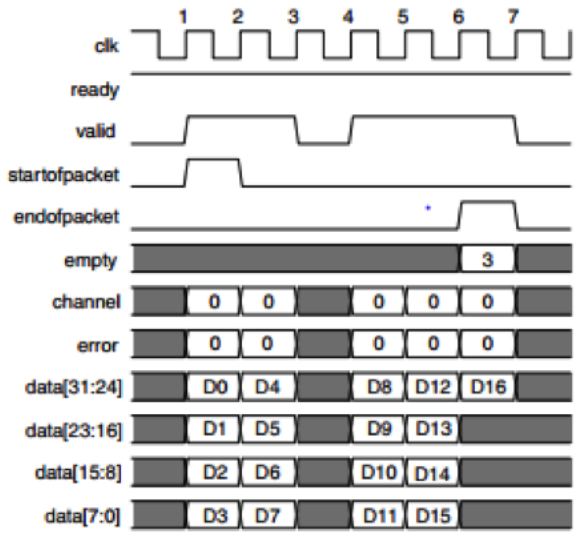

Figure 4: Timing Diagram for the Avalon ST Protocol

3.2 MoldUDP Packetizer & Itch Parser

The purpose of the MoldUDP packetizer is to remove MoldUDP header components, as well as to parse, filter, and output itch messages that are relevant from incoming data. To accomplish this task, two separate sub-modules, a data register state machine, and a message parser, were necessary. The MoldUDP module receives as inputs

64-bit wide data from the IP Packetizer as well as a data valid signal which is used to indicate the validity of the incoming data. If the valid signal is high, the input port is read and stored into the buffer. If the valid signal is low, the input port is ignored. The MoldUDP module sends as output parsed Itch messages to the FIFO towards the L3 book builder. The outputted message contains only the components required by the L3 and L2 books. While the parsing of messages is trivial, indexing the variable quantity and lengths of Itch messages within a single packet is not and our method is outlined below.

3.3 MoldUDP Implementation

The MoldUDP data register sub-module stores incoming data into a buffer and outputs messages to the message parser. The state machine initializes with default fields that are relevant to the header of a MoldUDP packet message length was set to the length of the MoldUDP header, and the isheader signal is set high.

As data is received it is stored into the data buffer, and a counter keeps count of the amount of bits stored named index. The first two bytes of each message, loaded into a message length register called meslen, contains information about the message length, which is used to determine the beginning and end of a message loaded into the data buffer. When index exceeds meslen, a message is sent from the data buffer to the message parser sub-module, while data inside the data buffer indexed between meslen and index are shifted to the front of the data buffer. The values of index and several other state variables are updated at the end of every cycle, while the value of meslen is updated every time after a message has been sent to the parser.

The interface between the message parser and data register state machine is that of a simple one-way request handshake. Due to the nature of the implementation of the parser, there are no logical elements which stores incoming data; the parser simply filters unwanted messages and unnecessary components of certain messages, all of which is done through combinational logic. As such, the message parser can always receive data from the data register state machine and is not required to send any acknowledge signal back to the data register state machine. Once the data is filtered, the output valid signal and the filtered data are sent to the MoldUDP message FIFO to be stored and read by the L3 book.

3.4 L3 Book Builder

At the heart of the Itch processing system is the L3 book, a database of all open orders on the Nasdaq waiting to be filled. Keeping this database in hardware and updating it in real time is not a trivial task. However, when implemented correctly, it provides additional functionalities for the user at a minor latency and resource cost. As can be seen in figure 5, only 'add order' messages provide the stock symbol as part of the Itch protocol. For all other messages, it is necessary to perform a lookup from the L3 book to determine the appropriate information to make the message useful. In software, as the L3 book is typically implemented, this lookup is a significant bottleneck and can cost tens to hundreds thousands of clock cycles of latency. In hardware, an iterative search will take a few thousand cycles. While the hardware implementation makes our system's lookup intrinsically faster, the iterative search still becomes a bottleneck in the system, limiting the number of stocks that can be tracked and more importantly the total tick-to-trade latency. To solve this problem, we designed a hardware implementation of a ternary tree, an AVL tree to be exact.

Add Order (no MPID)	Message Type	0	1 "A"
	Timestamp	1	4
	Order Reference Number	5	8
	Buy/Sell Indicator	13	1 "B"/"S"
	Shares	14	4
	Stock	18	8
	Price	24	4
			30

Add Order (w/ MPID)	Message Type	0	1 "F"
	Timestamp	1	4
	Order Reference Number	5	8
	Buy/Sell Indicator	13	1 "B"/"S"
	Shares	14	4
	Stock	18	8
	Price	24	4
	Attribution	28	4
			34

Order Executed	Message Type	0	1 "E"
	Timestamp	1	4
	Order Reference Number	5	8
	Executed Shares	13	4
	Match Number	17	8
			25

Order Executed (w/ Price)	Message Type	0	1 "C"
	Timestamp	1	4
	Order Reference Number	5	8
	Executed Shares	13	4
	Match Number	17	8
	Printable	25	1
	Execution Price	26	4
			30

Order Cancel	Message Type	0	1 "X"
	Timestamp	1	4
	Order Reference Number	5	8
	Cancelled Shares	13	4
			17

