# Design Review: Reception, Processing, and Display of Stock Market Data Using the Solarflare SFA6900 FPGA Board

Team Name: We Love Money
Naman Parashar (np2437), Miles Sherman (ms4543), Pranav Sood (ps2729), Kevin Wong (kw2500), and Art Iakovlev (ai2283)

March 26, 2013

## 1 Overview

As was discussed in class, a major obstacle for stock markets around the world is the management of market data that is continuously growing in quantity and complexity. Classically, the transmission, receipt, and processing of this data has been handled by software. However, because of the exponential growth in data as well as latency requirements, hardware is quickly making its way into the industry. The most notable example of this would be the way data is transmitted from the NASDAQ stock exchange. The NASDAQ currently transmits market data both from classical software platforms and more recently from a high speed FPGA platform for extremely low latency.

For our project, we plan to utilize the Solarflare AoE FPGA card to design a system to receive stock market data, log it in real time, and broadcast the data (to an x86 software we will develop). Thanks to this new card, which can receive and transmit data at up to 10Gbits/s, we will be able to maintain the L3 market data book for a number of stock symbols in real time (assuming access to real time NASDAQ market data as provided by David Lariviere, otherwise the market data stream will be simulated based on previous market snapshots), build a human-readable L2 book for each stock symbol we track, and output the information to a broadcast.

An L3 market book maintains up to date information on every open order associated with a certain stock. Each order contains information such as ticker symbol, order reference number, order price, bid or ask indicator, and order size (number of shares).

An L2 market book references the L3 book and maintains up to date information on the status of every stock in a certain exchange. For each stock, this information includes ticker symbol, the last traded price, and information regarding a limited number of bid and ask price levels.

## 2 Functionality

The high level block diagram of our design can be seen in Figure 1.

Data will initially be inputted in an 8-byte wide stream to the packetizer module. This module strips the IP header and feeds the UDP payload to the next module. The MoldUDP packetizer module will do a number of things. First, it will strip the Nasdaq MoldUDP header which contains information on the payload length as well as the number of messages contained within the payload. It will then parse the MoldUDP payload into individual 24-byte Nasdaq ITCH formatted messages. Each time a message is accumulated, it outputs it on a 24-byte wide bus to the L3 Book Builder module. This module controls the sorting and organization of the L3 Book RAM. The message is passed to the correct location in the L3 RAM and an enable signal is also fed to the L2 Book Builder module. The L2 book builder module grabs any changes in the L3 book and updates the necessary locations in the L2 Book RAM. Finally, the broadcaster is a simple x86 software that reads the L2 Book RAM and displays the information on the host client's monitor for human access.

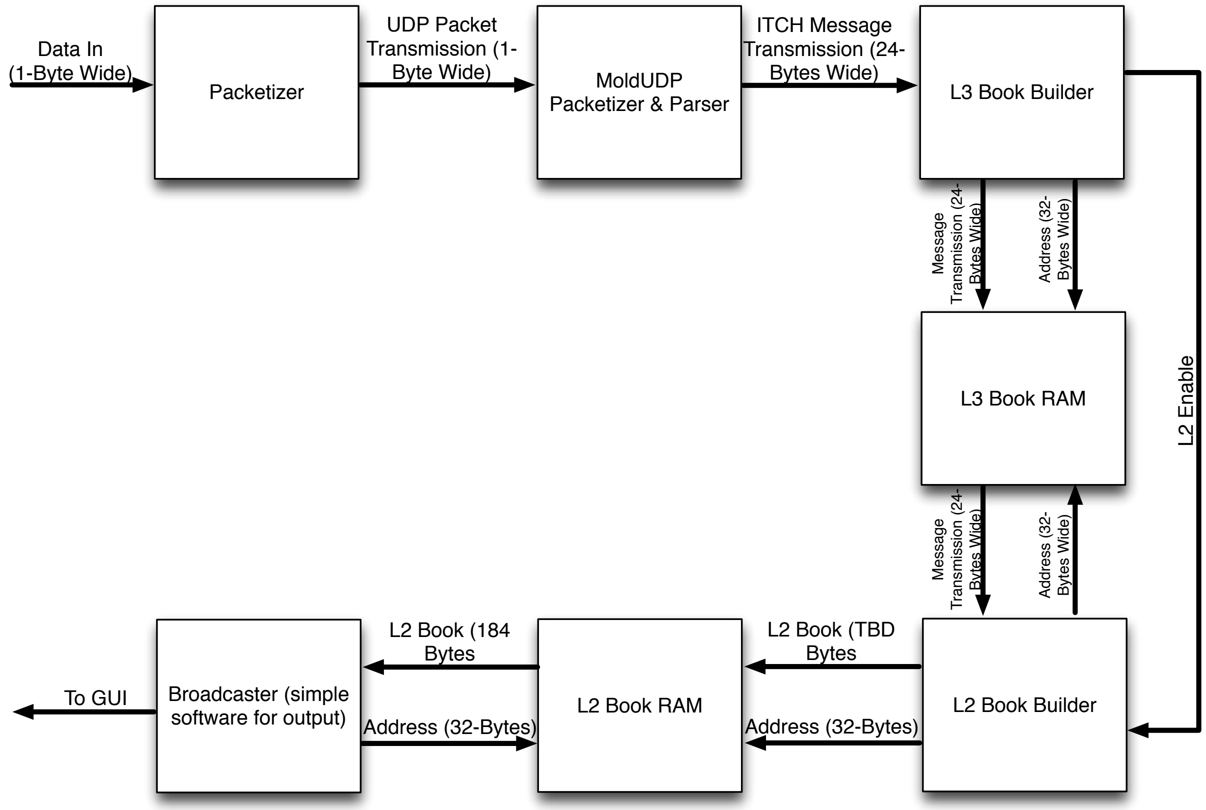
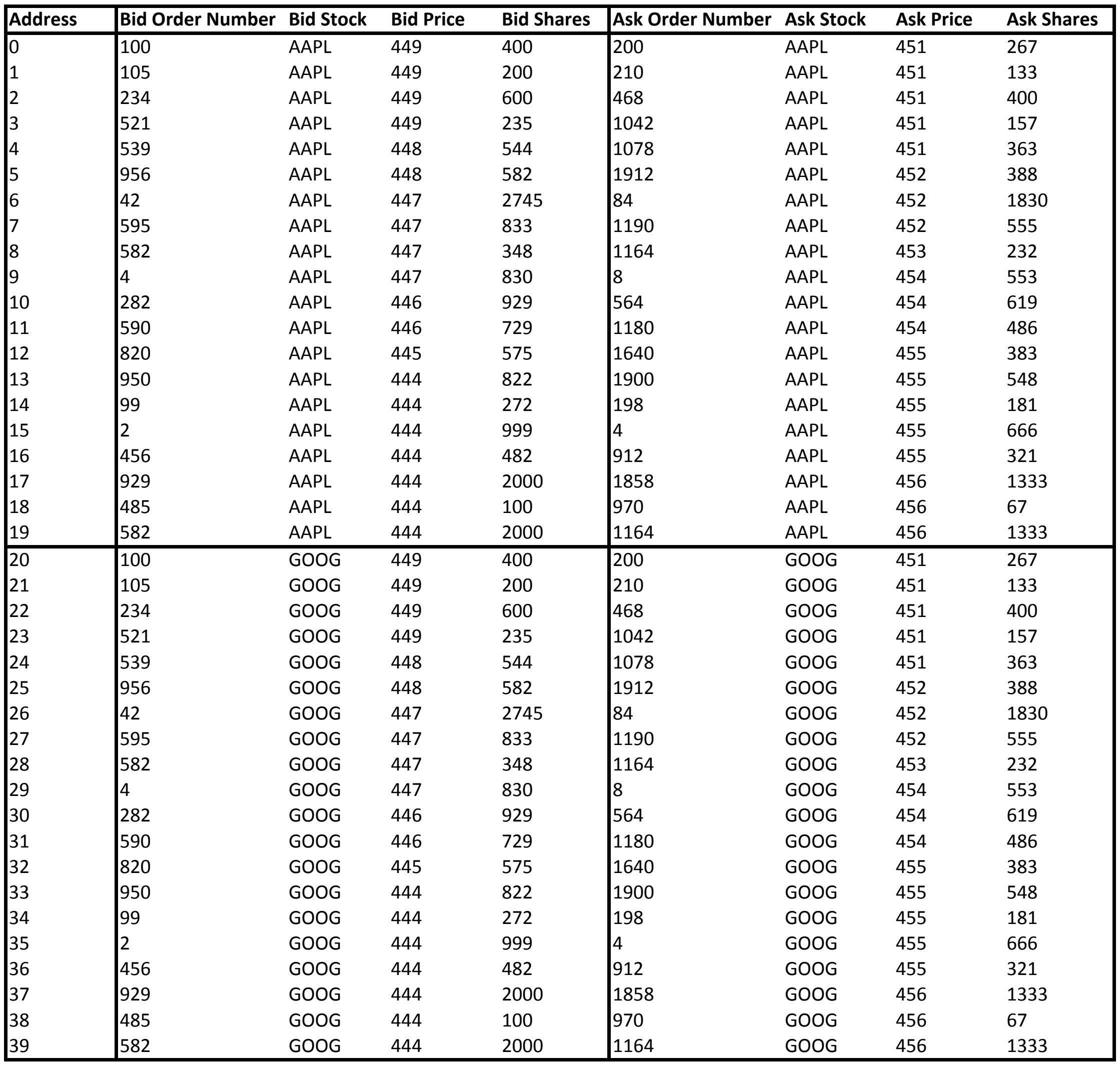

