Rapid Decoding of Digital Data Streams Using Field Programmable Gate Arrays

Andrew Hernandez
CUNY City College

How does access to this work benefit you? Let us know!
Follow this and additional works at: http://academicworks.cuny.edu/cc_etds_theses

Part of the Computer Sciences Commons

Recommended Citation
Hernandez, Andrew, "Rapid Decoding of Digital Data Streams Using Field Programmable Gate Arrays" (2012). CUNY Academic Works.
http://academicworks.cuny.edu/cc_etds_theses/89

This Thesis is brought to you for free and open access by the City College of New York at CUNY Academic Works. It has been accepted for inclusion in Master's Theses by an authorized administrator of CUNY Academic Works. For more information, please contact AcademicWorks@cuny.edu.
Rapid Decoding of Digital Data Streams Using Field Programmable Gate Arrays

THESIS

Submitted in partial fulfillment of the requirement for the degree

Master of Engineering (Computer Science)

At

The City College

of the

City University of New York

by

Andrew Hernandez

December 2010

Approved:

__________________________
Professor Izidor Gertner, Thesis Advisor

__________________________
Professor Douglas Troger, Chairman

Department of Computer Science
Contents

Acknowledgements ......................................................................................................................... 5
Introduction .................................................................................................................................... 5
Literature Review ............................................................................................................................ 6
Core Concepts ................................................................................................................................. 7
  Securities Trading ........................................................................................................................ 7
  NYSE Arca .................................................................................................................................... 8
    Market Share ........................................................................................................................... 8
    Arcabook .................................................................................................................................. 9
  Message Rates ........................................................................................................................... 9
  Data Rates ................................................................................................................................ 9
Time Matters ............................................................................................................................... 10
Operations ..................................................................................................................................... 11
  Parsing .................................................................................................................................... 11
  Storage ................................................................................................................................... 12
  Retrieval ................................................................................................................................. 12
Storage Message Types .................................................................................................................. 12
  Add ......................................................................................................................................... 12
  Modify ..................................................................................................................................... 12
  Delete ..................................................................................................................................... 13
Processing Options ......................................................................................................................... 13
  Binary vs. Compacted ............................................................................................................... 13
Methodology ................................................................................................................................... 14
  Data Homogeneity .................................................................................................................... 14
  CPU ............................................................................................................................................ 14
    Specification .......................................................................................................................... 14
    Procedure .............................................................................................................................. 15
  GPU ............................................................................................................................................ 15
    Specification .......................................................................................................................... 16
    Procedure .............................................................................................................................. 16
  CUDA ....................................................................................................................................... 16
    FPGA ...................................................................................................................................... 17
Hernandez, Rapid Decoding of Digital Data Streams Using Field Programmable Gate Arrays | 2
Acknowledgements

Axiom Markets LLC
Tag Team Trading LLC
New York Stock Exchange Arca
Professor Izidor Gertner (CCNY)

Introduction

In today’s modern marketplace, the overwhelming majority of financial exchange transactions are completed entirely on computers. These transactions range from shopping online, to transferring money between accounts, and they even include high frequency securities trading. This is especially true in competitive marketplaces like the New York Stock Exchange (NYSE) or the Chicago Mercantile Exchange (CME). The NYSE provides an environment for buyers and sellers to trade stock in companies that are registered for public ownership. The CME does the same for commodity futures contracts like oil or gold. In recent years, matching buyers and sellers has been taken out of the hands of human regulators and given to computers. Now the overwhelming majority of these transactions are handled completely electronically. Going even further, even the human trader, who initiates the trades, is being replaced by black box computer algorithms. It has become a race to show who has the fastest systems. As a result, these exchanges are fast moving and can be very chaotic. Securities trade hands multiple times per second and prices change just as quickly. On the micro level, prices change seemingly without warning or reason. In order to remain competitive, rapid access to accurate market data is mandatory. A superior electronic trading platform offers many advantages.

One obvious advantage is higher profits. If I have access to a timelier snapshot of marketplace data than a competitor, I will be in a position to react sooner, taking advantage of opportunities before anyone else. I could execute trades before competitors and subsequently capture the profits before they do. The ability to do this over and over the course of days and weeks would make me a leader in the industry, and makes the value of my company greater. Many high profile companies are engaged in this type of work. These companies include Citigroup, Goldman Sachs, and Merrill Lynch. Over the last few years, these large financial institutions have been employing highly intelligent researchers in the fields of math and science. All this in an effort to create even faster and more advanced trading platforms. Typically, these platforms have been developed and deployed on commodity based hardware, that is, general purpose x86 microprocessors. These processors are excellent for many functions, but they do
not offer many advantages in terms of speed or efficiency. I want to implement these operations on specially designed hardware.

A well designed hardware solution will outperform an x86 microprocessor solution since it has far less overhead and a much shorter critical path. A design such as mine would be immediately deployable in industry and provide any company with a technological distinction and advantage over its competitors.

There are a large number of considerations that must be taken into account when designing a hardware solution for these types of operations. The chief concern is that of data manipulation. Financial data is typically provided in a binary or encoded stream that must first be parsed, second stored and, third retrieved. Optimizing all three of these operations is essential. A bottleneck at any point will slow down the entire system.

**Literature Review**

String parsing, storage, and retrieval are not new concepts, nor are efforts to speed up electronic trading platforms. My investigation is primarily focused on the advantages gained using streams of financial data, but the same concepts can apply over many different fields. This technology could provide benefit to any industry that has need of rapid manipulation of raw data in real-time data.

IBM has a high performance xml parser solution called lxml\(^2\). This is a library of functions written in Python that focuses on decoding xml data rapidly\(^7\). My research does not utilize xml directly, but many of the same principles apply. Xml is used to provide metadata and processing information for many real time data streams. The xml language is largely composed of plain text strings, so having the ability to rapidly parse those strings to determine their exact nature, and store the gleaned content would be highly useful. The lxml library allows for the processing of extremely large data sets, 2 GB or more. IBM’s library is a good starting point, but my focus is on speed and a pure software solution it is not sufficient.

Another project dealing with these same concepts is the human genome project. Completed in 2003, the human genome project was under way during the late nineties and early two thousands. During those years, a number of different agencies and scientists were able to achieve some great goals\(^5\). They identified all the numerous genes, and determined the sequence of the chemical base pairs that make up human DNA. They stored this information about DNA and chemical base pairs using a series of string characters. The human genome
project required a string parser and string storage application. This led to another goal of the project, to improve tools for data analysis of human genes and DNA. In general, DNA is represented by series of repeating strings or characters. To optimize process the project developed software that handled string manipulation. The tools developed by the human genome project applied directly to their specific application, and do not necessarily translate to optimizations for more general purpose data sets. Data sets like those I dealt with in my research. Again their solutions were developed as software and are not as optimal as the hardware solutions I am proposing.

Microsoft markets a product called BizTalk Server8. BizTalk is a specialized service bus that speeds up certain types of business communication. Through the use of adapters, data streams can be translated from one technology into another. It is also useful for extracting information from data feeds and converting it to plain English. These types of operations are very similar to what I am trying to accomplish with my research. A fine solution, but as with the other existing systems I mentioned, this is a software solution implemented on commodity hardware.

My project deals with a data protocol called FIX or Financial Information eXchange. This format is ubiquitous within the industry16. Almost every exchange and financial server provider has at least some data transmitted in FIX. It is so commonplace that there are many companies that provide systems that exclusively handle data in FIX format. These companies include Rapid Addition, SAVVIS, and Realtime Systems Group (RTS)18. I extensive experience with the RTS solutions. They provide software that parses and even stores data feeds that are identical to what I deal with in my thesis. Their software is fast, reliable, and efficient - but again, it is software. The key difference between this, the other aforementioned solutions and my research is my solution is implemented on the hardware directly. The others use software. A proper hardware solution will be much faster, a better use of resources, and create a competitive edge that would be difficult to match.

Core Concepts

Securities Trading

The core impetus for this thesis is to improve on already existing technology as it is related to the field of securities trading. A security in this context is a fungible and negotiable instrument that has financial worth itself or abstractly represents that value. Typical types of securities are stocks, bond, futures contracts, options and swaps. Securities trading began centuries ago and has evolved in parallel with business practices, government regulations, and technology.
innovations. There are a myriad of reasons an individual or institution wants to trade securities. These range from speculation and a desire to make money to a hedge against wealth eroding factors like inflation. Regardless of the reasons, in order to deal in securities one must understand the market, what instruments are available to them, and how technology drives the whole system. Just like the reasons for doing so, there are numerous ways to trade securities. I can deal with an opaque market like a bank or credit union. I can trade over the counter with other individuals or companies, or I can go to an open exchange. My project deals with trading on an open exchange. There are many exchanges, such as the New York Stock Exchange, the NASDAQ, and the Chicago Mercantile Exchange. These exchanges are open to anyone who wants to trade. They are completely transparent, meaning anyone can see details of any transaction. Due to this open and transparent structure, the security markets on these exchanges are very competitive. Technology advantages can go a long way. Different exchanges can have slightly different features. I focused on the New York Stock Exchange (NYSE) Arca, due to its completely electronic nature.

**NYSE Arca**

NYSE Arca, formerly known as ArcaEx (an abbreviation of Archipelago Exchange) is a securities exchange on which both stocks and options are traded. NYSE Arca is an ideal exchange for my experimentation. Unlike many other exchanges, NYSE Arca is fully electronic. This means no trades occur on an open outcry trading floor. Trades done on a trading floor are often not disseminated through the electronic reporting system in real time. This component levels the field for everyone involved. Since it is fully electronic, technology plays an even larger role in gaining a trading advantage.

**Market Share**

As of 1 March 2007, NYSE Arca is the second largest Electronic Communication Network in terms of shares traded. Approximately one out of every six shares traded on the American financial markets is traded on the system. For New York Stock Exchange-listed securities or Tape A, it accounts for just over 10% of the shares traded. For NASDAQ-listed securities, NYSE Arca accounts for approximately 20% of the trading volume. For exchange-traded funds, NYSE Arca accounts for 30-40% of the traded volume. This volume makes NYSE Arca and even more ideal place to do my experimentation. A large amount of volume means a large amount of data.
**Arcabook**

The data distribution service for NYSE Arca is called Arcabook\(^1\). This service is provided directly from NYSE Arca and contains all the raw data of orders and transactions that have taken place on the exchange. As underscored by the previous section there are a tremendous amount of orders placed and executed on NYSE Arca. This equates to a substantial amount of streaming data from Arcabook. The following two sections outline those message (a complete unit of data) rates, and raw data rates.

**Message Rates\(^{10}\)**

<table>
<thead>
<tr>
<th>Arcabook</th>
<th>Current</th>
<th>2011 end of year projected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak message per second rate</td>
<td>310,000</td>
<td>450,000</td>
</tr>
<tr>
<td>Packet size</td>
<td>Variable</td>
<td>Variable</td>
</tr>
<tr>
<td>Maximum number of packets in a day</td>
<td>300,000,000</td>
<td>500,000,000</td>
</tr>
<tr>
<td>Maximum total number of individual book messages in a day</td>
<td>1,250,000,000</td>
<td>2,000,000,000</td>
</tr>
</tbody>
</table>

*Table 1 - Arcabook Message Rates*

**Data Rates \(^{10}\)**

<table>
<thead>
<tr>
<th>Stream</th>
<th>Current Rate Mbps</th>
<th>2011 end of year projected</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTC only</td>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>LX only</td>
<td>33.5</td>
<td>54.5</td>
</tr>
<tr>
<td>ETF only</td>
<td>28</td>
<td>45.5</td>
</tr>
<tr>
<td>BB only</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>All depth of book subscriptions (total)</td>
<td>77</td>
<td>125</td>
</tr>
<tr>
<td>Refresh Interval</td>
<td>40</td>
<td>65</td>
</tr>
<tr>
<td>Retransmission</td>
<td>40</td>
<td>65</td>
</tr>
<tr>
<td>Refresh Request</td>
<td>20</td>
<td>32.5</td>
</tr>
</tbody>
</table>

*Table 2 - Arcabook Data Rates*
Time Matters

Successful and profitable trading is largely determined by two factors: strategy and connectivity. Strategy includes concepts such as when to buy, when to sell, and how much to buy and sell. For my purposes, I am not concerned with these concepts as they lie outside the scope of this thesis. I assume that any hypothetical strategy is sound. Instead, I am more focused on the second factor, connectivity. Primarily, I am concerned with how quickly I can interpret and act upon incoming data. Reducing time required to make a decision leads to advantages over competitors.