Order Delete	Message Type	0	1 "D"
	Timestamp	1	4
	Order Reference Number	5	8
			13

Order Replace	Message Type	0	1 "U"
	Timestamp	1	4
	Original Order Reference Number	5	8
	New Order Reference Number	13	8
	Shares	21	4
	Price	25	4
			29

Figure 5: Relevant Itch Message Types and Their Fields

3.4.1 The Ternary Tree

More than anything, the ternary tree is important in our system because it enables searches of memory in $O(logn)$ time. The tree structure allows the search to be performed as follows. First, the search begins at the root node of the tree. It evaluates the data of that node and compares it against the data being queried. If the query data is greater than the node data, we move to the right child node. If the query data is less than the node data, we move to the left chile node. After each comparison at a node of the tree, the number of remaining addresses to search is halved. Once the node is found with data matching the query data, the address is outputted and the lookup is performed. We required a ternary tree as opposed to a standard binary tree because as the L2 builds its truncated version of the book, it requires access to the L3 book to build new price levels when they are either deleted or fully executed from the L2. This search requires that multiple orders be matched to the query before the search is complete. The center node of the ternary tree is accessed when data is matched to its parent and the search iterates through the center nodes until it reaches the bottom, streaming out addresses on each pass. Thanks to the balanced structure of the tree, lookups become much more efficient and somewhat trivial. What is not quite so simple is the process of building and maintaining the tree structure using only a single shared memory array (to hold the actual nodes of the tree) and logic gates in hardware. An abstract visualization of the

ternary tree system can be seen in figure 6.

We will discuss the implementation in two parts, the insert operation and the delete operation. Then we will describe how the ternary trees were integrated into a system to provide the full functionality required for the L3 book building.

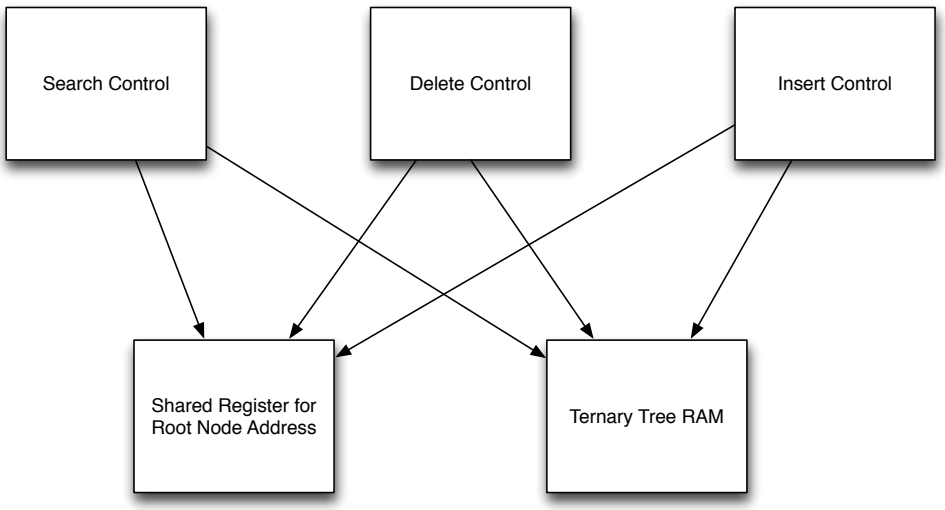

Figure 6: Top Level Block Diagram of the Ternary Tree System

3.4.2 Inserting to the Tree

Adding orders to the L3 book requires that not only the order also be added to the ternary tree but also that the tree be balanced to maintain the $O(logn)$ search latency. If the tree were to become unbalanced i.e. the left side of the tree has significantly more height than the right, the search latency would certainly be adversely affected. To ensure that the tree stays balanced, we chose the AVL flavor of balancing. The algorithm for inserting a new node is as follows. First the insert controller "climbs" down the tree in a similar fashion to the above mentioned search operation. Once the appropriate parent node is reached, the new node is written as that node's left, right, or center child depending on their relative data values.

The way by which the AVL tree is kept balanced is by means of maintaining a special value at each node called a balance factor. This balance factor value can range from -2 to 2 and is determined by subtracting the height of the right child nodes from the left child nodes.

$$Balance\ Factor = Height_{left} - Height_{right} \tag{1}$$

Upon writing the new node to memory, the insert controller proceeds to climb back up the tree, checking balance factors at each node. If a balance factor of -2 or 2 is read, the tree has become unbalanced and action is required to fix the issue. This action is called a rotation. There are four different types of rotations of an AVL tree: Left-Left, Left-Right, Right-Left, and Right-Right. Each corresponds to a specific configuration of the nodes below the unbalanced node. Assuming all previous rotations were performed properly, these are the only four possible cases of imbalance. The method by which the rotations are performed is outline in figure 7. In the figure, each circle is a node of the tree and the triangles below them represent perfectly balanced sub-trees which can be considered as one node. Left-Left and right-right rotations require one rebalance while left-right and right-left rotations require that a left-left or right-right rotation respectively follow them. Therefore, latency of the rebalance doesn't depend on the size of the tree like the search, instead it depends on the particular configuration of the tree's nodes.