Data In (1-Byte Wide)
Packetizer
UDP Packet Transmission (1-Byte Wide)
MoldUDP Packetizer & Parser
ITCH Message Transmission (24-Bytes Wide)
L3 Book Builder
Message Transmission (24-Bytes Wide)
Address (32-Bytes Wide)
L3 Book RAM
L2 Enable
Message Transmission (24-Bytes Wide)
Address (32-Bytes Wide)
L2 Book Builder
L2 Book (TBD Bytes
Address (32-Bytes)
L2 Book RAM
L2 Book (184 Bytes
Address (32-Bytes)
Broadcaster (simple software for output)
To GUI

Figure 1: Top Level Block Diagram

## 3 Memory Requirements

Our system will require two significant blocks of memory, the L3 Book RAM and the L2 Book RAM. We plan to implement these two blocks using the on-chip memory of the Stratix V FPGA chip. The total memory is on the order of 7MB which should suffice if we limit the number of stocks we track.

The L3 Book RAM will have the largest size requirement because it must maintain every open order for each tracked stock. While the exact memory requirements for this block will be determined by the amount of activity during a certain day for the chosen stocks, we can make an estimate. In addition, we have determined the exact structure of the memory including the width of the array. Figure 2 shows an example of how the memory will be organized. This configuration will require a RAM array that is 352 bits wide (22 bytes for the bid columns and 22 bytes for the ask columns). We estimate that if we track three low-volume stocks, the L3 book for each stock will reach a maximum of 5k bid or ask orders. This would give us a total requirement of 1.76Mb for the L3 Book RAM.



| Address | Bid Order Number | Bid Stock | Bid Price | Bid Shares | Ask Order Number | Ask Stock | Ask Price | Ask Shares |
|---|---|---|---|---|---|---|---|---|
| 0 | 100 | AAPL | 449 | 400 | 200 | AAPL | 451 | 267 |
| 1 | 105 | AAPL | 449 | 200 | 210 | AAPL | 451 | 133 |
| 2 | 234 | AAPL | 449 | 600 | 468 | AAPL | 451 | 400 |
| 3 | 521 | AAPL | 449 | 235 | 1042 | AAPL | 451 | 157 |
| 4 | 539 | AAPL | 448 | 544 | 1078 | AAPL | 451 | 363 |
| 5 | 956 | AAPL | 448 | 582 | 1912 | AAPL | 452 | 388 |
| 6 | 42 | AAPL | 447 | 2745 | 84 | AAPL | 452 | 1830 |
| 7 | 595 | AAPL | 447 | 833 | 1190 | AAPL | 452 | 555 |
| 8 | 582 | AAPL | 447 | 348 | 1164 | AAPL | 453 | 232 |
| 9 | 4 | AAPL | 447 | 830 | 8 | AAPL | 454 | 553 |
| 10 | 282 | AAPL | 446 | 929 | 564 | AAPL | 454 | 619 |
| 11 | 590 | AAPL | 446 | 729 | 1180 | AAPL | 454 | 486 |
| 12 | 820 | AAPL | 445 | 575 | 1640 | AAPL | 455 | 383 |
| 13 | 950 | AAPL | 444 | 822 | 1900 | AAPL | 455 | 548 |
| 14 | 99 | AAPL | 444 | 272 | 198 | AAPL | 455 | 181 |
| 15 | 2 | AAPL | 444 | 999 | 4 | AAPL | 455 | 666 |
| 16 | 456 | AAPL | 444 | 482 | 912 | AAPL | 455 | 321 |
| 17 | 929 | AAPL | 444 | 2000 | 1858 | AAPL | 456 | 1333 |
| 18 | 485 | AAPL | 444 | 100 | 970 | AAPL | 456 | 67 |
| 19 | 582 | AAPL | 444 | 2000 | 1164 | AAPL | 456 | 1333 |
| 20 | 100 | GOOG | 449 | 400 | 200 | GOOG | 451 | 267 |
| 21 | 105 | GOOG | 449 | 200 | 210 | GOOG | 451 | 133 |
| 22 | 234 | GOOG | 449 | 600 | 468 | GOOG | 451 | 400 |
| 23 | 521 | GOOG | 449 | 235 | 1042 | GOOG | 451 | 157 |
| 24 | 539 | GOOG | 448 | 544 | 1078 | GOOG | 451 | 363 |
| 25 | 956 | GOOG | 448 | 582 | 1912 | GOOG | 452 | 388 |
| 26 | 42 | GOOG | 447 | 2745 | 84 | GOOG | 452 | 1830 |
| 27 | 595 | GOOG | 447 | 833 | 1190 | GOOG | 452 | 555 |
| 28 | 582 | GOOG | 447 | 348 | 1164 | GOOG | 453 | 232 |
| 29 | 4 | GOOG | 447 | 830 | 8 | GOOG | 454 | 553 |
| 30 | 282 | GOOG | 446 | 929 | 564 | GOOG | 454 | 619 |
| 31 | 590 | GOOG | 446 | 729 | 1180 | GOOG | 454 | 486 |
| 32 | 820 | GOOG | 445 | 575 | 1640 | GOOG | 455 | 383 |
| 33 | 950 | GOOG | 444 | 822 | 1900 | GOOG | 455 | 548 |
| 34 | 99 | GOOG | 444 | 272 | 198 | GOOG | 455 | 181 |
| 35 | 2 | GOOG | 444 | 999 | 4 | GOOG | 455 | 666 |
| 36 | 456 | GOOG | 444 | 482 | 912 | GOOG | 455 | 321 |
| 37 | 929 | GOOG | 444 | 2000 | 1858 | GOOG | 456 | 1333 |
| 38 | 485 | GOOG | 444 | 100 | 970 | GOOG | 456 | 67 |
| 39 | 582 | GOOG | 444 | 2000 | 1164 | GOOG | 456 | 1333 |