For example, say I have a strategy that tells me to buy 1000 shares of XYZ every time it reaches a certain price. If the price is very good, there may be many investors that would like to purchase at that price. Likely, only a limited number of shares exist at the price at which I want to execute. I need to act faster than the others that also want to buy so I can secure my position. If I have a computer system that is able to act more quickly than my competitors I can ensure I get the shares I want at the price I want.

There are many components that determine exactly how much time is required from the beginning of a favorable market event to the execution of an order. First, market conditions are broadcast out by an exchange (NYSE Arca via Arcabook in my case). These messages include the current state of the order book, that is how many buy and sell orders exist, the quantity and price, for a specific product. There is network latency between the exchange and the local computer which will implement my strategy. This value is determined by my distance from the exchange and the type of connection I have. Next, I must process the data. This data comes in on a network stream; it may be encoded or raw. Either way, I must parse it out to get to the relevant information I require for my strategy. This is the process I hope to accelerate. After I have actionable data, I determine is anything is to be done, I create a response back to the exchange and send it off. Figure 1 shows a diagram depiction generally how this process happens.
Operations

There are three major operations that must be performed when decoding digital data streams of financial information. These operations relate to message types that flow into the system from the Arcabook data feeds. A more complete look at Arcabook message types and message field breakdowns are shown in Appendix D – Arcabook Messages.

Parsing

The first operation is parsing. It primarily involves taking a message from the data stream in its raw format, stripping away the pieces of data that I do not need, and pulling out the pieces of data I am concerned with. Depending on the objective of the trading strategy there are different pieces of information I need. The data fields in there raw form can be found in Appendix D. For example, I will almost always be interested in fields that concern prices, quantities and side, that is, buy or sell. I have less interest in fields like exchange timestamps and sequence codes. This can be a time consuming operation since I am required to randomly access different parts of a buffer array that holds each individual message and reform it into a smaller structure that contains just the information that I am concerned with.
Storage

Once the raw data has been parsed, it is ready for storage. The storage phase consists of three sub operations: add, modify, and delete. The message types for those sub operations are analyzed in the following section. Each is handled a little bit differently, but they each essentially operate in the same way by making a change to the locally stored order book. Each performs the specified operation on the underlying data structure, in my case, a dictionary structure. Depending on the type of message, the data structures are adjusted to reflect current market conditions.

Retrieval

The final operation is retrieval. Data that flows in is parsed out, and stored in a data structure. I must be able to retrieve that data structure at any moment in order for the human trader or a computer program to see the complete snapshot of market conditions. That snapshot is based on what is in the data structure, and using it, decisions can be made about whether to buy or sell.

Storage Message Types

Add

The first and most common message is the add type. A detailed breakdown of the message can be found in Appendix D. The add message type is disseminated over the data stream every time a new order is placed at the exchange. When an add message occurs I create a new entry in the order book data structure. That entry holds the specific information about the order, such as price, quantity, and order number.

Modify

The second and least common of the three types is the modify message. Again, the details of the message are shown in Appendix D. This message is very similar to the add type, but rather than representing a brand new order at the exchange, it’s an update to an already existing order. Modifications can occur in price or quantity. Modifications occur infrequently because the typical method of modification is to delete an existing order and add a new one (called a cancel/replace). When this message is received I use the order number to reference the stored order in the data structure and make whatever modifications are necessary.
Delete

The last message is the delete type. This occurs very frequently. Delete happens every time an order is removed from the exchange. This removal can happen because it was deleted by the originator, filled by counter-party, or canceled by the exchange. Also, removals can happen if the order expires because the market is too far away from it. As with the other the specific details can be seen in Appendix D. When I receive this message, I used the order number to reference the entry in the data structure, and I remove it.

Processing Options

There are many hardware and software combinations that could potentially process the Arcabook data streams. I explored three different possibilities. A typical and easily employable solution is pure software running on an x86 processor. This application could be nearly universally deployed and maintained. Conversely, this type of processor has little specialization for the operations I require. The second option is a graphics processor unit or GPU. This processor has distinct processing components to optimize certain types of calculations. This option is intriguing and worthwhile to explore due to new interest in General Purpose GPU (GPGPU), but this hardware is more appropriate for parallel operations. I am more concerned with fast serial processing. The final option is a field programmable gate array or FPGA. This should be the best since I can make the hardware gates function in any way I desire.

Binary vs. Compacted

The NYSE Arcabook data service offers two different types of data streams, binary and compacted. Binary streams are simply the raw data presented in a predictable, unadjusted format. Compacted streams implement the Financial Information eXchange (FIX) FAST protocols to reduce the total number of bytes sent to each client. FIX is a messaging standard used for real-time electronic transfer of securities transactions. FAST provides additional features that enhance the FIX standard. Each stream has its advantages. The major tradeoff of the two is between bandwidth and processing time. The binary streams do not need to be decoded first, but the compacted streams use less bandwidth. My experimentation focuses on one core concept: speed. For my purposes, bandwidth is not an issue. Therefore, the compacted streams are ruled out immediately. The extra step of decoding FIX compacted data will always make processing those streams slower than the binary.
Methodology

While there exist many ways to effectively speed up the processing of this financial data, my research focused on the three core operations of data manipulation previously outlined: parsing, storage and retrieval. These operations must be performed thousands of times a second and millions, even billions, of times per day. At that rate, even a single feed can provide an overwhelming amount of data. There exists countless different ways to realize these operations. One highly efficient method is with use of an FPGA. In order to show that effectiveness, I also developed similar programs using a general purpose x86 CPU and a specialized GPU.

Data Homogeny

After initial testing, I discovered that having each solution run using a live Arcabook feed was not ideal. Since the stream never repeats, results from these tests could never be accurately compared with one another. Since financial markets are always changing, for my intents and purposes, the data that was provided was random. This means that one solution may have been getting an easier load of data to process. In order to correct this problem, I recorded live data from an Arcabook feed. I then created a simulated feed using this data that could be replayed. This allowed for each solution to be tested on the same set of data and permitted a more exact comparison of the solutions.

CPU

This first and most obvious choice for these operations is a commodity x86 microprocessor. This type of hardware is universally available and provides all the necessary, albeit general, elements to carry out the core operations. Specifically, I used an Intel Xeon quad-core processor running at a speed of 2.5 GHz (more detailed specifications are provided in Table 3). Basic parsing and storage functions are coding in the C# programming language. C# is an excellent language, and is commonly used in the financial industry. It provides a good balance of ease of use and efficiency. There are five operations that I am concerned about parse, add, modify, delete, and retrieve.

Specification

The specifications for the CPU used in my procedure are as follows:
**Table 3 - CPU Specifications**

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU Name</strong></td>
<td>Intel Xeon E5420</td>
</tr>
<tr>
<td><strong>CPU Code Name</strong></td>
<td>Harpertown</td>
</tr>
<tr>
<td><strong>CPU Socket Type</strong></td>
<td>Socket 771 LGA</td>
</tr>
<tr>
<td><strong>CPU Technology</strong></td>
<td>45 nm</td>
</tr>
<tr>
<td><strong>CPU Core Speed</strong></td>
<td>2.50 GHz</td>
</tr>
<tr>
<td><strong>CPU Bus Speed</strong></td>
<td>332.5 MHz</td>
</tr>
<tr>
<td><strong>CPU Total Cores</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>Motherboard Name</strong></td>
<td>Dell 0RW199</td>
</tr>
<tr>
<td><strong>Motherboard Chipset</strong></td>
<td>Intel 5400B</td>
</tr>
<tr>
<td><strong>Motherboard Total Ram</strong></td>
<td>4096 MB</td>
</tr>
</tbody>
</table>

**Procedure**

Each of the five operations was measured individually. This helped me to identify bottlenecks in the processes and it served as a guide in designing my hardware solution. In order to reduce the impact of measurement on the overall time, measurements were taken every fifty operations and averaged. With a CPU solution there are many factors to consider. First, the core speed and capabilities of the CPU itself play a large role in the overall speed. I purposely chose a more powerful processor for this portion of the experiment. While even faster CPUs do exist, I think this one give a good general sense of what a CPU solution can do. Second is the efficiency of the code itself. A better code will give faster results, again, more optimum solutions are likely to exist, but the code I have created provides a good general sense of what this solution is like. Finally, on a general purpose CPU, it is difficult to filter out how much delay is caused by the other duties of the microprocessor. These include running the operating system, background programs, and anything else that is running concurrently with the program. In my experiment, I strived to reduce the total amount of extraneous work done by the CPU and focus on the solution.

**GPU**

The second solution is to use a GPU (also called visual processing unit or VPU) for the five functions described in the previous section. A GPU or is a specialized processor that offloads 3D graphics rendering from the microprocessor. A GPU solution consists of many of the same elements and therefore many of the same caveats of the CPU solution. GPUs are built for different kinds of operations and accordingly offer different functionality than a CPU.
a typically good for performing operations in parallel. Parallel operations can offer an advantage over a serial operation in terms of speed and efficiency\textsuperscript{17}.

**Specification\textsuperscript{4}**

The specifications for the GPU used in my procedure are as follows:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Name</td>
<td>NVIDIA GeForce GTS 250</td>
</tr>
<tr>
<td>GPU Code Name</td>
<td>G92</td>
</tr>
<tr>
<td>GPU Technology</td>
<td>65 nm</td>
</tr>
<tr>
<td>GPU Core Speed</td>
<td>738 MHz</td>
</tr>
<tr>
<td>GPU Total Memory</td>
<td>1024 MB</td>
</tr>
<tr>
<td>GPU Memory Type</td>
<td>GDDR3</td>
</tr>
<tr>
<td>GPU Interface Type</td>
<td>PCI-Express x16</td>
</tr>
</tbody>
</table>

Table 4 - GPU Specifications

**Procedure**

The procedure is similar to the procedure with the CPU. I measured each of the five operations independently, and recorded the results of the tests. These results are reported in a subsequent section.

**CUDA**

NVIDIA (the manufacturer of the GPU previously specified) provides a special computing architecture that runs on their chips. It is called the Compute Unified Device Architecture or CUDA\textsuperscript{15}. A main feature of CUDA is its parallel execution nature. With CUDA many general purpose, i.e. non-graphical, problems can be solved on a GPU. CUDA has been shown to be very useful to accelerate problems in biology, cryptography and other complex fields. An additional feature of a GPU is access to shared memory. CUDA focuses on running programs on many threads in parallel rather than one thread very fast. A shared memory region is accessible by all threads, allowing for faster throughput. Code that incorporates CUDA can be compiled in a number of different languages including C, Fortran, Java, .Net and MatLab.
FPGA

The final solution is an implementation of the abovementioned functions using a field programmable gate array (FPGA). FPGAs are integrated circuits with a feature of being configurable after manufacture. FPGAs offer many advantages over CPUs and even GPUs, primarily with regard to speed. Using a hardware description language, such as VHDL, I can specify the physical layout of a circuit with the ability to parse and store the data I am concerned with. Implementing a solution at the hardware level is faster due to specialization, and by having a dedicated device to provide the operations. A CPU or a GPU must perform other functions needed to maintain an operating system and other running programs, while an FPGA does not. The FPGA should provide the fastest string parsing and data storage among the three tests. Altera, a producer of a large number of FPGAs with wide range of functionality, manufactures the chip I used. I chose the latest Statix IV chip to take advantage of its speed and functionality, the core features of which are shown in Table 5 below.

Specifications

The specifications for the FPGA used in my procedure are as follows:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA Name</td>
<td>Stratix IV</td>
</tr>
<tr>
<td>FPGA Data Rates</td>
<td>Up to 11.3 Gbps</td>
</tr>
<tr>
<td>FPGA Power Consumption</td>
<td>100 mW</td>
</tr>
<tr>
<td>FPGA Ram Support</td>
<td>DDR, DDR2, DDR3, SDRAM</td>
</tr>
<tr>
<td>FPGA Interface Type</td>
<td>PCIe</td>
</tr>
</tbody>
</table>

Table 5 - FPGA Specifications

Procedure

Again, the procedure is similar to the experiments with the CPU and GPU. I measured each of the five operations separately. I recorded the results of the tests. These results are reported in a subsequent section.