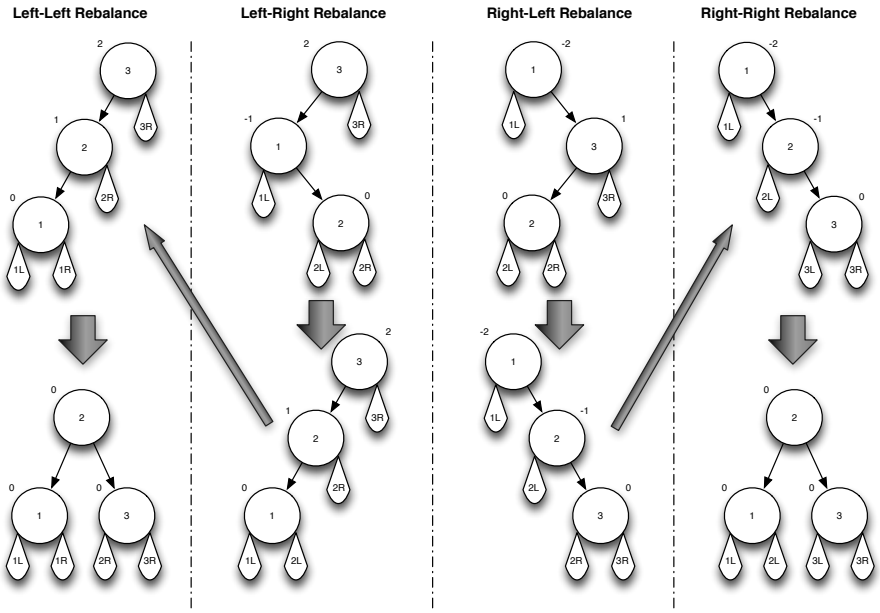

Figure 7: Top Level Block Diagram of the Ternary Tree System

At most one rebalance operation (for left-right or right-left this includes the following left-left or right-right rotation) is necessary per insert so the insert controller returns directly to the root node following the rebalance where it awaits its next command. A state diagram of the insert algorithm can be seen in figure 8.

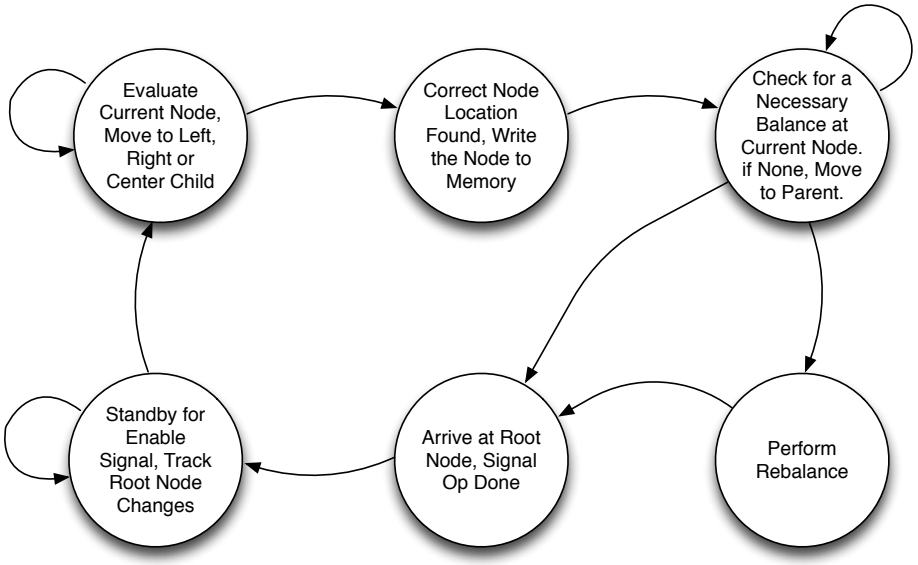

Figure 8: State Diagram of the Insert Operation for the Ternary Tree

3.4.3 Deleting From the Tree

Similar to insert and search, self-balancing Binary Trees were used to delete nodes from the L3 book. The algorithm similar to that of deleting nodes from the AVL tress was followed in general. A few changes were made to the standard algorithm since the each node could possibly have a center node indicating the same stock price level, although these nodes did not lead to rebalancing.

The search module provided the address of the node to be deleted to the delete module. Before the node was deleted, the children of the node were checked in order to fins the suitable replacement for the node. After deleting the node and replacing it with once of its children (if required), all the relevant nodes were updated with the address of the of replaced node. The next step was to check the balance factors. If the parent of the node, which was deleted, had a balance factor of 0, the balance factor of that node was updated and the entire operation of delete ended. If the parent of the deleted node had a balance factor of ± 1 then the balance factor was updated. If the updated balance for any node was ± 2 then a rotation (depending on which sub-tree was heavier) was done around that node. At this all the concerned nodes were updated with the changed addresses. If the updated balance factor was ± 1 or 0 then process of continuing the updating balance factors continued until we reached the root. If at any point in this process the root of the entire tree was changed the information was shared and the new root was updated to the insert module so as to keep track of the root.

The most difficult challenge of implementing a self-balancing search tree in VHDL was to update the balance factor in an efficient manner after each delete. Since we were not keeping a track of the current height at each node, a separating algorithm of updating the balance factors was implemented which added some over head to the each delete operation.

The delete was validated with random data with an initialized memory of 1000 nodes and it was made sure that each operation delete operation was completed.