Figure 2: L3 Book RAM Configuration

The L2 Book RAM will have a much smaller size requirement and this requirement will be fixed depending on the number of tracked stocks as well as the number of bid and ask price levels outputted. The L2 Book RAM will be 184 bits wide and will be configured according to Figure 3 and its size will be determined by

$$L2\ RAM\ Size\ Requirement = \\ (Number\ of\ Tracked\ Stocks) * (1 + 2 * (Number\ of\ Tracked\ Bid/Ask\ Levels)) * 184\ bits.$$

For a system tracking 3 stocks with 10 levels of bid/ask pricing, this will result in an L2 Book RAM memory requirement of only 11kb.

| Address | Field | Stock | Price | # of Orders |
|---|---|---|---|---|
| 0 | Last Trade | AAPL | 450 | X |
| 1 | Ask Level 10 | AAPL | 460 | 200 |
| 2 | Ask Level 9 | AAPL | 459 | 600 |
| 3 | Ask Level 8 | AAPL | 458 | 235 |
| 4 | Ask Level 7 | AAPL | 457 | 544 |
| 5 | Ask Level 6 | AAPL | 456 | 582 |
| 6 | Ask Level 5 | AAPL | 455 | 2745 |
| 7 | Ask Level 4 | AAPL | 454 | 833 |
| 8 | Ask Level 3 | AAPL | 453 | 348 |
| 9 | Ask Level 2 | AAPL | 452 | 830 |
| 10 | Ask Level 1 | AAPL | 451 | 929 |
| 11 | Bid Level 1 | AAPL | 449 | 729 |
| 12 | Bid Level 2 | AAPL | 448 | 575 |
| 13 | Bid Level 3 | AAPL | 447 | 822 |
| 14 | Bid Level 4 | AAPL | 446 | 272 |
| 15 | Bid Level 5 | AAPL | 445 | 999 |
| 16 | Bid Level 6 | AAPL | 444 | 482 |
| 17 | Bid Level 7 | AAPL | 443 | 2000 |
| 18 | Bid Level 8 | AAPL | 442 | 100 |
| 19 | Bid Level 9 | AAPL | 441 | 2000 |
| 20 | Bid Level 10 | AAPL | 440 | 400 |
| 21 | Last Trade | GOOG | 450 | X |
| 22 | Ask Level 10 | GOOG | 460 | 200 |
| 23 | Ask Level 9 | GOOG | 459 | 600 |
| 24 | Ask Level 8 | GOOG | 458 | 235 |
| 25 | Ask Level 7 | GOOG | 457 | 544 |
| 26 | Ask Level 6 | GOOG | 456 | 582 |
| 27 | Ask Level 5 | GOOG | 455 | 2745 |
| 28 | Ask Level 4 | GOOG | 454 | 833 |
| 29 | Ask Level 3 | GOOG | 453 | 348 |
| 30 | Ask Level 2 | GOOG | 452 | 830 |
| 31 | Ask Level 1 | GOOG | 451 | 929 |
| 32 | Bid Level 1 | GOOG | 449 | 729 |
| 33 | Bid Level 2 | GOOG | 448 | 575 |
| 34 | Bid Level 3 | GOOG | 447 | 822 |
| 35 | Bid Level 4 | GOOG | 446 | 272 |
| 36 | Bid Level 5 | GOOG | 445 | 999 |
| 37 | Bid Level 6 | GOOG | 444 | 482 |
| 38 | Bid Level 7 | GOOG | 443 | 2000 |
| 39 | Bid Level 8 | GOOG | 442 | 100 |
| 40 | Bid Level 9 | GOOG | 441 | 2000 |
| 41 | Bid Level 10 | GOOG | 440 | 400 |

Figure 3: L2 Book RAM Configuration

## 4 Critical Path

Our design has a number of small bottlenecks and one large bottleneck. The two most significant small bottlenecks are the accumulation of ITCH messages at the output of the MoldUDP packetizer and the refreshing of the L2 book. The large bottleneck is the algorithm to find the correct address for new orders added to the L3 Book (this takes place in the L3 Book builder). As a result of these bottlenecks, our critical data path is as follows (this is for an add-order operation):

$Data\ In \rightarrow Data\ To\ MoldUDP\ Packetizer \rightarrow Message\ Accumulates \rightarrow Message\ To\ L3\ Builder \rightarrow$
$Message\ Routed\ To\ Add\ Order\ Module\ (within\ L3\ Builder) \rightarrow Add\ Order\ Module\ Runs \rightarrow$
$Order\ Routed\ To\ L3\ RAM \rightarrow Order\ Read\ By\ L2\ Builder \rightarrow L2\ Book\ Refreshed \rightarrow$
$L2\ Book\ Routed\ To\ L2\ RAM \rightarrow L2\ Book\ Routed\ To\ Broadcaster$

## 5 Module-Level Functionality

### 5.1 Packetizer

The packetizer module is our system's interface to the raw incoming data (see Figure 4 for the block diagram). The module inputs a serial connections that is one byte wide. The data first interfaces with a multiplexer that routes the byte either to the IP Header register if it is one of the first 42 bytes of the sequence or directly to the output (also one byte wide) if the IP header has already been striped and the byte is determined to be part of the UDP payload.

To determine if the incoming byte is part of the header or the payload, the packetizer uses a comparator between the byte counter's output (the byte number of the current sequence) and the literal '42' (the length of the IP header. Once the IP header is striped, the module switches the multiplexer at the input and feeds the bytes through to the output until the sequence byte counter reaches the total which is indicated by the IP header. Once this condition is met, the system resets and inputs the next IP Header.

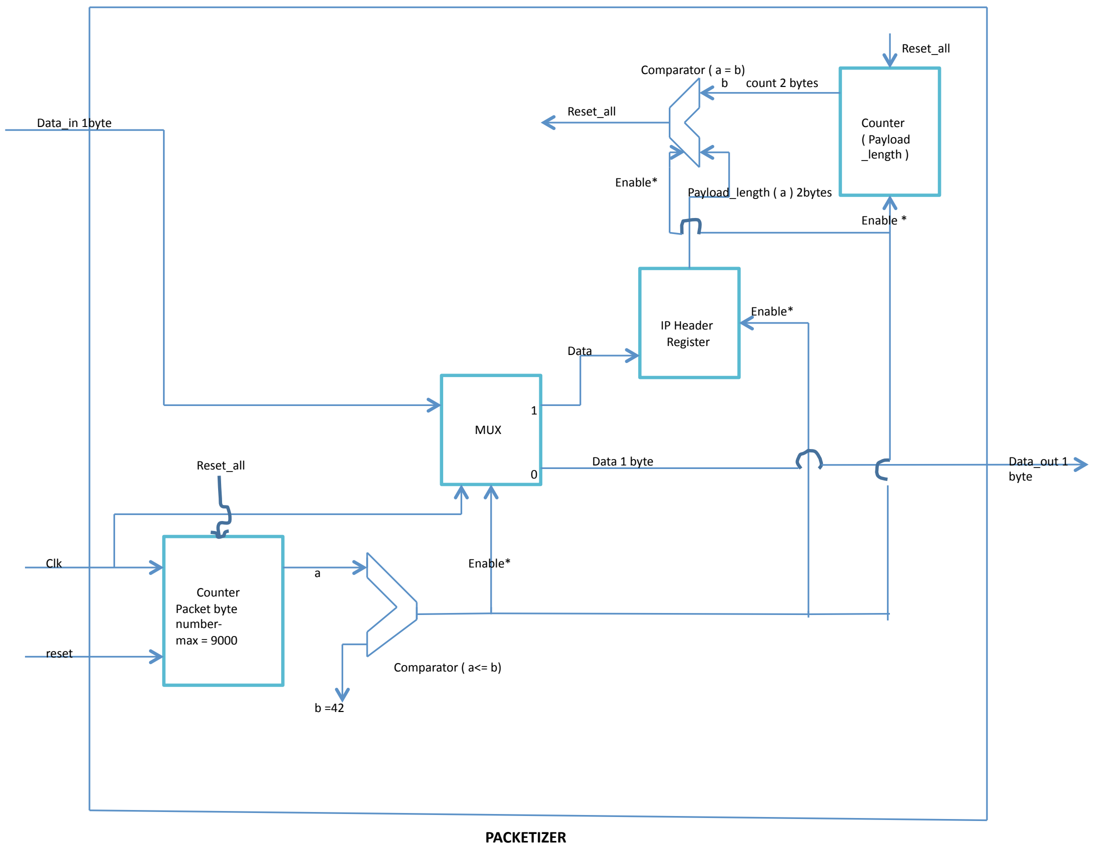


Figure 4: Packetizer Block Diagram

## 5.2 MoldUDP Packetizer & Parser

The MoldUDP module takes as input 1 byte of data and outputs a 24-byte long message.