VHDL

There exist a few different languages with which I could program my FPGA solution. I chose VHSIC hardware description language, or VHDL. VHDL was developed in the 1980s by the US Department of Defense to remedy a number of issues relating to hardware design. Over the
years it has been updated and revised and is now a top choice when designing hardware. I used Altera’s Quartus software to build my hardware solution with VHDL. Inside the software, individual modules can be prototyped, modeled, and unit tested. The modules are then interconnected inside the program and the complete hardware solution can be simulated. The software also provides an interface with the FPGA chip itself.

**Parsing**

There are a total of the eight modules that are used in the parsing process. Ideally, data flows through them freely and is independent of a clock signal, this allows for maximum speed. The first is similar to a random access memory (RAM) module, but is more narrowly focused. Its functionality is to act as a buffer to store each incoming message from the network socket. It is capable of storing 16 words of width 32 bits for total of 64 bytes. Each message is loaded into the RAM module one word at a time. Once the entire message is loaded, the module is turned off for writing, and the parsing process begins. The segmentation of data in this module allows for finer degree of retrieval during the parsing process. During parsing, some parts of the message play a more important role than the others. This segmentation enables me to grab important data that may be embedded in the middle more quickly. Figure 2 below shows the block diagram as it appears in the Quartus software, its underlying VHDL code can be found in Appendix C.

![Figure 2 - Ram Buffer Module](image)

The next block allows me to change the order of the data stream from network order to byte order. As in the previous parsing techniques on the CPU and GPU, this step is important so I can deal with the message in the correct order. Since the operation is simply switching the order of bytes on the bus wire, there is no cost in processing time. It can be wired so that the last by becomes the first byte and vice versa. Data can flow straight through without the need
for an extra clock cycle. The block diagram is shown in Figure 3 below, and its underlying code can be found in Appendix C.

The next phase in the parsing process requires me to look at the incoming message for the chunk of data that identifies the message type. There are three possibilities: add, modify and delete. The function of this block, shown in Figure 4 below, is to strip out that specific part of the message and enable the next phase accordingly. This module will read in the message, look at just the part that is of concern (the message type), send the message through the output, and send an activation signal to the module that corresponds with the message type. As with the previous block, there’s no time cost here since data flows straight through without clock advancement. The code is found in Appendix C.
The next three modules in the parsing process are designed specifically to handle the three message types, and, depending on the message, set-up the subsequent storage phase. The simplest of these three blocks is the delete module. Since the delete message does not require me to extract any additional data from the incoming message, on the next clock signal I can simply clear out the storage location at the corresponding address. The add and modify modules work in similar ways. They prepare a storage area in RAM to accept new data, or modify existing data on the next clock signal. They also activate the data extraction module so it is prepared for either an add or a modify message. The three individual blocks are shown in the three following Figures and their codes can be found in Appendix C.
The output of the previous three modules needs to be fed into the storage component. Since the component can only take one input, I need a way to select from the output of the add, delete, and modify modules. The next piece of the circuit is a multiplexor, shown in Figure 8, with underlying code in Appendix C. This takes the three outputs from the previous stage and conflates them down to a single line out. I can select with input goes to the output, and the subsequent storage module using a two bit selection signal.
The final stage before storage is a simple latch. This allows me to momentarily store the message ID number obtained during parsing during the storage phase. Without this latch, as soon as the input to the parsing stage changes, it would propagate into the storage stage. This latch prevents that problem and keeps the essential data in place to be processed. The data inside the latch remains there until a signal is fed to the gate input as shown in Figure 9 below.

Storage

Two modules are used in the storage process outside of the signals received at the end of the parsing process. At the end of the storage process, data is stored in a block called parsed RAM; this block is simply another random access memory component. Using real world data, order
books can grow quite large, especially for a large centralized exchange like the NYSE ARCA. Being able to store each order individually requires a substantial amount of RAM. Here the RAM is only 65536 words long. In reality that length should be much larger, somewhere in the order of gigabytes. Word length is 32 bits; Arca specifies their order ID have a length of the 4 bytes. Since every order is unique, I can take advantage of 32 bit word length by simply storing each order to a RAM location that corresponds to the order ID. This allows for maximum processing speed, since I am not required to perform any additional calculation such as a hash to come up with a unique storage location. This RAM module also allows for simultaneous reading and writing. This is ideal since I would wish to asynchronously be accessing this storage element for both read and write. Reading can take place at any time, and a real world system would likely be reading almost constantly. Writing only takes place when it is enabled by one of the three activation modules. The block diagram is shown below in Figure 10; underlying code is in Appendix C.

The final component used in the storage operation is the data extraction module. The key functionality here is to cut down the total message length. Initially, each individual message is quite long, almost 64 bytes. I want to remove any superfluous data that is not essential and just store the information I am concerned about. In reality, there are only a handful of fields of real concern, such as the symbol index, which identifies the underlying security quantity, the order price, and side (buy or sell). All other data such as source time, exchange IDs, and firm indices can be safely discarded. Depending on the input from the activation modules, the data extractor can operate in one of two modes, add or modify. Since add and modify messages are slightly different, the essential data lies in a different portion of the message, and therefore they must be extracted differently. In both cases, essential data can be pulled out from the message and reduced down to a 64 bit word length. Those 64 bit words are then fed in to the
parsed RAM module at the location that corresponds to the order ID. This is one of the slower operations in my system since it has to take time to read each of the words that make up the message from the segmented buffer. Block diagram is in figure below; see Appendix C for the VHDL code.

Retrieval

Retrieval is the final step in my system, and is also the most costly. There are two blocks involved in the reading process. One has already been covered, that is the parser module that stores the complete order book. The second is a simple counter module that cycles through the contents of the parsed RAM, causing it sequentially output the entire order book. Whenever a request for the order book is made, the module takes that single signal and then sets the read address to each memory address one by one at each new clock signal. This is a time consuming process requiring one clock cycle for each order. The exact time to complete the entire read is determined by the total length of the RAM. It can be made more intelligent by storing the last order ID number received and stopping and that point, reducing the number of extra clocks signals wasted reading empty data. The block diagram is shown below in Figure 12.
Complete Design

The complete design with all the inter-module connections is show in the next figure. Data flows from the left side to the right side. On the right side, a single output pin is provided to supply the retrieved message. All the data is written out in parallel for maximum speed.

Results

The results of each solution are presented, analyzed and compared in the following sections.
CPU

The results for the CPU solution are presented first and will be used as a baseline against which I will compare the other solutions.

Parsing

The first step in the process is parsing; it is displayed first in Figure 14. The results are about what I expected. There is a larger concentration at the left pushing down toward $10^{-6}$ with a steep then gentler slope down up until about $10^{-2}$. This makes sense; for the most part the operation is very quick, unless the CPU is otherwise engaged. Since there are numerous other activities the CPU could be performing the delay varies greatly. Most of the delays are not very long but some can last one hundred of a second.

Figure 14 - Parsing Data from Network Stream (CPU)
Storage

Add

The next operation to illustrate is add. The histogram of the combined trials is shown in Figure 15. The results here are bit different than those of the parse operation. There are two areas of note. First the spike on the left side, about half the time the operation takes between $10^{-7}$ and $10^{-6}$ seconds. This is the optimal performance on the CPU. The other half of the results are in the $10^{-6}$ and $10^{-5}$ ranges. This is caused by the CPU having to perform other operations. Similar to the delays during parsing, if the CPU has to perform other tasks, adding to the data structure may be delayed.

![Figure 15 - Adding to the Data Structure (CPU)](image)
Delete

The results of the delete operation are very similar to those of the add operation. Again, there are two notable areas: the spike on the left is in the same range as the one in the add histogram. The left area again represents optimal performance. There is also a group of trials further to the right. This group is a little bit slower than the same area on the add operation. Deleting an entry from the data structure can be a little bit slower. To add, I can just put the new entry into the structure, but to delete I must first retrieve the entry then clear it.

![Deleting Entry from Data Structure](Image)

Figure 16 - Deleting from the Data Structure (CPU)
Modify

The last of the storage operations is modify. The modify trials were much different than add and delete as shown in figure. There is only one concentration of results that fall nearly entirely into the $10^{-5}$ to $10^{-4}$ ranges. This makes the modify operation the slowest of the three. This would be expected, the modify operation is more complex than add or delete. Here I have to retrieve an entry in the data structure, update it with the new content and reinsert it into the structure.

![Figure 17 - Modifying the Data Structure (CPU)](image)
Retrieval

The last set of data if for the retrieval operation. The results here are quite typical. The data follows a regular bell shape center just below $10^{-5}$ seconds. The complete graph is in Figure 18.

![Graph showing retrieval data](image)

**Figure 18 - Reading from the Data Structure (CPU)**

GPU

The GPU results are very similar to those of the CPU. This is expected since a GPU and CPU would approach the procedures in a similar way. GPU advantages are more significant when performing parallel operations. My stream of financial data is serial by nature, so those advantages are not very pronounced in my results. Nevertheless, a GPU does offer a few other advantages. The GPU is not concerned with many of the operations the CPU must perform, like those to maintain an operating system and other running programs. Clock speeds and overall transistor topologies are different as well.
**Parsing**

As before, the first graph presented is for parsing, it is shown in *Figure 19*. I found these results particularly interesting. First, the elapsed time for this operation on the GPU seems to push harder down in the $10^{-6}$ range. Overall, more results are down in that areas, meaning that there is some speed gain in the GPU. The shape of the histogram as a whole is different from that of the CPU. The CPU showed a nice curve with results tapering off as they approached the $10^{-4}$ range. Here there are two distinct peaks, one around $10^{-6}$ and one at $10^{-5}$. There are fewer outliers beyond those ranges. This shows that there must be other factors that almost half the time, take the focus of the GPU off parsing. This could be any number of things. The GPU has many other responsibilities such as maintaining a graphical user interface (GPU) and any other tasks relating to graphics or video.

![Figure 19 - Parsing Data from the Network Stream (GPU)](image)

Hernandez, Rapid Decoding of Digital Data Streams Using Field Programmable Gate Arrays | 31
Storing

Next I present the three graphs that I created from the measurement data of the storage operations. The parsing operation was comparable to the CPU solution as far as time is concerned, but the storage operations as a whole seem to be slower on the GPU. I am beginning to conclude that the GPU solution is not an optimal one.

Add

First, is the add operation. Here the results seem to be contained mostly in the $10^{-6}$ to $10^{-5}$ range. The CPU solution was mostly pushing into the sub $10^{-6}$ territory. Again I see a similar dual peak pattern similar to the parsing results. To me, this suggests the normal operation of the GPU affects the latency of the add operation about half of the time.

![Figure 20 - Adding to the Data Structure (GPU)](image)
Delete

The delete operation is next. The aggregated data is show in histogram form in Figure 21. Again, I see the dual peaks; although, these are a little bit different. One of the peaks, the one showing the slower trials, is much less pronounced. As with the CPU most of the trials fall below $10^{-6}$, but they are a bit more spread out. Overall, as with add, the operation is slightly slower.

![Figure 21 - Deleting from the Data Structure (GPU)](image)

Modify

The data gathered from the last of the storage operations, modify, is shown in the following figure. This operation is a little bit slower than delete and slower than the same operation on the CPU. One significant detail is the dual peaks of previous operations are now almost gone. This suggests that the modify operation is interrupted less often by other tasks.
Finally, I will look at the results of the retrieval operation on the GPU. The data here is very interesting. This operation is slightly faster than in the CPU solution. The data still falls in the same $10^{-6}$ to $10^{-5}$ range but it is skewed more toward the former. Also as with the modify operation the double peaks are gone. This suggests that this type of operation is not affected by other operations performed by the GPU itself. The graph is shown in Figure 23.

![Figure 22 - Modifying the Data Structure (GPU)](image)

**Retrieval**

Finally, I will look at the results of the retrieval operation on the GPU. The data here is very interesting. This operation is slightly faster than in the CPU solution. The data still falls in the same $10^{-6}$ to $10^{-5}$ range but it is skewed more toward the former. Also as with the modify operation the double peaks are gone. This suggests that this type of operation is not affected by other operations performed by the GPU itself. The graph is shown in *Figure 23*. 