After integration with the rest of the L3, a bug was found in the corner case of delete, which came up after a lot of nodes were deleted. This has not been successfully debugged yet.

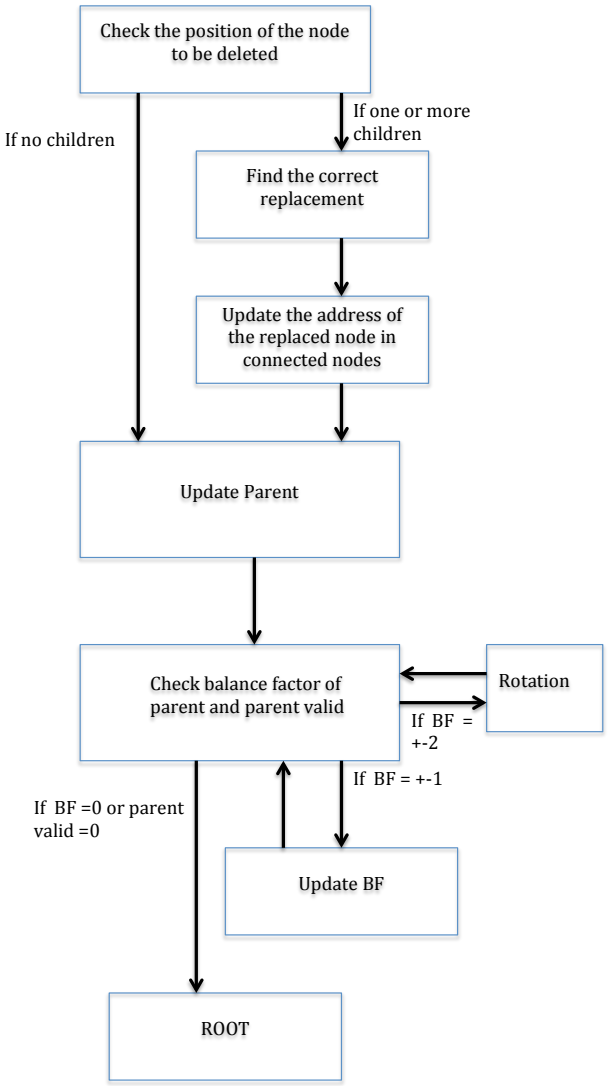

Figure 9: Block Diagram of the Ternary Tree Delete Operation

3.4.4 High Level Control of the L3 Book Builder

In order to support the ternary tree, it was necessary to design a module that dynamically allocates memory locations for the insert and de-allocates memory locations that have been deleted. To accomplish this task, we implemented a simple module that keeps track of the range of open memory locations and has the ability to wrap around from large addresses to small addresses when the top of the stack reaches the last address. It acts much like an over glorified barrel shifter. Beyond this memory allocation module, the top level of the L3 book builder

consists of a significant RAM array which holds the actual contents of the L3 book as well as a ternary tree which indexes based on order number and another ternary tree which indexes based on the concatenation of stock symbol and price. The nodes of the latter still correspond to individual order numbers; however, implementing the data field of this tree as stock symbol and price allows the L2 book to request a search to build a brand new price level for a particular stock. That algorithm is to be discussed below. A high level block diagram of the L3 book builder can be seen in figure 10.

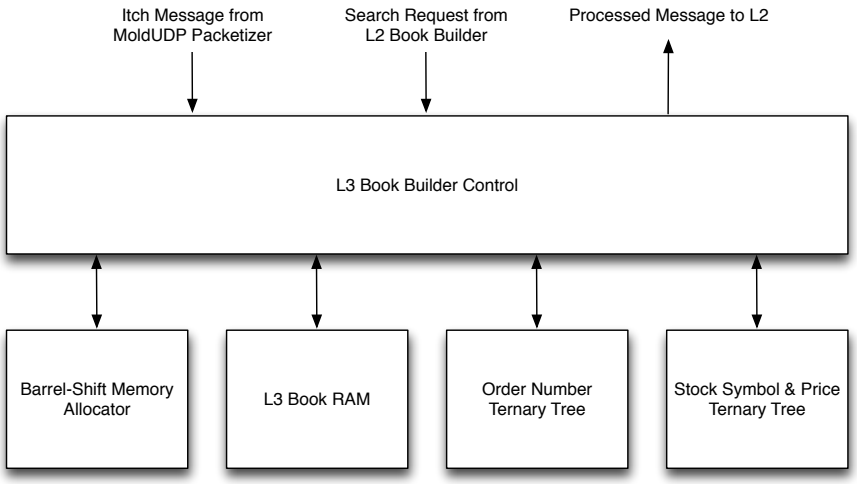

Figure 10: Block Diagram of the L3 Book Builder

Because of the generic and parameterizable nature of our ternary tree, we were able to instantiate it easily and recurse upon it. This helped the implementation of the top level module to progress smoothly. The top level of the L3 book builder not only acts to hold the database of open orders but it also acts as an interface to the L2 book builder. The main output of the L3 is to a FIFO to the L2 to where it sends messages using an internal protocol. This message passing enables the L2 to builds its book with minimal access to the L3 database. It is only necessary for the L2 to search the L3 book when a price level is eliminated completely from the L2 and a new level must be built. A detailed state diagram of the L3 book builder is shown in figure 11.