The MoldUDP Header counter keeps track of the number of bytes that has been received by the module for each UDP payload packet. Once the packet is completely received, the counter resets itself to 0. The only use for this counter is to separate the MoldUDP header from the payload, as inferred by the comparator that the counter feeds into only switching once per UDP packet.

Comparator 1 controls the mux which data-in directly feeds into. The purpose of this comparator and mux is to solely separate the MoldUDP header and MoldUDP message. The length of the MoldUDP header is constant so the comparator evaluates $20 > b$, where b is the MoldUDP Header counter value. Initially, Comparator 1 will evaluate for true as long as the header is still being loaded. When the header length is exceeded, as indicated by the counter, the comparator output will flip and remain to 0 for the duration of the MoldUDP packet. The comparator output also serves as an enable signal for the Message Length counter, which will be described later.

The first twenty bytes of data that enters the module will be the header of the packet and as such, will be routed to the block labeled MoldUDP Header Register. The two bytes in the header corresponding to the total amount of messages included in the packet will be sent to Comparator 2.

After the first twenty bytes of data is routed to the MoldUDP Header Register, the rest of the message bytes will be sent to the MoldUDP Message Register. The first two bytes of each message contains the length of that message, and as such will be extracted to a comparator ($a = b?$) which compares said value to the value of the Message Length Counter. The output of the comparator is a single bit signal back to the MoldUDP Message

Register which enables the register to send the 42-byte message to the Message Compressor block.

The third byte of the MoldUDP Message Register, which indicates the message type, is fed to the Message Type Lookup Table. The Message Type Lookup Table determines the type of message that is in the Message Register, and send the value to the Message compressor.

The Message Length counter keeps count of the amount of bytes corresponding to a message. This is accomplished by disabling the counter when the MoldUDP header is being loaded, and resetting the counter back to 0 every time a message is sent out (signal controlled by Message Compressor).

The Message Compressor takes in the 42-byte message in parallel from the Message register when a message is fully loaded into the registers (comparator $a = b?$ evaluates to 1). The compressor also takes in the message type and based on the message type value, crops out the unneeded portions of the message, as well as ensuring that the message output is 24 bytes long. When the Message compressor is sends the data to the L3 Builder, it also sends a reset signal to Message Length Counter as well as an enable signal to Message Counter.

The Message Counter keeps count of the number of messages that has been launched by the Message Compressor. This counter also takes a reset signal whenever Comparator 2 becomes high, indicating that the total amount of messages in the UDP packet has been reached.

The function of comparator 2 is to send a signal when the number of messages that the module has processed is equal to the number of messages indicated in the MoleUDP Header (end of packet). The output signal resets Message Counter as explained above, and also resets the MoldUDP header counter so that the next MoldUDP packet can be read in properly.

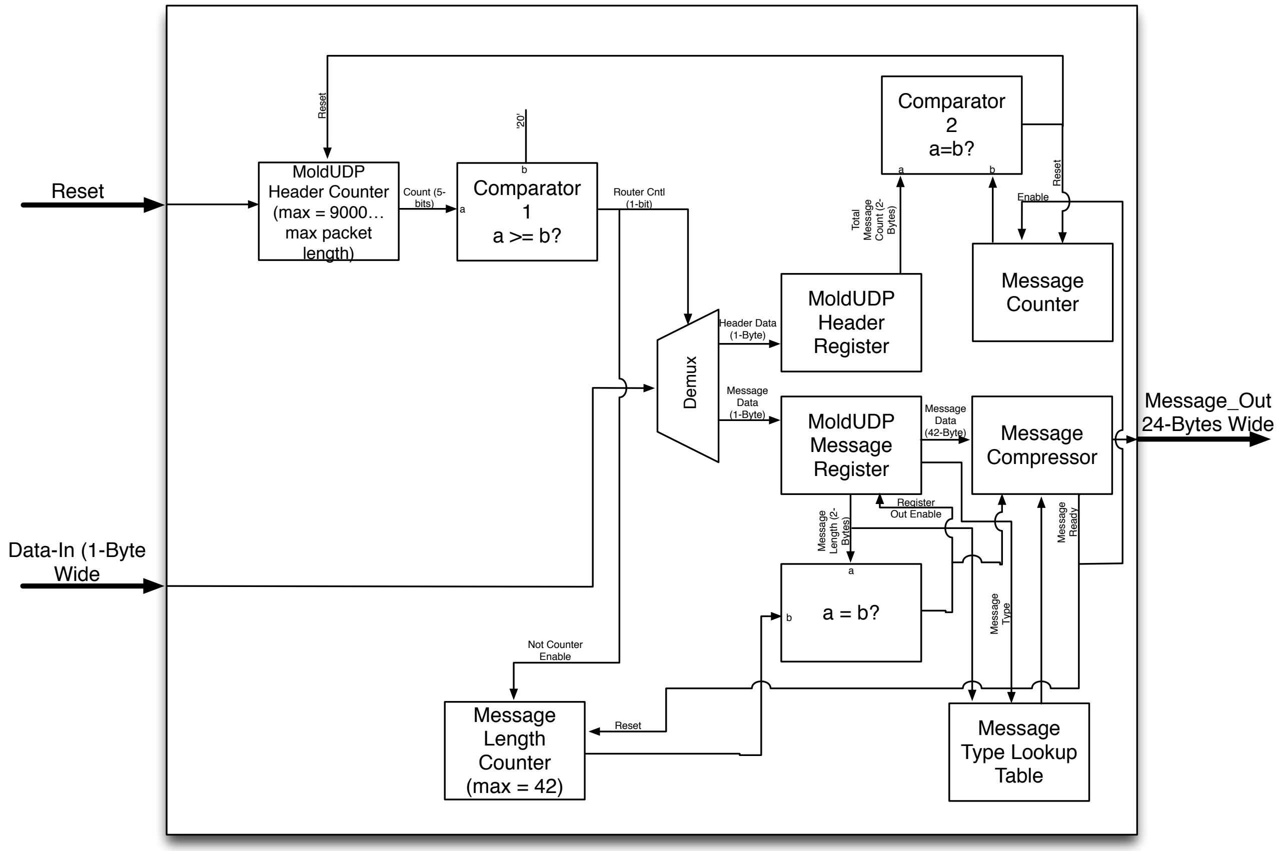


Figure 5: MoldUDP Packetizer & Parser Block Diagram

## 5.3 L3 Book Builder

The L3 Book Builder module is an integral part of the book building process. The purpose of this module, who's block diagram can be seen in Figure 6 is to input a pre-parsed Nasdaq ITCH message, determine the message type (this could be add order, remove order, modify order, transaction, etc), and perform all necessary memory modifications to update the L3 Book RAM appropriately. The first stage of the module detects the message type which is always contained in the first byte of the message. It strips this byte from the bus and uses a Demux (controlled by the message type) to route the data to the appropriate submodule.