![Figure 22 - Modifying the Data Structure (GPU)](image)
Due to the sheer expense of a prototype board with a Stratix IV type FPGA, one was not made available to me for testing. Therefore, simulation results are the data gathered on a hardware simulator in Altera’s Quartus software. While not one hundred percent realistic (it provides more of a test in theory), it gives a good approximation of latencies. These results at least allow for comparison against the results of the GPU and CPU tests.

**Parsing**

The first result graph is for parsing. As with previous parsing tests, this graph shows the elapsed time for each parsing operation versus the number of times that elapsed time occurred. This graph is a little different; here the elapsed time axis is on a linear rather than a logarithmic
scale. It also shows that one hundred percent of the operations lasted the same amount of time, specifically about $1 \times 10^{-4}$ seconds. This is one of the great features of an FPGA. There is no overhead to run this and the subsequent operations. The hardware chip has only to do the operations I specify. This allows for a static and predictable latency. This helps to optimize strategies since I can know exactly how long an action will take to complete. While this is a desirable feature of the FPGA, it should be noted that these results are overall slower than the GPU and CPU tests. Those showed latencies in the $10^{-5}$ and $10^{-6}$ ranges. This is due to those processors running at much higher clock speeds. Here my simulation is limited to a slower speed. Although, this obstacle can be easily overcome; I could simply use a faster clock speed. The elapsed time for this operation is directly related to the clock speed. These results are in line with what would be expected. The parsing process is the slowest of all the operations.

![Figure 24 - Parsing Data from the Network Stream (FPGA)](image)

Storing

Next, the storage operations are discussed. Again, these are simulation results so the graphs show a single static value occurring over and over.
Add

Below, in Figure 25, I have displayed the results of the add operation simulation on the FPGA. Here I am beginning to see the real improvements of the FPGA over the CPU and GPU solutions. The time required to store a new message into RAM is about 1.6x10^-7 seconds. CPU and GPU solutions were in the 10^-6 range. Since this is the most common of the storage operations, this gain in speed will be very significant.

![Figure 25 - Adding to the Data Structure (FPGA)](image)

Delete

Now I will look at the results for delete. Delete is an extremely fast operation on the FPGA. Essentially, to perform a delete I need only to identify a single number from the incoming message (the order number) and clear that location in memory. The results in Figure 26 show this operation taking about 2x10^8 seconds. This is much faster than both the CPU and GPU. Deletes happen very often, having the operation sped up by that much provides a nice advantage.
Modify

Again, the last storage operation is modify. The results of the simulation of this operation are show in Figure 27. As with add and delete, the modify operation is much faster on the FPGA than on the CPU or GPU. This operation takes $1.6 \times 10^{-7}$. Notably, this is the same duration as the add message. This is expected since the modify operation is very similar to the add operation. Furthermore, in my design the operation for both is almost identical. The only difference is that the modify message modifies content already in memory, and the add loads new content. Either way, the data is parsed the same way and travels down the circuit via the same path.
Finally, I will discuss the last operation, retrieval. This result was surprising to me. This operation overall is very slow compared to the CPU and GPU. The data graph is shown in Figure 28. The elapsed time for this step is $6 \times 10^{-4}$. The CPU and GPU perform this same operation down in the $10^{-6}$ area. This increase in time is due to the limitations of the Stratix IV chip itself. The clock speed is limited on the Stratix IV chip and is substantially slower than the GPU or CPU. Given that my implementation retrieves data serially, this could be a drawback when many retrievals must be performed.

Figure 27 - Modifying the Data Structure (FPGA)
Conclusion

The FPGA solution is excellent. It can outperform the CPU and GPU solutions by implementing all necessary functionality on hardware rather than software. There exists one stipulation. The entire process is dependent on the clock speed of the hardware design. I used a 100 Mhz clock in my design. While this performed admirably, it did not speed up every operation. This is easily overcome by increasing clock speed. If I were to use 500 Mhz or even 1 Ghz I would see enormous speed improvements. In fact, for all three solutions, performance is dependent on clock speed. By nature, parsing data from an incoming data stream is purely a serial operation. I receive data at a fixed speed, and I have to process messages as they come in. The only true way to improve speed is to increase the rate at which the processor performs calculations. This means increasing the clock speed. There exist a myriad of solutions for parsing financial data feeds, or any data feed. These solutions are implemented in many different ways using many...
different technologies. By allowing these operations to be performed directly on hardware, I eliminated any bottleneck or extraneous calculations. This research shows that data feed parsing at this level is fast and efficient.

Abstract

My study reduces time required to parse, store, and retrieve information from high bandwidth digital data streams. I achieve this reduction with the creation of unique hardware based on field programmable gate arrays (FPGAs). I first created software that performs the aforementioned operations using a standard x86 microprocessor, as well as a dedicated graphics processing unit. Measurements taken using this software provided a baseline to establish the improvements achieved with my FPGA solution. I then created a virtual prototype of FPGA based hardware that performs the same operations. As this design was tested using real world data, I made improvements and eliminated bottlenecks in the operations. Ultimately, I achieved a design that converts a data stream into usable information as quickly as possible. My research has many potential benefits. I primarily focused on advantages gained using streams of financial data. By reducing the time needed to have usable information from a financial exchange, investors would have an advantage over competitors. Similarly, this technology provides benefit to any industry that has need of rapid decoding of real-time data. Much of my research is a continuation of similar efforts to increase data stream decoding performance such as those by the Human Genome Project or IBM’s lxml project. My work is distinctive by introducing the FPGA hardware element. Working at the hardware level, a greater level of time reduction can be achieved.
References


   <http://codespeak.net/lxml/>.


Figure List

Figure 1 - General Flow of Data ................................................................. 11
Figure 2 - Ram Buffer Module ................................................................. 18
Figure 3 - Network to Byte Order Module ............................................... 19
Figure 4 - Message Type Parser Module ................................................ 20
Figure 5 - Add Module ................................................................. 20
Figure 6 - Modify Module ................................................................. 21
Figure 7 - Delete Module ................................................................. 21
Figure 8 - Multiplexer ................................................................. 22
Figure 9 - Latch ................................................................. 22
Figure 10 - Parsed RAM Storage ......................................................... 23
Figure 11 - Data Extractor ................................................................. 24
Figure 12 - Retrieval Counter ............................................................... 25
Figure 13 - Complete Design with Interconnects ................................. 25
Figure 14 - Parsing Data from Network Stream (CPU) ......................... 26
Figure 15 - Adding to the Data Structure (CPU) ................................. 27
Figure 16 - Deleting from the Data Structure (CPU) ......................... 28
Figure 17 - Modifying the Data Structure (CPU) ................................. 29
Figure 18 - Reading from the Data Structure (CPU) ............................ 30
Figure 19 - Parsing Data from the Network Stream (GPU) ............... 31
Figure 20 - Adding to the Data Structure (GPU) ................................. 32
Figure 21 - Deleting from the Data Structure (GPU) ......................... 33
Figure 22 - Modifying the Data Structure (GPU) ................................. 34
Figure 23 - Reading from the Data Structure (GPU) ............................ 35
Figure 24 - Parsing Data from the Network Stream (FPGA) ........... 36
Figure 25 - Adding to the Data Structure (FPGA) ............................... 37
Figure 26 - Deleting from the Data Structure (FPGA) ......................... 38
Figure 27 - Modifying the Data Structure (FPGA) ............................... 39
Figure 28 - Reading from the Data Structure (FPGA) ......................... 40
Table List

Table 1 - Arcabook Message Rates ................................................................. 9
Table 2 - Arcabook Data Rates ................................................................. 9
Table 3 - CPU Specifications ............................................................... 15
Table 4 - GPU Specifications ............................................................... 16
Table 5 - FPGA Specifications ............................................................. 17
Appendix A – C# Codes

Parsing

Stopwatch stop = new Stopwatch();
    stop.Start();
    while (start <= end)
    {
        try
        {
            _messageRetransSocket.ReceiveFrom(bufRefReceive, ref _ep);
            _secondaryMessageRetransSocket.ReceiveFrom(secondaryBufRefReceive, ref _ep);
        } catch (SocketException e)
        {
            _messageRetransSocket.ReceiveBufferSize = 0;
            _secondaryMessageRetransSocket.ReceiveBufferSize = 0;
            break;
        }

        int seqNum = (int)IPAddress.NetworkToHostOrder(BitConverter.ToInt32(bufRefReceive, 4));
        int sSeqNum = (int)IPAddress.NetworkToHostOrder(BitConverter.ToInt32(secondaryBufRefReceive, 4));
        if (seqNum == start)
        {
            int bMsgType = BigEndToInt16(bufRefReceive, 2);
            sw.WriteLine(System.Text.Encoding.ASCII.GetString(bufRefReceive, 0, bufRefReceive.Length));
            // messag++;
            // if (messag >= 10000)
            // {
            //     sw.Close();
            //     sw = new StreamWriter("C:\\Temp\\file" + files++.ToString() + ".txt");
            //     messag = 0;
            // }
            if (bMsgType == Defs.msgType_GenericBook)
                GenericBookHandler(bufRefReceive);
            else
                NonBookHandler(bufRefReceive);
            start++;
            _backupData.TotalRecovered++;
        } else if (sSeqNum == start)
        {
            int bMsgType = BigEndToInt16(secondaryBufRefReceive, 2);
            sw.WriteLine(System.Text.Encoding.ASCII.GetString(secondaryBufRefReceive, 0, secondaryBufRefReceive.Length));
            // messag++;
            // if (messag >= 10000)
            // {
            //     sw.Close();
            //     sw = new StreamWriter("C:\\Temp\\file" + files++.ToString() + ".txt");
            //     messag = 0;
            // }
            if (bMsgType == Defs.msgType_GenericBook)
                GenericBookHandler(bufRefReceive);
            else
                NonBookHandler(bufRefReceive);
            start++;
            _backupData.TotalRecovered++;
        }
if (bMsgType == Defs.msgType_GenericBook)
    GenericBookHandler(secondaryBufRefReceive);
else
    NonBookHandler(secondaryBufRefReceive);
start++;
_backupData.TotalRecovered++;
}

bufRefReceive = new byte[2048];
secondaryBufRefReceive = new byte[2048];
i++;
if (i > 1000 || stop.ElapsedMilliseconds > 500)
{
    _messageRetransSocket.ReceiveBufferSize = 0;
    _secondaryMessageRetransSocket.ReceiveBufferSize = 0;
    break;
}

_messageRetransSocket.SetSocketOption(SocketOptionLevel.IP,
    SocketOptionName.DropMembership,
    moPrimary);