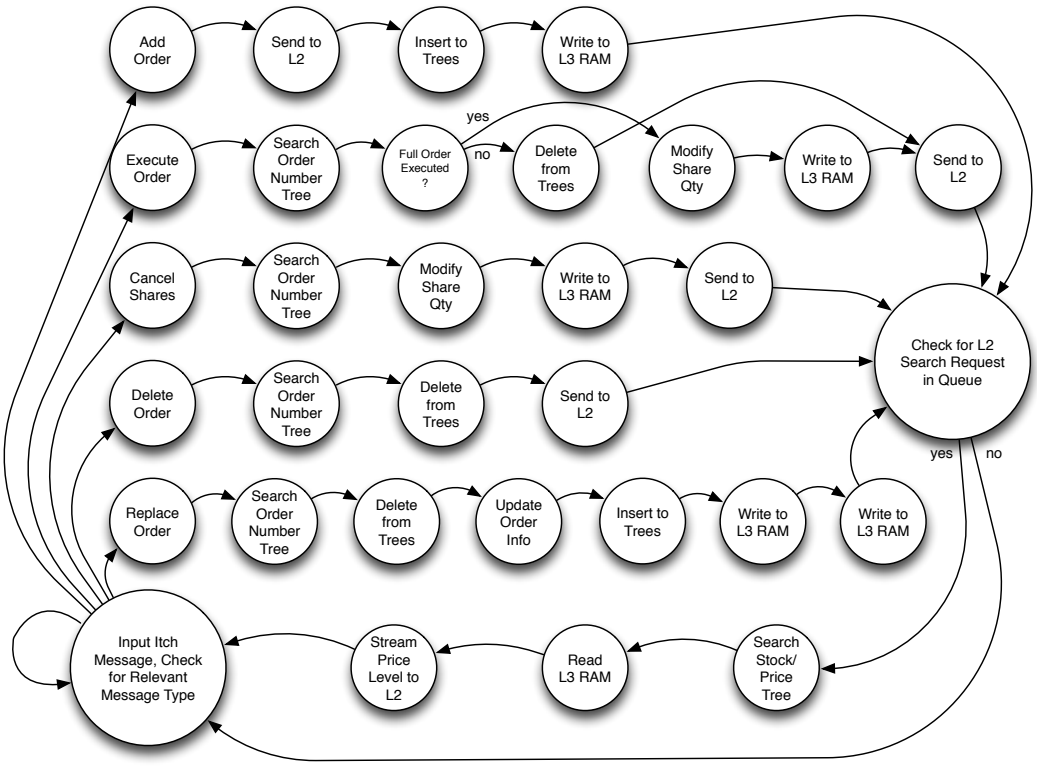

Figure 11: State Diagram of the L3 Book Builder

3.5 L2 Book Builder

The L2 book builder receives the internal messages sent by the L3 and builds its own summarized user-readable version of the book. Instead of storing all the open orders in a database like the L3, the L2 tracks only a subset of those orders corresponding to the ten best "bid" or best "ask" price levels of the relevant stock. Also, unlike the L3, the L2 does not keep track of the individual order numbers but instead it logs statistics of the open orders for each price level that is within ten levels of the best bid or ask. These statistics include stock symbol, price, number of orders at the price level, and number of total shares at the price level. The internal protocol between the L3 and L2 provides the L2 with all of the information it needs to build the L2 book without having to access the L3 for reference information except for one case. This case comes after the L2 becomes full and later an entire price level of the L2 becomes either fully executed or deleted. When this happens, the L2 sends a request for search to the L3. The requested data to search for is the stock symbol in question concatenated with the next most relevant price level to the level that was deleted. The L3 book searches the stock symbol and price tree starting on its next free cycle and streams the quantity of shares on each order back to the L2 for building the new level. If no orders are found in the stock symbol and price tree for the requested price, the L2 increments the price by one penny and requests another search. In the meantime, any messages the L3 sends to the L2 will be queued in a FIFO.

Beyond the ten levels of bid and ask, the L2 keeps track of the last executed price and number of shares for each relevant stock. After each change to the book, the L2 signals to the broadcaster to read the changed data from the L2 RAM and output as a UDP packet.

The L2 consists of several control modules and an order parser that are implemented as finite state machines with one-way and two-way handshaking (see the block diagram for this module in figure 12). In addition, six

RAM modules are used to build both ask and bid books for the three tracked stock symbols. Each RAM (word) is 12 bytes wide and 10 levels deep. The reason behind implementing six separate RAM modules is to minimize utilization of memory by subtracting the 8 bytes required to store the stock symbol from the word. The stock symbol is hard-coded to its appropriate RAM, which decreases logic utilization as well.