For add-order messages, the buy/sell (bid/ask) indicator is striped from the message and controls an output signal called Ask Enable. The Ask Enable signal tells the L3 RAM whether to use the left (bid) or right (ask) columns of the array. The new order is then passed to the add order lookup algorithm which does a number of things. It first finds the correct address for the new order using a pseudo binary tree approach, it then shifts all of the orders above that address up by one address in RAM, and finally it inserts the new order in the correct position in RAM.

For delete-order messages, the message is passed directly to the remove order algorithm. This submodule first finds the correct address of the order to be removed using the order's reference number as a test condition for indexing. It then shifts all orders above that address down by one, effectively removing the correct order from memory. The execute-order message does the same as the delete-order message except it passes information on the executed order to the L2-Book Builder to update the last-trade price.

For order-executed with price orders, the message is passed directly to the L2 Book Builder to update the last-trade price. This operation is the equivalent of adding an order and immediately executing it so no action is necessary on the L3 book.

For order-replace and order-cancel messages, we use an algorithm called the replace order algorithm. This submodule simply finds the correct address based on the order reference number, and replaces that address with the new order inputted if the message is for order replace. If the message is order-cancel (removing shares from the order), then the order currently at the calculated address is read into a temporary register, the quantity of shares is modified, and the order is written back to that same memory location.

The ports of this module interface only with other hardware modules including the MoldUDP Packetizer, the L3 Book RAM, and the L2 Book Builder.

The pseudo codes for the three submodules in the L3 Book Builder can be seen in Figures 7, 8, and 9.

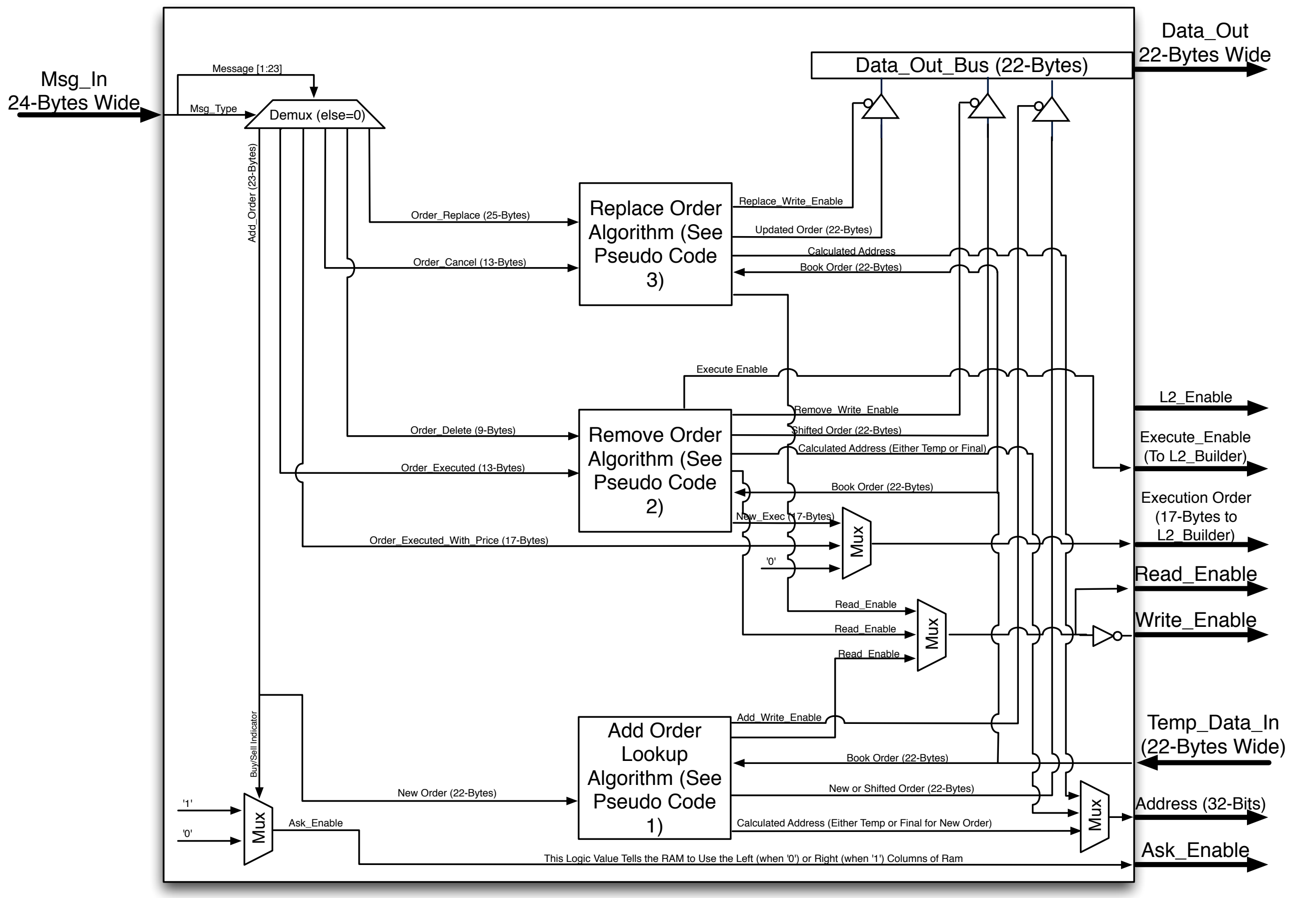
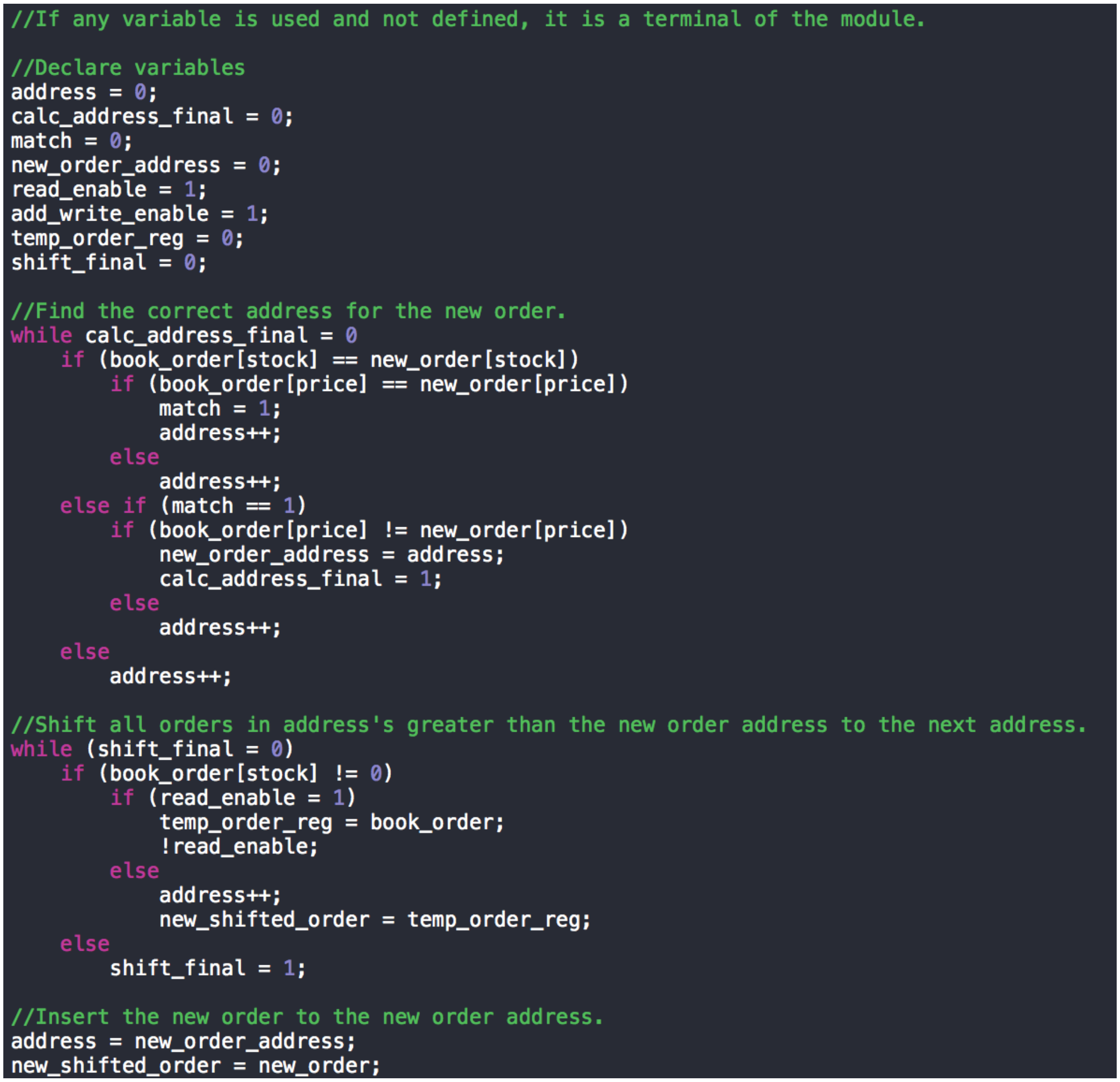
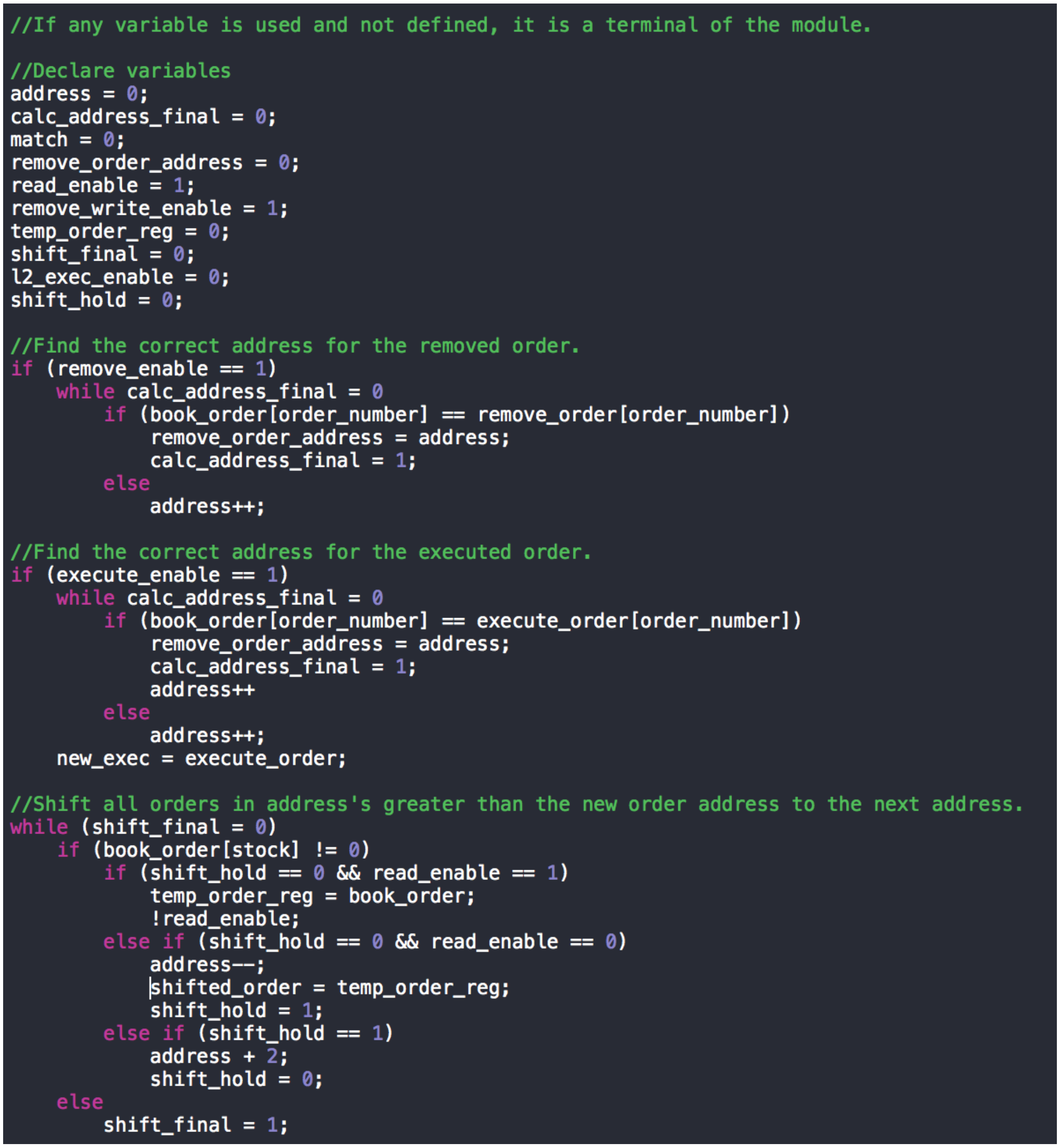