.secondaryMessageRetransSocket.SetSocketOption(SocketOptionLevel.IP,
    SocketOptionName.DropMembership,
    moSecondary);

while (_messageRetransSocket.Available > 0)
    _messageRetransSocket.ReceiveFrom(bufRefReceive, ref _ep);
while (_secondaryMessageRetransSocket.Available > 0)
    _secondaryMessageRetransSocket.ReceiveFrom(secondaryBufRefReceive, ref _ep);

int msgType = BigEndToInt16(bufReceive, 2);
sw.WriteLine(System.Text.Encoding.ASCII.GetString(bufReceive, 0,
    bufReceive.Length));
    //messag++;
    //if (messag >= 10000)
    {//
        // sw.Close();
        // sw = new StreamWriter("C:\Temp\file" + files++.ToString()
+ ".txt");
        // messag = 0;
        //}
    if (msgType == Defs.msgType_GenericBook)
        GenericBookHandler(bufReceive);
    else
        NonBookHandler(bufReceive);
} else if (missed == 0)
{
    int msgType = BigEndToInt16(bufReceive, 2);
    sw.WriteLine(System.Text.Encoding.ASCII.GetString(bufReceive, 0,
        bufReceive.Length));
    //messag++;
    //if (messag >= 10000)
    {//
        // sw.Close();
        // sw = new StreamWriter("C:\Temp\file" + files++.ToString()
+ ".txt");
        // messag = 0;
        //}
    } else
if (msgType == Defs.msgType_GenericBook)
    GenericBookHandler(bufReceive);
else
    NonBookHandler(bufReceive);
//sw.Stop();

int start = 0;
Order workingOrder;
int numBodyEntries = (int)buffer[14];
for (int j = 0; j < numBodyEntries; j++)
{
    workingOrder.symbolIndex = BigEndToInt16(buffer, start + 16);
    int innerMsgType = BigEndToInt16(buffer, start + 18);
    workingOrder.sequenceNumber = BigEndToInt32(buffer, start + 20);

    switch (innerMsgType)
    {
    case Defs.msgType_Add:
        long orderID = BigEndToInt64(buffer, start + 28);
        workingOrder.volume = BigEndToInt32(buffer, start + 36);
        int priceNumerator = BigEndToInt32(buffer, start + 40);
        int priceScaleCode = (int)buffer[start + 44];
        workingOrder.price = PriceScale(priceNumerator,
            priceScaleCode);
        workingOrder.side = ASCIIEncoding.ASCII.GetString(buffer,
            start + 45, 1);
        workingOrder.firmIndex = BigEndToInt16(buffer, start + 48);
        workingOrder.sessionID = (int)buffer[start + 50];
        start += 36;
        //pw.GWrite("parse", sp.Elapsed.ToString());
        //if (MakeKey(workingOrder.sessionID,
            workingOrder.symbolIndex) == 6.692)
            _orderBook[MakeKey(workingOrder.sessionID,
            workingOrder.symbolIndex)].AddEntryPrimary(orderID, workingOrder);
        break;
    case Defs.msgType_Modify:
        orderID = BigEndToInt64(buffer, start + 28);
        workingOrder.volume = BigEndToInt32(buffer, start + 36);
        priceNumerator = BigEndToInt32(buffer, start + 40);
        priceScaleCode = (int)buffer[start + 44];
        workingOrder.price = PriceScale(priceNumerator,
            priceScaleCode);
        workingOrder.side = ASCIIEncoding.ASCII.GetString(buffer,
            start + 45, 1);
        workingOrder.firmIndex = BigEndToInt16(buffer, start + 48);
        workingOrder.sessionID = (int)buffer[start + 50];
        start += 36;
        //pw.GWrite("parse", sp.Elapsed.ToString());
        //if (MakeKey(workingOrder.sessionID,
            workingOrder.symbolIndex) == 6.692)
            _orderBook[MakeKey(workingOrder.sessionID,
            workingOrder.symbolIndex)].ModifyEntryPrimary(orderID, workingOrder);
        break;
    case Defs.msgType_Delete:
        orderID = BigEndToInt64(buffer, start + 28);
workingOrder.side = ASCIIEncoding.ASCII.GetString(buffer, start + 36, 1);
workingOrder.sessionID = (int)buffer[start + 39];
workingOrder.firmIndex = BigEndToInt16(buffer, start + 40);
start += 28;
if (MakeKey(workingOrder.sessionID, workingOrder.symbolIndex) == 6.692)
    //
    //pw.GWrite("parse", sp.Elapsed.ToString());
    if (!_isInitialized == true)
        _orderBook[MakeKey(workingOrder.sessionID, workingOrder.symbolIndex)].DeleteEntryPrimary(orderID, workingOrder.sequenceNumber);
    //
    break;

    case Defs.msgType_Imbalance:
        //volume = BigEndToInt32(buffer, start + 28);
        //int totalImbalance = BigEndToInt32(buffer, start + 32);
        //int marketImbalance = BigEndToInt32(buffer, start + 36);
        //priceNumerator = BigEndToInt32(buffer, start + 40);
        //priceScaleCode = (int)buffer[44];
        //string auctionType = ASCIIEncoding.ASCII.GetString(buffer, start + 45, 1);
        //exchangeID = ASCIIEncoding.ASCII.GetString(buffer, start + 46, 1);
        //securityType = ASCIIEncoding.ASCII.GetString(buffer, start + 47, 1);
        workingOrder.sessionID = (int)buffer[start + 48];
        //int auctionTime = BigEndToInt16(buffer, start + 50);
        start += 36;
        //if (MakeKey(workingOrder.sessionID, workingOrder.symbolIndex) == 6.692)
            _orderBook[MakeKey(workingOrder.sessionID, workingOrder.symbolIndex)].Imbalance(workingOrder.sequenceNumber);
        break;

    default:
        //you shouldn't get here
        break;
    
}

Storage

/// <summary>
/// Adds entry to both the order book and consolidated prices list
/// </summary>
/// <param name="id">Order ID</param>
/// <param name="tmp">Order data</param>
public void AddEntryPrimary(long id, Order tmp)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    int check = _sequencer.CheckSource(tmp.sequenceNumber);
    if (check > 0)
        _isCorrupted = true;
    if (check != -1)
if (tmp.side == "B")
{
    if (!_bids.ContainsKey(tmp.price))
    {
        _bids.Add(tmp.price, new ConsolidatedEntry());
        _bids[tmp.price].orderList = new SortedDictionary<long, int>();
    }
    if (!_bids[tmp.price].orderList.ContainsKey(id))
    {
        _bids[tmp.price].orderList.Add(id, tmp.volume);
        _bids[tmp.price].totalVolume = _bids[tmp.price].orderList.Sum(ent => ent.Value);
    }
}

} else if (tmp.side == "S")
{
    if (!_asks.ContainsKey(tmp.price))
    {
        _asks.Add(tmp.price, new ConsolidatedEntry());
        _asks[tmp.price].orderList = new SortedDictionary<long, int>();
    }
    if (!_asks[tmp.price].orderList.ContainsKey(id))
    {
        _asks[tmp.price].orderList.Add(id, tmp.volume);
        _asks[tmp.price].totalVolume = _asks[tmp.price].orderList.Sum(ent => ent.Value);
    }
}

    _primaryDataStore.Add(id, tmp);
    _totalAdds++;
}

sw.Stop();
//pw.GWrite("storeAdd", sw.Elapsed.ToString());

/// <summary>
/// Removes entry from both the order book and consolidated prices list
/// </summary>
/// <param name="id">Order ID</param>
/// <param name="sequenceNumber">Source Sequence Number</param>
public void DeleteEntryPrimary(long id, int sequenceNumber)
{
    Stopwatch sw = new Stopwatch();
    //sw.Start();
    int check = _sequencer.CheckSource(sequenceNumber);
    if (check > 0)
    {
        _isCorrupted = true;
    }
    if (check != -1)
    {
        if (_primaryDataStore.ContainsKey(id))
        {
            if (_primaryDataStore[id].side == "B")
            {
                _bids[_primaryDataStore[id].price].orderList.Remove(id);
                if (_bids[_primaryDataStore[id].price].orderList.Count == 0)
                    _bids.Remove(_primaryDataStore[id].price);
            }
            else
                _bids.Remove(_primaryDataStore[id].price);
if (_primaryDataStore.ContainsKey(id))
{
    if (_primaryDataStore[id].side == "B")
    {
        _bids[_primaryDataStore[id].price].orderList.Remove(id);
        if (_bids[_primaryDataStore[id].price].orderList.Count == 0)
            _bids.Remove(_primaryDataStore[id].price);
        else
            _bids[_primaryDataStore[id].price].totalVolume =
            _bids[_primaryDataStore[id].price].orderList.Sum(ent => ent.Value);
    }
    else if (_primaryDataStore[id].side == "S")
    {
        _asks[_primaryDataStore[id].price].orderList.Remove(id);
        if (_asks[_primaryDataStore[id].price].orderList.Count == 0)
            _asks.Remove(_primaryDataStore[id].price);
        else
            _asks[_primaryDataStore[id].price].totalVolume =
            _asks[_primaryDataStore[id].price].orderList.Sum(ent => ent.Value);
    }
    _primaryDataStore.Remove(id);
    _totalDeletes++;
}
else
    _isCorrupted = true;

//sw.Stop();
//pw.GWrite("storeDel", sw.Elapsed.ToString());

/// <summary>
/// Modifies entry in both the order book and consolidated prices list
/// </summary>
/// <param name="id">Order ID</param>
/// <param name="tmp">Order Data</param>
public void ModifyEntryPrimary(long id, Order tmp)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    int check = _sequencer.CheckSource(tmp.sequenceNumber);

    if (check > 0)
        _isCorrupted = true;
    if (check != -1)
    {
        if (_primaryDataStore.ContainsKey(id))
        {
            if (_primaryDataStore[id].side == "B")
            {
                _bids[_primaryDataStore[id].price].orderList.Remove(id);
                if (_bids[_primaryDataStore[id].price].orderList.Count == 0)
                    _bids.Remove(_primaryDataStore[id].price);
                else
                    _bids[_primaryDataStore[id].price].totalVolume =
                    _bids[_primaryDataStore[id].price].orderList.Sum(ent => ent.Value);
            }
            else if (_primaryDataStore[id].side == "S")
            {
                _asks[_primaryDataStore[id].price].orderList.Remove(id);
                if (_asks[_primaryDataStore[id].price].orderList.Count == 0)
                    _asks.Remove(_primaryDataStore[id].price);
                else
                    _asks[_primaryDataStore[id].price].totalVolume =
                    _asks[_primaryDataStore[id].price].orderList.Sum(ent => ent.Value);
            }
            if (tmp.side == "B")
            {
            }
if (!_bids.ContainsKey(tmp.price))
{
    _bids.Add(tmp.price, new ConsolidatedEntry());
    _bids[tmp.price].orderList = new SortedDictionary<long, int>();
}
_bids[tmp.price].orderList.Add(id, tmp.volume);
_bids[tmp.price].totalVolume =
_bids[tmp.price].orderList.Sum(ent => ent.Value);
else if (tmp.side == "S")
{
    if (!_asks.ContainsKey(tmp.price))
    {
        _asks.Add(tmp.price, new ConsolidatedEntry());
        _asks[tmp.price].orderList = new SortedDictionary<long, int>();
    }
    _asks[tmp.price].orderList.Add(id, tmp.volume);
    _asks[tmp.price].totalVolume =
    _asks[tmp.price].orderList.Sum(ent => ent.Value);
    _primaryDataStore.Remove(id);
    _primaryDataStore.Add(id, tmp);
}
else
_isCorrupted = true;
}
//sw.Stop();
//pw.GWrite("storeMod", sw.Elapsed.ToString());

Retrieval
/// <summary>
/// Retrives the bid side of the order book for the specified symbol
/// </summary>
/// <param name="symbol">The symbol being requested</param>
/// <returns>Bid side of the orderbook sorted by price</returns>
public SortedDictionary<double, ConsolidatedEntry> GetBidPrices(string symbol)
{
    lock (_orderBook)
    {
        Stopwatch sw = new Stopwatch();
        double value;
        _symbolMap.TryGetValue(symbol, out value);
        SortedDictionary<double, ConsolidatedEntry> tmp =
        _orderBook[value].ConsolidatedBidPrices;
        //sw.Stop();
        //pw.GWrite("read", sw.Elapsed.ToString());
        //if (tmp.Count > 5)
        //    _guiLatency = sw.ElapsedMilliseconds;
        return tmp;
    }
}
Appendix B – CUDA Codes