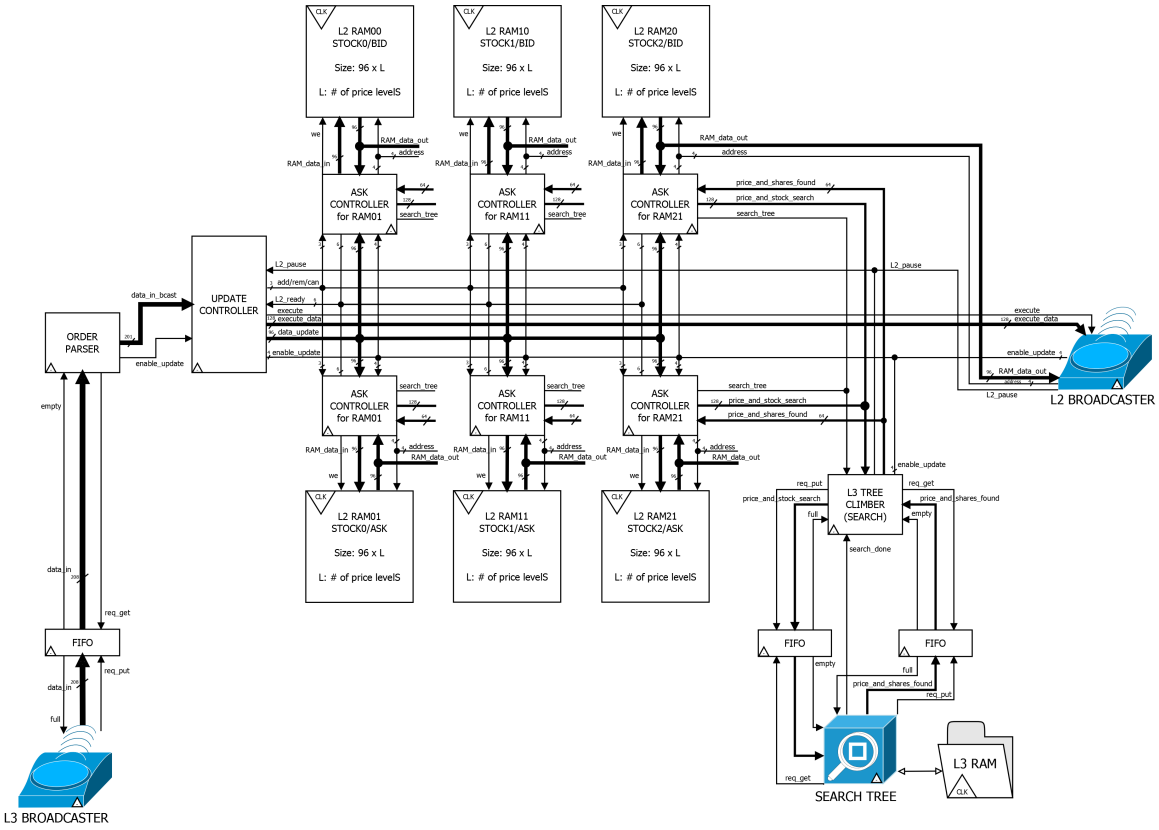

Figure 12: Block Diagram of the L2 Book Builder

As the data arrives in the FIFO from the L3 broadcaster, the order parser sub-module accesses that data and parses it into the L2-recognizable data format. Once the data is parsed, it is sent out to the update controller sub-module via a one-way handshake. Based on the received message type, the update controller decides where to send the update by hashing/encoding the stock symbol received in the message into a 4-bit wide signal used to enable a mapped ask/bid controller sub-module. Moreover, the update controller not only decides where to send the data, but also how to send it. When the message type is either "execute" or "execute w/ price," the update controller will establish a one-way handshake and pass the executed data directly to the L2 broadcaster. Otherwise, based on the message type, the update controller will send out a combination of a 3-bit wide one-hot signal to correctly utilize the selected ask/bid controller for the book update. The ask/bid controller sub-modules have a direct read and write access to their own dedicated RAM modules and are responsible for maintaining each book.

The bid controller sub-module's function is explained below, while the function of the ask controller is simply reversed. The bid controller always keeps track of the maximum (top) and the minimum (bottom) price levels, as well as the location of the minimum price level within the RAM. When a new price level arrives with an add message type, the bid controller compares the incoming price level to the maximum and minimum price levels currently stored in the RAM. If the incoming price level is less than the minimum price level and the RAM is not full, the incoming price level simply gets pushed to the tail of the RAM and both the minimum price level and its address location are updated. For the same case but with a full RAM, the incoming price level is ignored. Now, when the incoming price level is greater than the maximum price level currently in the RAM, the incoming price

level gets pushed to the top of the RAM, the maximum price level is updated, and all the price levels below the new price level gets shifted down the RAM, one at a time. The more challenging scenario is when the incoming price level is within the bounds of the maximum and minimum price levels, which infers that the incoming price level needs to be iteratively compared to each existent price level in the RAM. When the incoming price level is matched to an existing price level in the RAM, the data in that address gets updated appropriately, according to the incoming message type. However, when no match is found in the RAM, the incoming price level is inserted into the RAM into its appropriate location and everything below the inserted price level gets shifted down, one level at a time. If the RAM was already full, the last element (minimum price level) gets thrown out of the RAM.

Further, as a RAM is being updated, its ready signal goes low, signifying that that particular RAM is busy. The concatenated 6-bit long one-hot ready signal is passed back to the update controller to monitor the update activity. Unless the L2 broadcaster is busy, once a particular RAM is done updating, the update controller sends out an enable signal along with the encoded RAM selection (hashed stock symbol and ask/bid book selection) to the L2 broadcaster module notifying it that the L2 has just been updated. At this point, the L2 broadcaster establishes a reverse handshake with the L2 book builder, pausing the operation of the L2 book builder, and iterates through the mapped RAM module via the direct memory access (DMA) protocol, bringing itself up to date. Since each RAM is only 10 levels deep, this operation takes approximately 10 clock cycles. However, for a larger number of price levels, a better topology should be used to improve the latency of the L2 broadcaster update operation.