Figure 6: L3 Book Builder Block Diagram



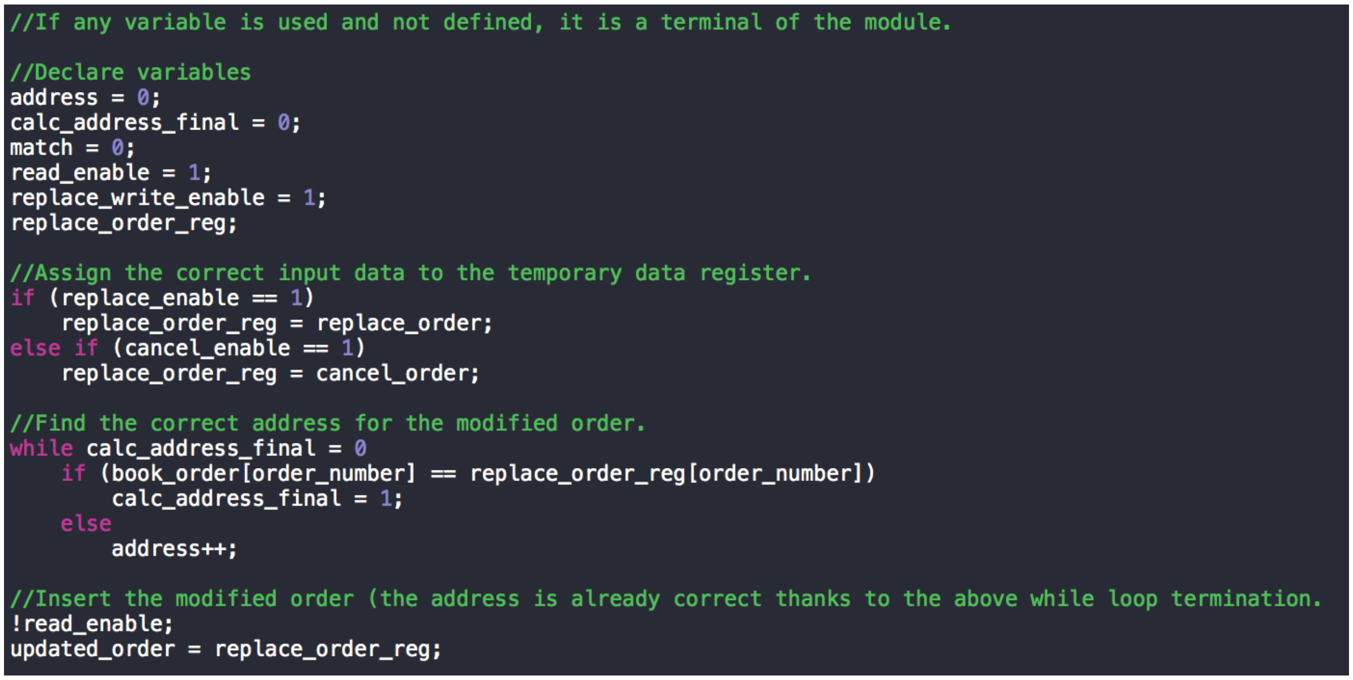
```
//If any variable is used and not defined, it is a terminal of the module.

//Declare variables
address = 0;
calc_address_final = 0;
match = 0;
new_order_address = 0;
read_enable = 1;
add_write_enable = 1;
temp_order_reg = 0;
shift_final = 0;

//Find the correct address for the new order.
while calc_address_final = 0
    if (book_order[stock] == new_order[stock])
        if (book_order[price] == new_order[price])
            match = 1;
            address++;
        else
            address++;
    else if (match == 1)
        if (book_order[price] != new_order[price])
            new_order_address = address;
            calc_address_final = 1;
        else
            address++;
    else
        address++;

//Shift all orders in address's greater than the new order address to the next address.
while (shift_final = 0)
    if (book_order[stock] != 0)
        if (read_enable = 1)
            temp_order_reg = book_order;
            !read_enable;
        else
            address++;
            new_shifted_order = temp_order_reg;
    else
        shift_final = 1;

//Insert the new order to the new order address.
address = new_order_address;
new_shifted_order = new_order;
```