Parsing

```csharp
public void Updater()
{
    byte[] bufReceive = new byte[2048];
    //Stopwatch sw = new Stopwatch();

    string test;
    GASS.CUDA.Types.Int1 inx;
    inx.x = 1;
    while (true)
    {
        string file = "C://Temp/file" + inx.x.ToString() + ".txt";
        StreamReader sr = new StreamReader(file);
        inx.x++;
        if (inx.x > 500)
            break;
        while ((test = sr.ReadLine()) != null)
        {
            if (test == "")
                continue;
            sp = new Stopwatch();
            sp.Reset();
            sp.Start();

            System.Text.UTF8Encoding encoding = new
System.Text.UTF8Encoding(encoding);
            bufReceive = encoding.GetBytes(test);
            if (bufReceive.Length < 4)
                continue;

            GASS.CUDA.Types.Int1 msgType;
            msgType.x = BigEndToInt16(bufReceive, 2);

            if (msgType.x == Defs.msgType_GenericBook)
                GenericBookHandler(bufReceive);
            else
                NonBookHandler(bufReceive);
        }
        sr.Close();
    }
    getter.Abort();
    return;
}

private void GenericBookHandler(byte[] buffer)
{
    if (buffer.Length < 40)
        return;
    GASS.CUDA.Types.Int1 start;
    start.x = 0;
    Order currentOrder;
    GASS.CUDA.Types.Int1 numBodyEntries;
    numBodyEntries.x = (int)buffer[14];
    GASS.CUDA.Types.Int1 j;
    for (j.x = 0; j.x < numBodyEntries.x; j.x++)
    {
        if (start.x + 50 > buffer.Length)
            return;
```
GASS.CUDA.Types.Int1 innerMsgType;
currentOrder.symbolIndex = BigEndianToInt16(buffer, start.x + 16);
innerMsgType.x = BigEndianToInt16(buffer, start.x + 18);
currentOrder.sequenceNumber = BigEndianToInt32(buffer, start.x + 20);

switch (innerMsgType.x)
{
case Defs.msgType_Add:
    GASS.CUDA.Types.Long1 orderID;
    orderID.x = BigEndianToInt64(buffer, start.x + 28);
    currentOrder.volume = BigEndianToInt32(buffer, start.x + 36);
    GASS.CUDA.Types.Int1 priceNumerator;
    priceNumerator.x = BigEndianToInt32(buffer, start.x + 40);
    GASS.CUDA.Types.Int1 priceScaleCode;
    priceScaleCode.x = (int)buffer[start.x + 44];
    currentOrder.price = PriceScale(priceNumerator.x, priceScaleCode.x);
    currentOrder.side = ASCIIEncoding.ASCII.GetString(buffer, start.x + 45, 1);
    currentOrder.firmIndex = BigEndianToInt16(buffer, start.x + 48);
    currentOrder.sessionID = (int)buffer[start.x + 50];
    start.x += 36;
    parseWrite.GWrite("parse", sp.Elapsed.ToString());
    if (!_orderBook.ContainsKey(MakeKey(currentOrder.sessionID, currentOrder.symbolIndex)))
        _orderBook.Add(MakeKey(currentOrder.sessionID, currentOrder.symbolIndex), new OrderBook());
    _orderBook[MakeKey(currentOrder.sessionID, currentOrder.symbolIndex)].AddEntryPrimary(orderID.x, currentOrder);
    break;

case Defs.msgType_Modify:
    orderID.x = BigEndianToInt64(buffer, start.x + 28);
    currentOrder.volume = BigEndianToInt32(buffer, start.x + 36);
    priceNumerator.x = BigEndianToInt32(buffer, start.x + 40);
    priceScaleCode.x = (int)buffer[start.x + 44];
    currentOrder.price = PriceScale(priceNumerator.x, priceScaleCode.x);
    currentOrder.side = ASCIIEncoding.ASCII.GetString(buffer, start.x + 45, 1);
    currentOrder.firmIndex = BigEndianToInt16(buffer, start.x + 48);
    currentOrder.sessionID = (int)buffer[start.x + 50];
    start.x += 36;
    parseWrite.GWrite("parse", sp.Elapsed.ToString());
    if (MakeKey(currentOrder.sessionID, currentOrder.symbolIndex) == 6.692)
        return;
    if (!_orderBook.ContainsKey(MakeKey(currentOrder.sessionID, currentOrder.symbolIndex)))
        return;
    _orderBook[MakeKey(currentOrder.sessionID, currentOrder.symbolIndex)].ModifyEntryPrimary(orderID.x, currentOrder);
    break;

case Defs.msgType_Delete:
    orderID.x = BigEndianToInt64(buffer, start.x + 28);
    currentOrder.side = ASCIIEncoding.ASCII.GetString(buffer, start.x + 36, 1);
    currentOrder.firmIndex = BigEndianToInt16(buffer, start.x + 40);
    currentOrder.sessionID = (int)buffer[start.x + 39];
    start.x += 28;
// if (MakeKey(currentOrder.sessionID, currentOrder.symbolIndex) == 6.692)
// {
//   parseWrite.GWrite("parse", sp.Elapsed.ToString());
//   if (!_orderBook.ContainsKey(MakeKey(currentOrder.sessionID, currentOrder.symbolIndex)))
//     return;
//   _orderBook[MakeKey(currentOrder.sessionID, currentOrder.symbolIndex)].DeleteEntryPrimary(orderID.x, currentOrder.sequenceNumber);
//   //}
//   break;
// }
//
// default:
//   // you shouldn't get here
//   break;
// }
}

Storage

/// <summary>
/// Adds entry to both the order book and consolidated prices list
/// </summary>
/// <param name="id">Order ID</param>
/// <param name="tmp">Order data</param>
public void AddEntryPrimary(long id, Order tmp)
{
  Stopwatch sw = new Stopwatch();
  sw.Start();

  if (tmp.side == "B")
  {
    if (!_bids.ContainsKey(tmp.price))
    {
      _bids.Add(tmp.price, new ConsolidatedEntry());
      _bids[tmp.price].orderList = new SortedDictionary<long, int>();
    }
    if (!_bids[tmp.price].orderList.ContainsKey(id))
    {
      _bids[tmp.price].orderList.Add(id, tmp.volume);
      _bids[tmp.price].totalVolume = _bids[tmp.price].orderList.Sum(ent
      => ent.Value);
    }
  } else if (tmp.side == "S")
  {
    if (!_asks.ContainsKey(tmp.price))
    {
      _asks.Add(tmp.price, new ConsolidatedEntry());
      _asks[tmp.price].orderList = new SortedDictionary<long, int>();
    }
    if (!_asks[tmp.price].orderList.ContainsKey(id))
    {
      _asks[tmp.price].orderList.Add(id, tmp.volume);
      _asks[tmp.price].totalVolume = _asks[tmp.price].orderList.Sum(ent
      => ent.Value);
    }
  }

  if (!_primaryDataStore.ContainsKey(id))
    _primaryDataStore.Add(id, tmp);
/// <summary>
/// Removes entry from both the order book and consolidated prices list
/// </summary>
/// <param name="id">Order ID</param>
/// <param name="sequenceNumber">Source Sequence Number</param>
public void DeleteEntryPrimary(long id, int sequenceNumber)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    if (_primaryDataStore.ContainsKey(id))
    {
        if (_primaryDataStore[id].side == "B")
        {
            _bids[_primaryDataStore[id].price].orderList.Remove(id);
            if (_bids[_primaryDataStore[id].price].orderList.Count == 0)
            {
                _bids.Remove(_primaryDataStore[id].price);
            }
            else
            {
                _bids[_primaryDataStore[id].price].totalVolume =
                _bids[_primaryDataStore[id].price].orderList.Sum(ent => ent.Value);
            }
        }
        else if (_primaryDataStore[id].side == "S")
        {
            _asks[_primaryDataStore[id].price].orderList.Remove(id);
            if (_asks[_primaryDataStore[id].price].orderList.Count == 0)
            {
                _asks.Remove(_primaryDataStore[id].price);
            }
            else
            {
                _asks[_primaryDataStore[id].price].totalVolume =
                _asks[_primaryDataStore[id].price].orderList.Sum(ent => ent.Value);
            }
        }
    }
    _primaryDataStore.Remove(id);
    _totalDeletes++;
    }
    else
    {
        _isCorrupted = true;
    }
    sw.Stop();
    storeWrite.GWrite("storeDel", sw.Elapsed.ToString());
}

/// <summary>
/// Modifies entry in both the order book and consolidated prices list
/// </summary>
/// <param name="id">Order ID</param>
/// <param name="tmp">Order Data</param>
public void ModifyEntryPrimary(long id, Order tmp)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    if (_primaryDataStore.ContainsKey(id))
    {
        if (_primaryDataStore[id].side == "B")
        {
            _bids[_primaryDataStore[id].price].orderList.Remove(id);
            if (_bids[_primaryDataStore[id].price].orderList.Count == 0)
            {
                _bids.Remove(_primaryDataStore[id].price);
            }
        }
    }
}
else if (_primaryDataStore[id].side == "S")
{
    _asks[_primaryDataStore[id].price].orderList.Remove(id);
    if (_asks[_primaryDataStore[id].price].orderList.Count == 0)
        _asks.Remove(_primaryDataStore[id].price);
    else
        _asks[_primaryDataStore[id].price].totalVolume =
        _asks[_primaryDataStore[id].price].orderList.Sum(ent => ent.Value);
}

if (tmp.side == "B")
{
    if (!_bids.ContainsKey(tmp.price))
    {
        _bids.Add(tmp.price, new ConsolidatedEntry());
        _bids[tmp.price].orderList = new SortedDictionary<long, int>();
    }
    if (!_bids[tmp.price].orderList.ContainsKey(id))
        _bids[tmp.price].orderList.Add(id, tmp.volume);
    _bids[tmp.price].totalVolume =
    _bids[tmp.price].orderList.Sum(ent => ent.Value);
}
else if (tmp.side == "S")
{
    if (!_asks.ContainsKey(tmp.price))
    {
        _asks.Add(tmp.price, new ConsolidatedEntry());
        _asks[tmp.price].orderList = new SortedDictionary<long, int>();
    }
    if (!_asks[tmp.price].orderList.ContainsKey(id))
        _asks[tmp.price].orderList.Add(id, tmp.volume);
    _asks[tmp.price].totalVolume =
    _asks[tmp.price].orderList.Sum(ent => ent.Value);
}

    _primaryDataStore.Remove(id);
    _primaryDataStore.Add(id, tmp);
}
else
    _isCorrupted = true;

sw.Stop();
storeWrite.GWrite("storeMod", sw.Elapsed.ToString());

Retrieval

/// <summary>
/// Retrieves the bid side of the order book for the specified symbol
/// </summary>
/// <param name="symbol">The symbol being requested</param>
/// <returns>Bid side of the orderbook sorted by price</returns>
public void GetBidPrices()
{
    while (true)
    {


if (_orderBook.Count < 6)
    continue;
lock (_orderBook)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    double value;
    //tmp dictionary to facilitate stopwatch;
    SortedDictionary<double, ConsolidatedEntry> tmp =
        _orderBook.ElementAt(5).Value.ConsolodatedBidPrices;
    sw.Stop();
    parseWrite.GWrite("read", sw.Elapsed.ToString());
    //if (tmp.Count > 5)
    //  _guiLatency = sw.ElapsedMilliseconds;

}
Thread.Sleep(5);
}
Appendix C – VHDL Codes

Parsing Modules

Load Counter

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.all;

ENTITY LoadCounter IS
 PORT
 { clock : IN STD_LOGIC ;
   cnt_en : IN STD_LOGIC ;
   q : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
 };
END LoadCounter;

ARCHITECTURE SYN OF loadcounter IS

SIGNAL sub_wire0 : STD_LOGIC_VECTOR (3 DOWNTO 0);

COMPONENT lpm_counter
 GENERIC {
   lpm_direction : STRING;
   lpm_port_updown : STRING;
   lpm_type : STRING;
   lpm_width : NATURAL
 }
 PORT {
   clock : IN STD_LOGIC ;
   q : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
   cnt_en : IN STD_LOGIC
 }
END COMPONENT;

BEGIN
 q <= sub_wire0(3 DOWNTO 0);

 lpm_counter_component : lpm_counter
 GENERIC MAP {
   lpm_direction => "UP",
   lpm_port_updown => "PORT_UNUSED",
   lpm_type => "LPM_COUNTER",
   lpm_width => 4
 }
 PORT MAP {
   clock => clock,
   cnt_en => cnt_en,
   q => sub_wire0
 ”};
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.all;

ENTITY Ram IS
PORT
(
  clock  : IN STD_LOGIC ;
  data  : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
  rdaddress  : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
  wraddress  : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
  wren  : IN STD_LOGIC  := '1';
  q  : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
);
END Ram;

ARCHITECTURE SYN OF ram IS
SIGNAL sub_wire0 : STD_LOGIC_VECTOR (31 DOWNTO 0);

COMPONENT altsyncram
GENERIC (
  address_aclr_b  : STRING;
  address_reg_b  : STRING;
  clock_enable_input_a  : STRING;
  clock_enable_input_b  : STRING;
  clock_enable_output_b  : STRING;
  intended_device_family  : STRING;
  lpm_type  : STRING;
  numwords_a  : NATURAL;
  numwords_b  : NATURAL;
  operation_mode  : STRING;
  outdata_aclr_b  : STRING;
  outdata_reg_b  : STRING;
  power_up_uninitialized  : STRING;
  read_during_write_mode_mixed_ports  : STRING;
  widthad_a  : NATURAL;
  widthad_b  : NATURAL;
  width_a  : NATURAL;
  width_b  : NATURAL;
  width_byteena_a  : NATURAL
);
PORT (
  wren_a : IN STD_LOGIC ;
  clock0 : IN STD_LOGIC ;
  address_a  : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
  address_b  : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
  q_b  : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
  data_a : IN STD_LOGIC_VECTOR (31 DOWNTO 0)
);
END COMPONENT;