Finally, the L3 tree climber sub-module is designed to perform the operation explained above for the case that comes after the L2 gets full and later an entire price level of the L2 becomes either fully executed or deleted. For this case, as the L3 tree climber is operating and/or waiting for the search to complete, it establishes a handshake with the update controller, temporarily shutting down the entire L2 book builder. Meanwhile, the incoming messages from the L3 broadcaster, if any, are queued in the FIFO. If search is successful, the results are stored in the FIFO as well. From then on, the L3 tree climber directly updates the corresponding RAM via its dedicated ask/bid controller. When the L3 tree climber is done, the L2 book builder resumes its normal operation, de-queuing the FIFO at the input from the L3 broadcaster.

A block diagram of the L2 book builder's operation can be seen in figure 12.

3.6 Broadcaster

The broadcaster receives data from the L2 book RAM and packages the data into a UDP packet to be sent through ethernet to a display. The module receives from the L2 book an update signal as well as an execute, an execute data, and ram access signals. When the update signal becomes high, the broadcaster reads in the ram access signal, which determines which RAM in the L2 should be accessed, and also checks the execute bit. The execute data is read into a register only if the execute bit is high and is ignored otherwise. Once update becomes high and the ram access signal is read, the module will send the access signal out along with an address, which iterates from zero up to the number of levels in an L2 book. The access signal serves to select the proper RAM to read from, while the address signal selects each row within a RAM. The L2 RAMs are expected to have a one-cycle delay, and so the RAM data will be read consecutively starting with the second cycle after the update signal becomes high. Data being read from the RAM is stores into a buffer until all the relevant data within a RAM has been read. The data at this point is ready to be outputted through ethernet to a display. A data packet is generated from the broadcaster by annealing a hardcoded header to the payload of the packet, which includes the data read in from each address of the RAM, as well as the execute data if the execute signal was high. Once the packet is generated, it is streamed to the output 64-bits at a time. As the output stage of the system, the broadcast must also comply with the protocol of the Avalon Streaming bus. Recall, that protocol is shown in figure 4.

3.7 Display of Output

The data display, implemented in software, is the receiving terminal for the ethernet packaged data sent by the broadcaster. The program listens to ethernet UDP packets, sent from the broadcaster and displays the raw contents of the ethernet packet.

4 Design Flow

While the typical design flow for FPGA based work is very well established, we had the rare opportunity to implement our design on a brand new hardware new to the field which often required us to adapt to certain situations. There were a number of obstacles we faced that slowed down our process at first but we were able to make changes to our flow to accommodate for them.

4.1 Algorithm Development

Because the goal of this project was not only functionality but also extreme efficiency, we were very careful in our initial development and went through many iterations of our proposed system before beginning implementation. We spent most of the first few weeks of this project developing different algorithms for the lookup and the L2 building. For the lookup, our first ideas included iterative searches, which quickly proved to be too slow. We then moved to a hash function approach, which still fell short. Finally, we agreed that the best way to implement this segment of the design was with a lookup tree. This would provide the most consistent and efficient results. For the L2, we originally planned to build a truncated version of the L3 book but only display a portion of the orders. We devised this method in order to improve efficiency but ultimately the concept didn't last. For one, the book will eventually run out of price levels and the L3 is necessary. For two, working to improve efficiency in the L2 book was not very useful when the bottleneck of the design still lies in the L3. Ultimately, we settled to a simplified algorithm which requests data from the L3 book when necessary and builds the L2 book 10 levels deep on each message passed.

4.2 Golden Software Model

Prior to implementing the design, a software model of the system was coded in Perl based on the sample Itch data posted by Nasdaq. This model provided us with intuition regarding what to expect off the wire as well as all of the data formats. This model was ultimately discarded when day old data was provided to us by Nasdaq.

4.3 Behavioral Design

The design of this system was implemented in VHDL. It was quite heavy on the logic design side and took a significant time to complete even the first pass of coding. Upon validating the modules (see below), many bugs were exploited and multiple iterations of each design were necessary. Ultimately, the design was functional at the behavioral level with some minor bugs.

4.4 Validation

The design was first validated at the module level using a suite of randomly generated vectors. For the ternary tree, vectors were streamed at the module and the entire RAM array was dumped to a file after each operation. The file was then brought into a Perl based flow which provided a visual interpretation of the tree as well as all necessary parsed information at each node. Another file was also generated from this module's validation which was a set of outputs towards the L2 FIFO. This file was then inputted to the L2 book builder module for extensive validation.

For the UDP and MoldUDP packetizes, a pcap file of real Itch data was streamed at the input of the packetizer. This proved to be a full proof method of validation.

4.5 Integration and Hardware Validation

Once module level validation was complete, the system was integrated. First the Packetizer was integrated with a model of our output broadcaster. This small system was synthesized and placed on the Stratix V chip. To validate the design on the chip we streamed the same pcap file to an SFP port on the AoE board using tcpreplay. The packets were seen to be received on the chip using Altera's Signal Tap tool and with some debugging, packets were successfully passed through.