Figure 7: Pseudo Code 1: The Add Order Lookup Algorithm



```
//If any variable is used and not defined, it is a terminal of the module.

//Declare variables
address = 0;
calc_address_final = 0;
match = 0;
remove_order_address = 0;
read_enable = 1;
remove_write_enable = 1;
temp_order_reg = 0;
shift_final = 0;
l2_exec_enable = 0;
shift_hold = 0;

//Find the correct address for the removed order.
if (remove_enable == 1)
    while calc_address_final = 0
        if (book_order[order_number] == remove_order[order_number])
            remove_order_address = address;
            calc_address_final = 1;
        else
            address++;

//Find the correct address for the executed order.
if (execute_enable == 1)
    while calc_address_final = 0
        if (book_order[order_number] == execute_order[order_number])
            remove_order_address = address;
            calc_address_final = 1;
            address++
        else
            address++;
    new_exec = execute_order;

//Shift all orders in address's greater than the new order address to the next address.
while (shift_final = 0)
    if (book_order[stock] != 0)
        if (shift_hold == 0 && read_enable == 1)
            temp_order_reg = book_order;
            !read_enable;
        else if (shift_hold == 0 && read_enable == 0)
            address--;
            shifted_order = temp_order_reg;
            shift_hold = 1;
        else if (shift_hold == 1)
            address + 2;
            shift_hold = 0;
    else
        shift_final = 1;
```

Figure 8: Pseudo Code 2: The Remove Order Lookup Algorithm



```
//If any variable is used and not defined, it is a terminal of the module.

//Declare variables
address = 0;
calc_address_final = 0;
match = 0;
read_enable = 1;
replace_write_enable = 1;
replace_order_reg;

//Assign the correct input data to the temporary data register.
if (replace_enable == 1)
    replace_order_reg = replace_order;
else if (cancel_enable == 1)
    replace_order_reg = cancel_order;

//Find the correct address for the modified order.
while calc_address_final = 0
    if (book_order[order_number] == replace_order_reg[order_number])
        calc_address_final = 1;
    else
        address++;

//Insert the modified order (the address is already correct thanks to the above while loop termination.
!read_enable;
updated_order = replace_order_reg;
```

Figure 9: Pseudo Code 3: The Replace Order Lookup Algorithm

## 5.4 L2 Book Builder

The L2 Book Builder module is the interface between the raw sorted data of the L3 Book RAM and the L2 book which will present data that is easily understood and utilized by the user. This module (block diagram shown in Figure 10) has a number of inputs. First it inputs the execution order and if the execute enable signal (from the L3 Book Builder) is high, extracts the price of the execution order and writes it to the last trade field for the correct stock (fixed address for each stock) in the L2 Book RAM.

When execute enable is low, the L2 Builder operates as a slave to the L2 Enable signal which is sent from the L3 Book Builder. When this signal is high, the module sweeps through all of the orders in the L3 Book for each stock. It completely builds the updated version of the L2 book by totaling the number of orders at each price level for each stock. This algorithm is represented in Figure 10 as the Levels for Ask/Bid submodule and can be seen in the pseudo code of Figure 11.

Similar to the L3 Book Builder, this module interfaces only to hardware modules. These modules include the L3 Book RAM and the L2 Book RAM.