BEGIN
Hernandez, Rapid Decoding of Digital Data Streams Using Field Programmable Gate Arrays | 60
q <= sub_wire0(31 DOWNTO 0);

altsyncram_component : altsyncram
GENERIC MAP {
  address_aclr_b => "NONE",
  address_reg_b => "CLOCK0",
  clock_enable_input_a => "BYPASS",
  clock_enable_input_b => "BYPASS",
  clock_enable_output_b => "BYPASS",
  intended_device_family => "Stratix IV",
  lpm_type => "altsyncram",
  numwords_a => 16,
  numwords_b => 16,
  operation_mode => "DUAL_PORT",
  outdata_aclr_b => "NONE",
  outdata_reg_b => "CLOCK0",
  power_up_uninitialized => "FALSE",
  read_during_write_mode_mixed_ports => "DONT_CARE",
  widthad_a => 4,
  widthad_b => 4,
  width_a => 32,
  width_b => 32,
  width_byteena_a => 1
}
PORT MAP {
  wren_a => wren,
  clock0 => clock,
  address_a => wraddress,
  address_b => rdaddress,
  data_a => data,
  q_b => sub_wire0
};

END SYN;

**ByteOrder**

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity byteOrder is
  port (  data :  in std_logic_vector(31 downto 0);
          q    :  out std_logic_vector(31 downto 0)
);
end byteOrder;

architecture behv of byteOrder is
begin

  q(0) <= data(31);
  q(1) <= data(30);
  q(2) <= data(29);
  q(3) <= data(28);
  q(4) <= data(27);
  q(5) <= data(26);
  q(6) <= data(25);
  q(7) <= data(24);

Hernandez, Rapid Decoding of Digital Data Streams Using Field Programmable Gate Arrays | 61
Parser
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Parser is
port ( data : in std_logic_vector(31 downto 0);
        q : out std_logic_vector(31 downto 0);
        add : out std_logic;
        modify : out std_logic;
        delete : out std_logic
        );
end Parser;

architecture behv of Parser is
begin
  
  end behv;

q(8) <= data(23);
q(9) <= data(22);
q(10) <= data(21);
q(11) <= data(20);
q(12) <= data(19);
q(13) <= data(18);
q(14) <= data(17);
q(15) <= data(16);
q(16) <= data(15);
q(17) <= data(14);
q(18) <= data(13);
q(19) <= data(12);
q(20) <= data(11);
q(21) <= data(10);
q(22) <= data(9);
q(23) <= data(8);
q(24) <= data(7);
q(25) <= data(6);
q(26) <= data(5);
q(27) <= data(4);
q(28) <= data(3);
q(29) <= data(2);
q(30) <= data(1);
q(31) <= data(0);
elsif data(15 downto 0) = "1100110" then
    add <= '0';
    modify <= '0';
    delete <= '1';
else
    add <= '0';
    modify <= '0';
    delete <= '0';
end if;
end process;
end behv;

Storage Modules

Add
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Add is
    port ( act : in std_logic;
           data : in std_logic_vector(31 downto 0);
           en : out std_logic;
           q : out std_logic_vector(15 downto 0)
    );
end Add;

architecture behv of Add is
begin
    process (act)
    begin
        if act = '1' then
            en <= '1';

            q(0) <= data(16);
            q(1) <= data(17);
            q(2) <= data(18);
            q(3) <= data(19);
            q(4) <= data(20);
            q(5) <= data(21);
            q(6) <= data(22);
            q(7) <= data(23);
            q(8) <= data(24);
            q(9) <= data(25);
            q(10) <= data(26);
            q(11) <= data(27);
            q(12) <= data(28);
            q(13) <= data(29);
            q(14) <= data(30);
            q(15) <= data(31);
        else
            en <= '0';
        end if;
    end process;
end behv;
end process;
end behv;

**Delete**

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Delete is
port (  act  :  in std_logic;
        data :  in std_logic_vector(31 downto 0);
        en  : out std_logic;
        q    :  out std_logic_vector(15 downto 0)
);
end Delete;

architecture behv of Delete is
begin
  process (act)
  begin
    if act = '1' then
      en <= '1';
      q(0) <= data(16);
      q(1) <= data(17);
      q(2) <= data(18);
      q(3) <= data(19);
      q(4) <= data(20);
      q(5) <= data(21);
      q(6) <= data(22);
      q(7) <= data(23);
      q(8) <= data(24);
      q(9) <= data(25);
      q(10) <= data(26);
      q(11) <= data(27);
      q(12) <= data(28);
      q(13) <= data(29);
      q(14) <= data(30);
      q(15) <= data(31);
    else
      en <= '0';
    end if;
  end process;
end behv;

**Modify**

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Modify is
port (  act  :  in std_logic;
        data :  in std_logic_vector(31 downto 0);
        en  : out std_logic;
        q    :  out std_logic_vector(15 downto 0)
);
end Modify;
end Modify;

architecture behv of Modify is
begin

process (act)
begin
if act = '1' then
  en <= '1';

  q(0) <= data(16);
  q(1) <= data(17);
  q(2) <= data(18);
  q(3) <= data(19);
  q(4) <= data(20);
  q(5) <= data(21);
  q(6) <= data(22);
  q(7) <= data(23);
  q(8) <= data(24);
  q(9) <= data(25);
  q(10) <= data(26);
  q(11) <= data(27);
  q(12) <= data(28);
  q(13) <= data(29);
  q(14) <= data(30);
  q(15) <= data(31);
else
  en <= '0';
end if;
end process;
end behv;

ASMMux
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY ASMMux IS
PORT
  (data0x  : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
data1x  : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
data2x  : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
  sel  : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
result  : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
);
END ASMMux;

ARCHITECTURE SYN OF asmmux IS

-- type STD_LOGIC_2D is array (NATURAL RANGE <>, NATURAL RANGE <=) of STD_LOGIC;

SIGNAL sub_wire0 : STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL sub_wire1 : STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL sub_wire2 : STD_LOGIC_2D (2 DOWNTO 0, 15 DOWNTO 0);
BEGIN

sub_wire4 <= data0x(15 DOWNTO 0);
sub_wire3 <= data1x(15 DOWNTO 0);
result <= sub_wire0(15 DOWNTO 0);
sub_wire1 <= data2x(15 DOWNTO 0);
sub_wire2(2, 0) <= sub_wire1(0);
sub_wire2(2, 1) <= sub_wire1(1);
sub_wire2(2, 2) <= sub_wire1(2);
sub_wire2(2, 3) <= sub_wire1(3);
sub_wire2(2, 4) <= sub_wire1(4);
sub_wire2(2, 5) <= sub_wire1(5);
sub_wire2(2, 6) <= sub_wire1(6);
sub_wire2(2, 7) <= sub_wire1(7);
sub_wire2(2, 8) <= sub_wire1(8);
sub_wire2(2, 9) <= sub_wire1(9);
sub_wire2(2, 10) <= sub_wire1(10);
sub_wire2(2, 11) <= sub_wire1(11);
sub_wire2(2, 12) <= sub_wire1(12);
sub_wire2(2, 13) <= sub_wire1(13);
sub_wire2(2, 14) <= sub_wire1(14);
sub_wire2(2, 15) <= sub_wire1(15);
sub_wire2(1, 0) <= sub_wire3(0);
sub_wire2(1, 1) <= sub_wire3(1);
sub_wire2(1, 2) <= sub_wire3(2);
sub_wire2(1, 3) <= sub_wire3(3);
sub_wire2(1, 4) <= sub_wire3(4);
sub_wire2(1, 5) <= sub_wire3(5);
sub_wire2(1, 6) <= sub_wire3(6);
sub_wire2(1, 7) <= sub_wire3(7);
sub_wire2(1, 8) <= sub_wire3(8);
sub_wire2(1, 9) <= sub_wire3(9);
sub_wire2(1, 10) <= sub_wire3(10);
sub_wire2(1, 11) <= sub_wire3(11);
sub_wire2(1, 12) <= sub_wire3(12);
sub_wire2(1, 13) <= sub_wire3(13);
sub_wire2(1, 14) <= sub_wire3(14);
sub_wire2(1, 15) <= sub_wire3(15);
sub_wire2(0, 0) <= sub_wire4(0);
sub_wire2(0, 1) <= sub_wire4(1);
sub_wire2(0, 2) <= sub_wire4(2);
sub_wire2(0, 3) <= sub_wire4(3);
sub_wire2(0, 4) <= sub_wire4(4);
sub_wire2(0, 5) <= sub_wire4(5);
sub_wire2(0, 6) <= sub_wire4(6);
sub_wire2(0, 7) <= sub_wire4(7);
sub_wire2(0, 8) <= sub_wire4(8);
sub_wire2(0, 9) <= sub_wire4(9);
sub_wire2(0, 10) <= sub_wire4(10);
sub_wire2(0, 11) <= sub_wire4(11);
sub_wire2(0, 12) <= sub_wire4(12);
sub_wire2(0, 13) <= sub_wire4(13);
sub_wire2(0, 14) <= sub_wire4(14);
sub_wire2(0, 15) <= sub_wire4(15);

lpm_mux_component : lpm_mux
GENERIC MAP (  
lpm_size => 3,
lpm_type => "LPM_MUX",
lpm_width => 16,
END SYN;

Retrieval Modules

RetrieveCounter
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.all;

ENTITY RetrieveCounter IS
  PORT
   ( clock  : IN STD_LOGIC ;
     cnt_en  : IN STD_LOGIC ;
     q  : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
   );
END RetrieveCounter;

ARCHITECTURE SYN OF retrievecounter IS
  SIGNAL sub_wire0 : STD_LOGIC_VECTOR (15 DOWNTO 0);
  COMPONENT lpm_counter
    GENERIC
    ( lpm_direction  : STRING;
    lpm_port_updown  : STRING;
    lpm_type  : STRING;
    lpm_width  : NATURAL
    );
    PORT
    ( clock : IN STD_LOGIC ;
    q : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    cnt_en : IN STD_LOGIC
    );
    END COMPONENT;
  BEGIN
    q <= sub_wire0(15 DOWNTO 0);
    lpm_counter_component : lpm_counter
    GENERIC MAP
    ( lpm_direction => "UP",
      lpm_port_updown => "PORT_UNUSED",
      lpm_type => "LPM_COUNTER",
      lpm_width => 16
    )
    PORT MAP
    ( clock => clock,

  END ARCHITECTURE SYN;
cnt_en => cnt_en,
q => sub_wire0
);

END SYN;

**ParsedRam**

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.all;

ENTITY ParsedRam IS
PORT
(
clock : IN STD_LOGIC;
data : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
rdaddress : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
waddress : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
wren : IN STD_LOGIC := '1';
q : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
);
END ParsedRam;

ARCHITECTURE SYN OF parsedram IS

SIGNAL sub_wire0 : STD_LOGIC_VECTOR (63 DOWNTO 0);

COMPONENT altsyncram
GENERIC
(
address_aclr_b : STRING;
address_reg_b : STRING;
clock_enable_input_a : STRING;
clock_enable_input_b : STRING;
clock_enable_output_b : STRING;
intended_device_family : STRING;
lpm_type : STRING;
umwords_a : NATURAL;
umwords_b : NATURAL;
operation_mode : STRING;
outdata_aclr_b : STRING;
outdata_reg_b : STRING;
power_up_uninitialized : STRING;
read_during_write_modeMixed_ports : STRING;
widthad_a : NATURAL;
widthad_b : NATURAL;
width_a : NATURAL;
width_b : NATURAL;
width_byteena_a : NATURAL
);
PORT
(
wren_a : IN STD_LOGIC;
clock0 : IN STD_LOGIC;
address_a : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
address_b : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
q_b : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
);
data_a : IN STD_LOGIC_VECTOR (63 DOWNTO 0)

END COMPONENT;

BEGIN

q <= sub_wire0(63 DOWNTO 0);

altsyncram_component : altsyncram
GENERIC MAP (
    address_aclr_b => "NONE",
    address_reg_b => "CLOCK0",
    clock_enable_input_a => "BYPASS",
    clock_enable_input_b => "BYPASS",
    clock_enable_output_b => "BYPASS",
    intended_device_family => "Stratix IV",
    lpm_type => "altsyncram",
    numwords_a => 65536,
    numwords_b => 65536,
    operation_mode => "DUAL_PORT",
    outdata_aclr_b => "NONE",
    outdata_reg_b => "CLOCK0",
    power_up_uninitialized => "FALSE",
    read_during_write_mode_mixed_ports => "DONT_CARE",
    width_a => 16,
    width_b => 16,
    width_a => 64,
    width_b => 64,
    width_byteena_a => 1
)
PORT MAP (
    wren_a => wren,
    clock0 => clock,
    address_a => wraddress,
    address_b => rdaddress,
    data_a => data,
    q_b => sub_wire0
);

END SYN;
## Appendix D – Arcabook Messages

### Message Header Format

All messages are preceded by a standard header format with the exception of the Order Book Refresh Message. The table on the next page describes the header fields of an NYSE ARCA Quote message.