Table 1: Expected Achievements of the Itch Ticker Plant

Specification	**Value**
Memory Allocation	$21Mb$
Add Order Latency	$205ns$
Delete Order Latency	$205ns$
Execute Order Latency	$180ns$
Modify Order Latency	$180ns$
Replace Order Latency	$180ns$
Delete Order Latency	$180ns$

The next module to be integrated was the MoldUDP packetizer. We put this build through the same validation flow and it proved to have some issues with timing. Although no violations were reported in timing analysis, we are sure that the method by which this module was implemented is leading to the bug. We are yet to sort this issue and will be redesigning this segment of the system.

Despite the bug in the MoldUDP (which did not show up in simulation), we were able to integrate the L3 builder to the system and see some success on the board. The L2 is yet to be integrated to the system at this point.

5 Results

At this point in time, we have not reached our goal of the full Nasdaq ticker plant working on the board. However, we have made significant progress and we plan to continue the project over the summer and next semester. We have seen success in simulation and on the board at times but there is a lot of work to be done to perfect the design and begin further optimization. Despite not completing this design in its entirety yet, we have tackled what is in our opinion the most difficult parts of the system and it is minor bugs that are holding us back.

Once the current build is complete, we will reach the following specs. The specs for latency can also be seen in figure 13.

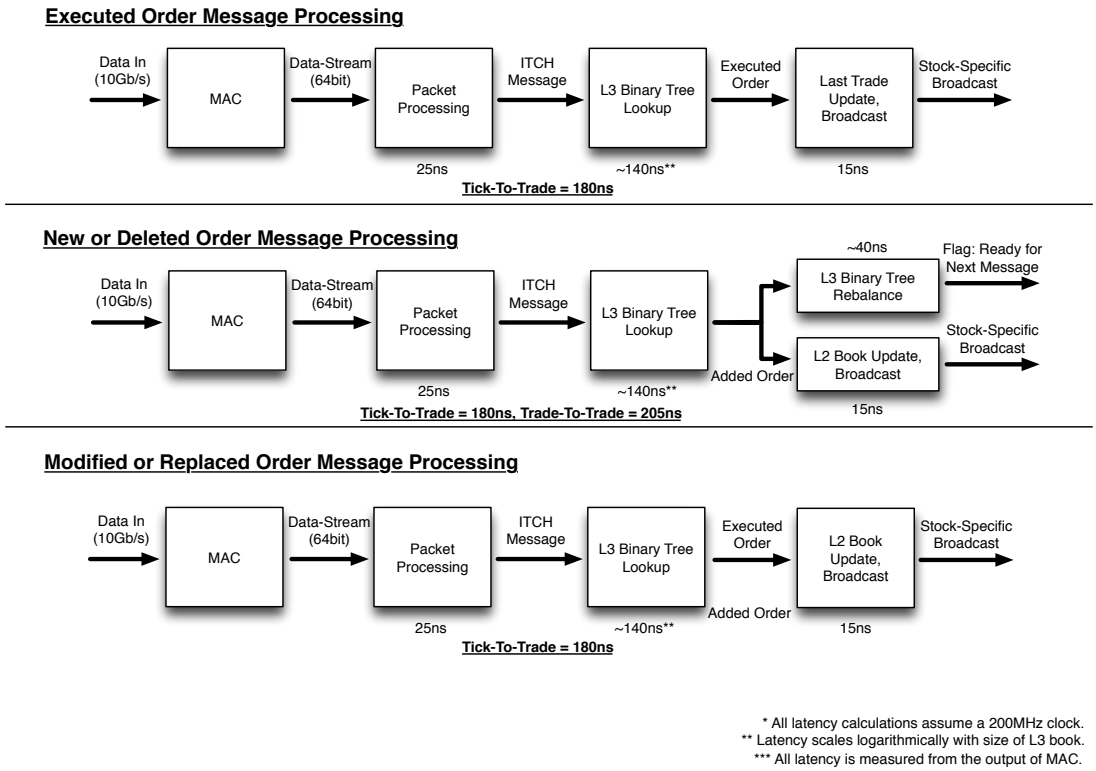

Figure 13: Latency Descriptions of the Most Common Itch Messages

6 Conclusion

Through the first segment of this project, we have learned not only about the do's and don'ts of design on a high technology piece of hardware, we have also learned a good deal about the stock market. We believe that with the knowledge attained this semester, we can continue this project over the summer and turn it into a real product that has relevance in the finance industry. Our design, once complete, will provide a high speed service to its users and will reach databasing speeds of the L3 book that no software design can reach. In addition, building the L3 book in hardware is such a difficult task, that it hasn't really hit the market yet. We plan to be the first to reach that accomplishment. Our plans for continuing work on this project are as follows.

- Complete the originally planned system.
- Increase the quantity of ternary trees and hash to them to decrease the latency.
- Utilize DDR3 memory on the AoE board to expand the functionality to industry level.
- Integrate our design with the fix/fast, CME projects.
- Implement the integrated projects as an in-line risk system with less latency than any other solution on the market.

7 Special Thanks To

- Solarflare
- Altera Corporation
- Nasdaq Stock Market
- Professor David Lariviere
- Professor Stephen Edwards