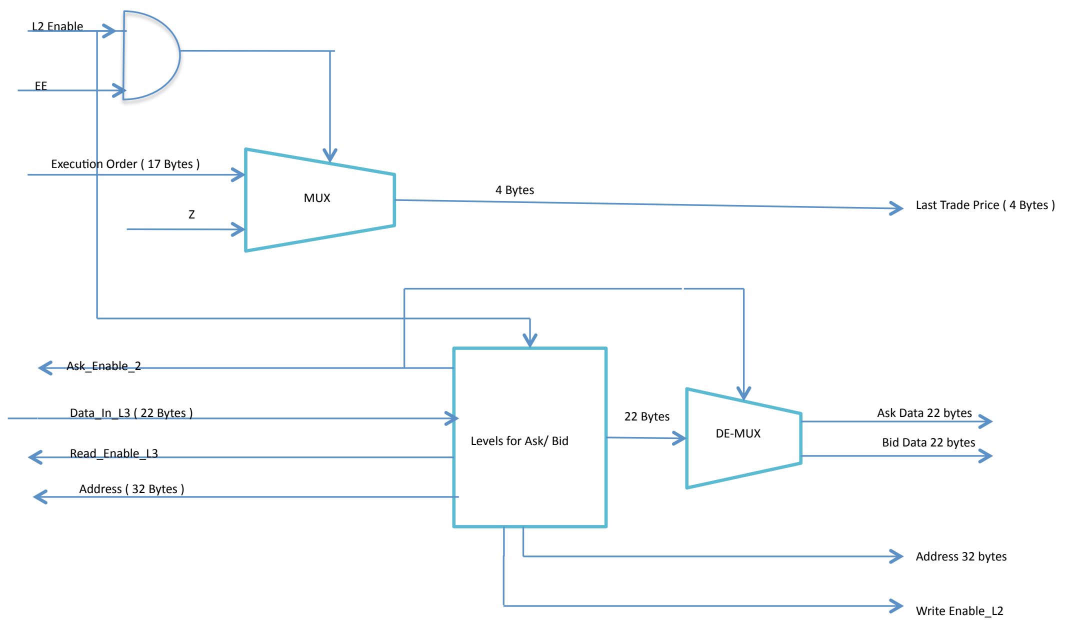
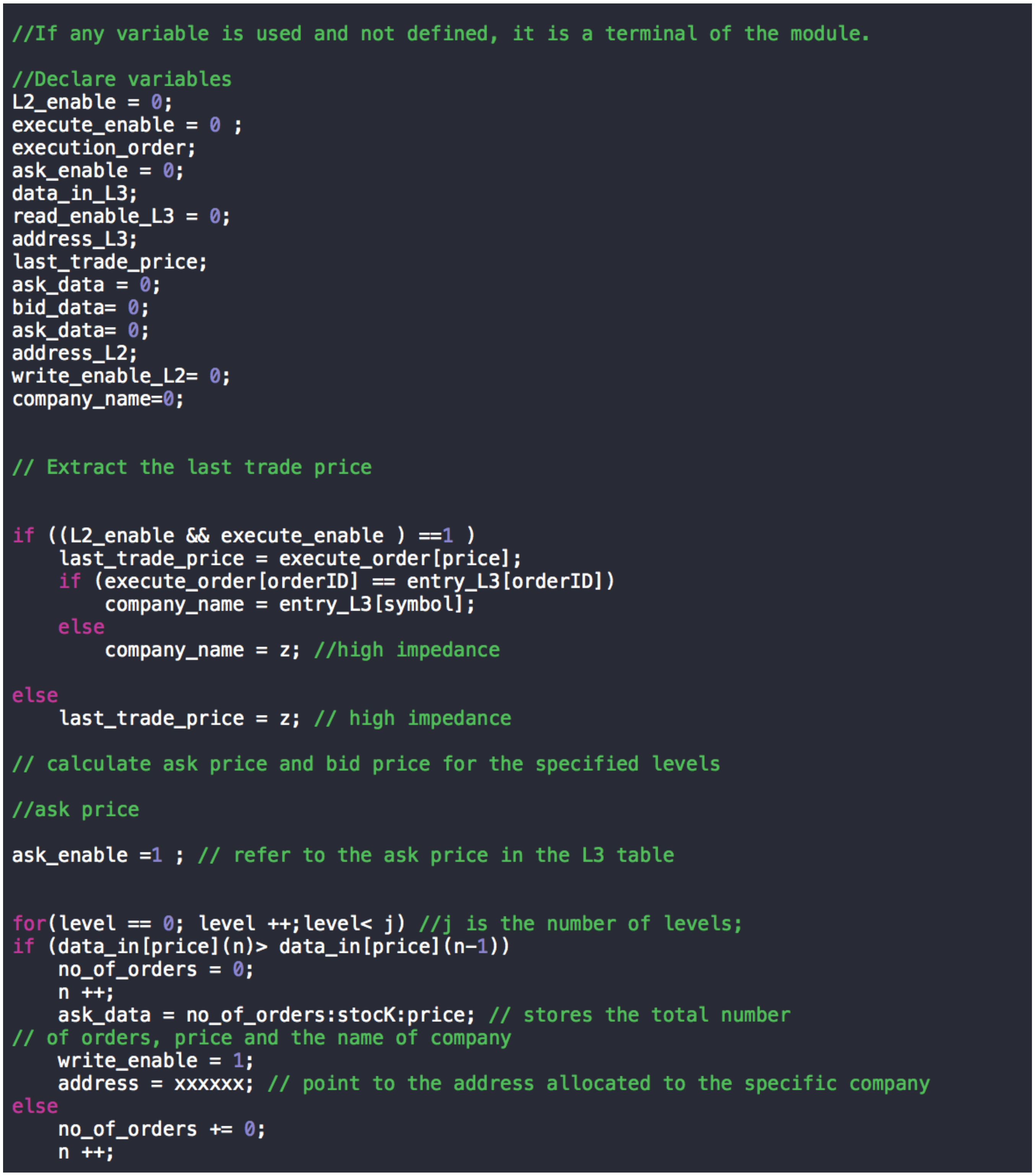
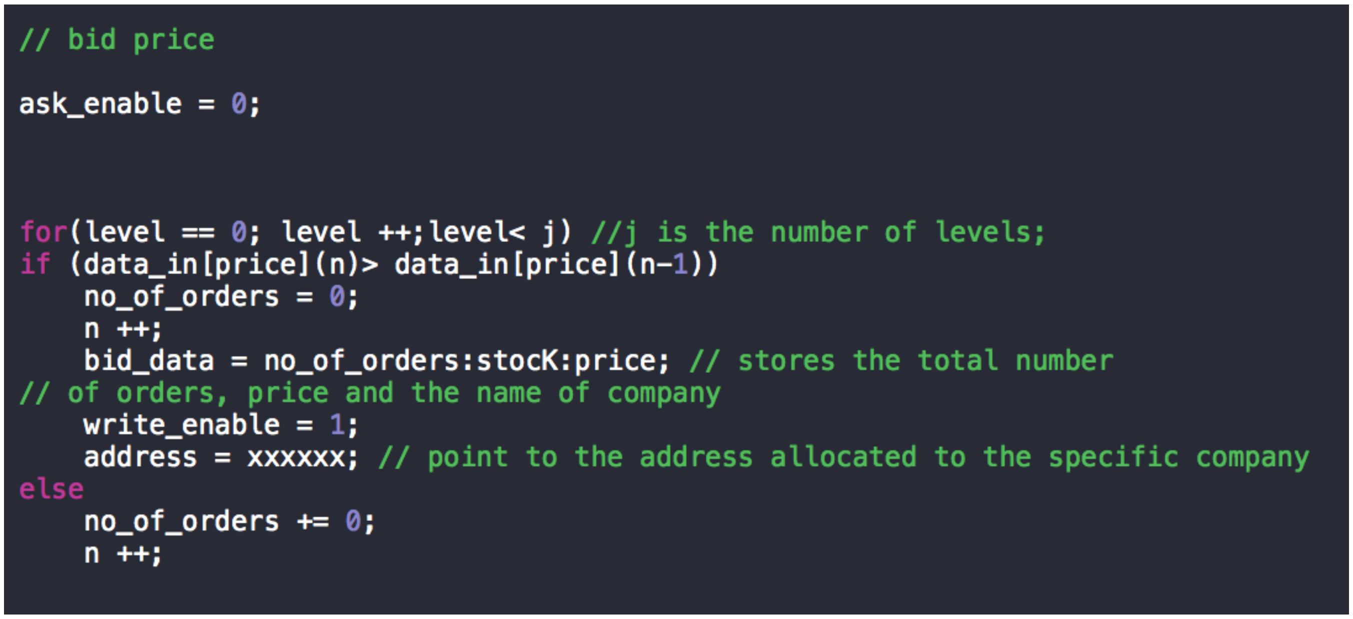


Figure 10: Block Diagram for the L2 Book Builder



```
//If any variable is used and not defined, it is a terminal of the module.

//Declare variables
L2_enable = 0;
execute_enable = 0 ;
execution_order;
ask_enable = 0;
data_in_L3;
read_enable_L3 = 0;
address_L3;
last_trade_price;
ask_data = 0;
bid_data= 0;
ask_data= 0;
address_L2;
write_enable_L2= 0;
company_name=0;


// Extract the last trade price


if ((L2_enable && execute_enable ) ==1 )
    last_trade_price = execute_order[price];
    if (execute_order[orderID] == entry_L3[orderID])
        company_name = entry_L3[symbol];
    else
        company_name = z; //high impedance

else
    last_trade_price = z; // high impedance

// calculate ask price and bid price for the specified levels

//ask price

ask_enable =1 ; // refer to the ask price in the L3 table


for(level == 0; level ++;level< j) //j is the number of levels;
if (data_in[price](n)> data_in[price](n-1))
    no_of_orders = 0;
    n ++;
    ask_data = no_of_orders:stocK:price; // stores the total number
// of orders, price and the name of company
    write_enable = 1;
    address = xxxxxx; // point to the address allocated to the specific company
else
    no_of_orders += 0;
    n ++;
```

Figure 11: Pseudo Code 4: The L2 Book Building Algorithm (Part 1)



```
// bid price

ask_enable = 0;



for(level == 0; level ++;level< j) //j is the number of levels;
if (data_in[price](n)> data_in[price](n-1))
    no_of_orders = 0;
    n ++;
    bid_data = no_of_orders:stocK:price; // stores the total number
// of orders, price and the name of company
    write_enable = 1;
    address = xxxxxx; // point to the address allocated to the specific company
else
    no_of_orders += 0;
    n ++;
```

Figure 12: Pseudo Code 4: The L2 Book Building Algorithm (Part 2)

## 5.5 Broadcaster

This module is strictly a software module that shares the L2 Book RAM using memory-mapped IO. This software simply reads the contents of this RAM block every five seconds or so and outputs its contents into a graphic user interface on the host computer's monitor. This GUI will contain information sorted by stock and then price level for the bids and asks.