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Size (Bytes)</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MsgSize</td>
<td>0</td>
<td>2</td>
<td>Binary Integer</td>
<td>This field indicates the minimum size of the message body in bytes. Total size can vary with the number of bodies in the message:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sequence Number Reset – ‘18 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Heartbeat Message – ‘14 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Heartbeat Response Message – ‘34 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message Unavailable – ‘22 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Retransmission Request Message – ‘42 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Retransmission Response Message – ‘42 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Book Refresh Request Message – ‘38 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Imbalance Refresh Request Message – ‘38 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Book Refresh Message – ‘46 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Imbalance Refresh Message – ‘50 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Symbol Index Mapping Request Message – ‘38 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Symbol Index Mapping Message – ‘34 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Firm Index Mapping Request Message – ‘38 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Firm Index Mapping Message – ‘26 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Symbol Clear – ‘22 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Add Order Message – ‘46 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Modify Order Message – ‘46 Bytes’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Delete Order Message – ‘38 Bytes’</td>
</tr>
<tr>
<td>MsgType</td>
<td>2</td>
<td>2</td>
<td>Binary Integer</td>
<td>This field identifies the type of message ‘1’ – Sequence Number Reset</td>
</tr>
</tbody>
</table>

Hernandez, Rapid Decoding of Digital Data Streams Using Field Programmable Gate Arrays | 70
<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MsgSeqNum</td>
<td>4</td>
<td>Uint4</td>
<td>This field contains the message sequence number assigned by PDP for each product. It is used for gap detection. Also known as Line Sequence Number (LSN).</td>
</tr>
<tr>
<td>SendTime</td>
<td>8</td>
<td>Uint4</td>
<td>This field specifies the time message was created by PDP. The number represents the number of milliseconds since midnight of the same day.</td>
</tr>
<tr>
<td>ProductId</td>
<td>12</td>
<td>Uint1</td>
<td>‘115’ is the product value used in the PDP header to identify the ArcaBook feed</td>
</tr>
<tr>
<td>RetransFlag</td>
<td>13</td>
<td>Uint1</td>
<td>A flag that indicates whether this is an original, retransmitted, or ‘replayed’ message. Valid values include: ‘1’ – Original message ‘2’ – Retransmitted message ‘5’ – Refresh Retransmission ‘6’ – Failover Retransmission ‘7’ – Start of Update ‘8’ – End of Update</td>
</tr>
</tbody>
</table>
‘9’ – Only one packet in update

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Size (Bytes)</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumBodyEntries</td>
<td>14</td>
<td>1</td>
<td>Binary Integer</td>
<td>The number of times the message body repeats in the message. For example, if the body consists of a field (named Volume) and the “NumBodyEntries” field is 2, the number of bytes in the message body will be 8</td>
</tr>
<tr>
<td>FILLER</td>
<td>15</td>
<td>1</td>
<td>ASCII String</td>
<td>This is a filler, reserved for future use</td>
</tr>
</tbody>
</table>

**Add Message Body Format**

The table below describes the body fields of an Arcabook Add message (MsgType = ‘100’). Arcabook sends this message for a new open order.

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Size (Bytes)</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SymbolIndex</td>
<td>16</td>
<td>2</td>
<td>Binary Integer</td>
<td>This field identifies the numerical representation of the symbol. User can combine this value with the session id to obtain a unique key</td>
</tr>
<tr>
<td>MsgType</td>
<td>18</td>
<td>2</td>
<td>Binary Integer</td>
<td>This field identifies the type of message Message ‘100’ – Add Order Message</td>
</tr>
<tr>
<td>SourceSeqNum</td>
<td>20</td>
<td>4</td>
<td>Binary Integer</td>
<td>This field contains the sequence number assigned by the source system to this message. The sequence number is unique only to a given stock. Hence orders for two different stocks may share the same source sequence number. Please note that the sequence number while it</td>
</tr>
</tbody>
</table>
increases serially, it does not increase monotonically.

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Binary Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceTime</td>
<td>24</td>
<td>Binary</td>
<td>This field specifies the quote generation time. The number in this field represents the number of milliseconds since midnight of the same day.</td>
</tr>
<tr>
<td>OrderId</td>
<td>28</td>
<td>Binary</td>
<td>The Order ID identifies a unique order and will allow customers of NYSE Arca Trades to correlate trades to the order</td>
</tr>
<tr>
<td>Volume</td>
<td>32</td>
<td>Binary</td>
<td>This field contains the size of the order. Please note we do not send Odd Lot (&lt;100) quotes.</td>
</tr>
<tr>
<td>PriceNumerator</td>
<td>36</td>
<td>Binary</td>
<td>This field specifies the price of the order.</td>
</tr>
<tr>
<td>PriceScaleCode</td>
<td>40</td>
<td>Binary</td>
<td>Decimal Placement</td>
</tr>
<tr>
<td>Side</td>
<td>41</td>
<td>ASCII</td>
<td>This field indicates the side of the order Buy/sell. Valid Values: ‘B’ – Buy ‘S’ – Sell</td>
</tr>
<tr>
<td>SecurityType</td>
<td>43</td>
<td>ASCII</td>
<td>This field specifies the security type for this message. Valid values: ‘E’ – Equity ‘B’ – BB</td>
</tr>
<tr>
<td>FirmIndex</td>
<td>44</td>
<td>Binary</td>
<td>This field identifies the numerical representation of the firm sending the quote if attributed.</td>
</tr>
</tbody>
</table>
Modify Message Body Format

The table below describes the body fields of an ArcaBook Modify message (MsgType = ‘101’). ArcaBook sends this message when an order in an ArcaBook is modified. The order id refers to the original order sent in the add order message. The following events trigger a modify order message.

- The price of an order changes
- The size of an order changes
- An order is partially filled
- An order is routed to an away market with some shares remaining in the ArcaBook.

**Note:** If an away market declines the NYSE Arca preference, a Modify Order message is sent to “add” the declined shares back to the Archipelago book.

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Size (Bytes)</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SymbolIndex</td>
<td>16</td>
<td>2</td>
<td>Binary Integer</td>
<td>This field identifies the numerical representation of the symbol. User can combine this value with the session id to obtain a unique key</td>
</tr>
<tr>
<td>MsgType</td>
<td>18</td>
<td>2</td>
<td>Binary Integer</td>
<td>This field identifies the type of message Message ‘101’ – Modify Order Message</td>
</tr>
<tr>
<td>SourceSeqNum</td>
<td>20</td>
<td>4</td>
<td>Binary Integer</td>
<td>This field contains the sequence number assigned by the source system to this message. The sequence number is unique only to a given stock. Hence orders for two different stocks may share the same source sequence number. Please note that the sequence number while it</td>
</tr>
<tr>
<td>Field</td>
<td>Length</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>--------</td>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>SourceTime</td>
<td>24</td>
<td>4</td>
<td>Binary Integer, This field specifies the quote generation time. The number in this field represents the number of milliseconds since midnight of the same day.</td>
<td></td>
</tr>
<tr>
<td>OrderId</td>
<td>28</td>
<td>4</td>
<td>Binary Integer, The Order ID identifies a unique order and will allow customers of NYSE Arca Trades to correlate trades to the order.</td>
<td></td>
</tr>
<tr>
<td>Volume</td>
<td>32</td>
<td>4</td>
<td>Binary Integer, This field contains the size of the order. Please note we do not send Odd Lot (&lt;100) quotes.</td>
<td></td>
</tr>
<tr>
<td>PriceNumerator</td>
<td>36</td>
<td>4</td>
<td>Binary Integer, This field specifies the price of the order.</td>
<td></td>
</tr>
<tr>
<td>PriceScaleCode</td>
<td>40</td>
<td>1</td>
<td>Binary Integer, Decimal Placement</td>
<td></td>
</tr>
<tr>
<td>Side</td>
<td>41</td>
<td>1</td>
<td>ASCII Character, This field indicates the side of the order Buy/sell. Valid Values: ‘B’ – Buy, ‘S’ – Sell</td>
<td></td>
</tr>
<tr>
<td>SecurityType</td>
<td>43</td>
<td>1</td>
<td>ASCII Character, This field specifies the security type for this message. Valid values: ‘E’ – Equity, ‘B’ – BB</td>
<td></td>
</tr>
<tr>
<td>FirmIndex</td>
<td>44</td>
<td>2</td>
<td>Binary Integer, This field identifies the numerical representation of the firm sending the quote if attributed.</td>
<td></td>
</tr>
</tbody>
</table>
Delete Message Body Format
The table below describes the body fields of an ArcaBook Delete message (MsgType = ‘102’). ArcaBook sends this message when an order is taken off of the NYSE Arca open order book. The following events will trigger the transmission of a delete order message.
• An order is cancelled
• An order expires
• An order is routed to an away market. Note: If the away market declines the NYSE ARCA preference, an Add Order message with the original order id will be sent to return the order to the ArcaBook.
• An order is filled

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Size (Bytes)</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SymbolIndex</td>
<td>16</td>
<td>2</td>
<td>Binary Integer</td>
<td>This field identifies the numerical representation of the symbol. User can combine this value with the session id to obtain a unique key</td>
</tr>
<tr>
<td>MsgType</td>
<td>18</td>
<td>2</td>
<td>Binary Integer</td>
<td>This field identifies the type of message Message ‘102’ – Delete Order Message</td>
</tr>
<tr>
<td>SourceSeqNum</td>
<td>20</td>
<td>4</td>
<td>Binary Integer</td>
<td>This field contains the sequence number assigned by the source system to this message. The sequence number is unique only to a given stock. Hence orders for two different stocks may share the same source sequence number. Please note that the sequence number while it increases serially, it does not increase monotonically.</td>
</tr>
<tr>
<td>SourceTime</td>
<td>24</td>
<td>4</td>
<td>Binary Integer</td>
<td>This field specifies the quote generation time. The number in this field represents the number of milliseconds since midnight of the same day.</td>
</tr>
<tr>
<td>Field</td>
<td>Length</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>OrderId</td>
<td>28</td>
<td>Binary Integer</td>
<td>The Order ID identifies a unique order and will allow customers of NYSE Arca Trades to correlate trades to the order.</td>
<td></td>
</tr>
<tr>
<td>Side</td>
<td>41</td>
<td>ASCII Character</td>
<td>This field indicates the side of the order Buy/sell. Valid Values: ‘B’ – Buy ‘S’ – Sell</td>
<td></td>
</tr>
<tr>
<td>SecurityType</td>
<td>43</td>
<td>ASCII Character</td>
<td>This field specifies the security type for this message. Valid values: ‘E’ – Equity ‘B’ – BB</td>
<td></td>
</tr>
<tr>
<td>SessionID</td>
<td>46</td>
<td>Binary Integer</td>
<td>Identifies the Source Session of the Symbol.</td>
<td></td>
</tr>
<tr>
<td>FirmIndex</td>
<td>44</td>
<td>Binary Integer</td>
<td>This field identifies the numerical representation of the firm sending the quote if attributed.</td>
<td></td>
</tr>
<tr>
<td>FILLER</td>
<td>47</td>
<td>ASCII String</td>
<td>This is a filler, reserved for future use.</td>
<td></td>
</tr>
</tbody>
</table>