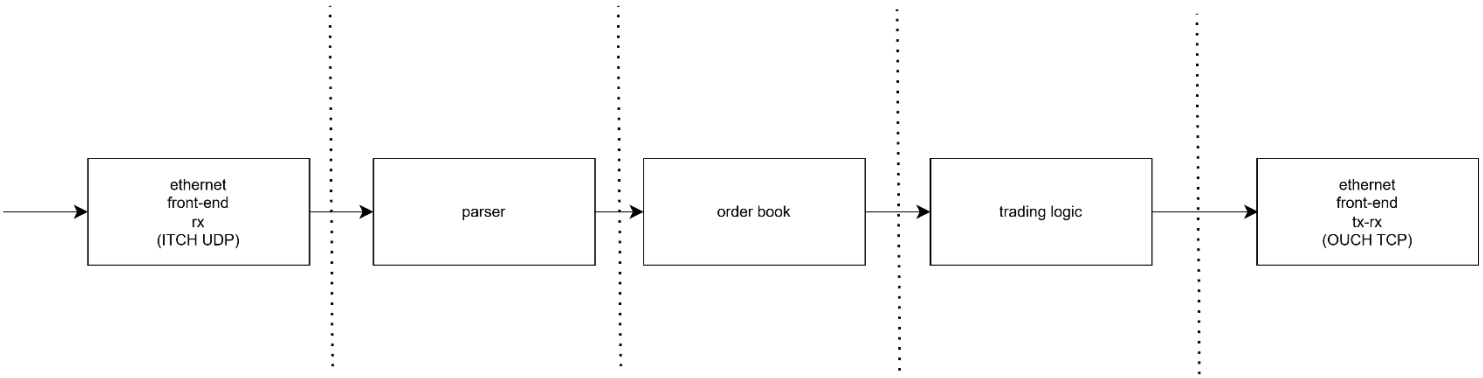
# Nasdaq-ITCH-5.0-OrderBook

Contact Info:
Varadraj Sinai Kakodkar
varadrajsinaikakodkar@gmail.com

Disclaimer: *The information provided here is very abstract. Please reach out to me for source code or to discuss more.

## The Big Picture

This module is one of the components in the HFT system level block diagram shown below.




The idea is to develop each of the modules individually and eventually integrate into a system targeting an FPGA application.

- **ethernet front-end(s):** the input to the system works over the UDP protocol receiving the broadcast itch 5.0 packets over MoldUDP64. The output side is based on the TCP/IP protocol stack that is able to talk the OUCH protocol to communicate with the exchange about buy/sell orders generated by the logic.
  I was hoping to piggyback on Xilinx ethernet IPs in order to simplify the problem at hand and receive the ethernet data over the axi-stream but realized that all Xilinx Gbps frontend IPs require the TEMAC IP license for synthesis. And finding a suitable FPGA with ethernet GTx transceiver connected to PL which supports a rmii interface for the free AXI Ethernet Lite IP is something I saved for later.
  For simulation purposes, the ethernet frontend will be replaced by an AXI stream traffic generator.

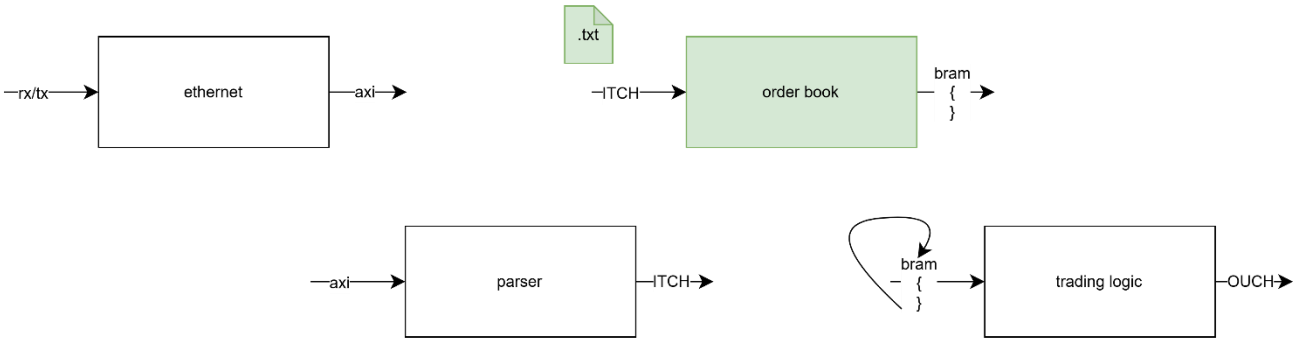
- **parser:** This module acts as a filter + decoder which accepts data over axi and outputs it over a custom interface which mimics the ITCH 5.0. The python notebook in the git repo does exactly this function.

- **order book:** This module reads the data output from the parser and is capable of tracking multiple tickers by building an order book based off the buy/sell orders. The module is capable of accepting 1 order per cycle and is pipelined so as to not drop/miss any incoming requests.

- **trading logic:** I don't know what or how about this at the moment but this is a module I would be touching last after I have enough theoretical knowledge. Currently just thinking of

implementing a simple L2 scalping strategy. Long term goal is to use a microblaze/zynq core to scan the order book and output buy/sell signals based off strategy in C.

In order to break down the project into phases the following diagram shows the roadmap




*The order book does not sort the orders based on B/S and the order price and hence the trading logic needs to scan the entire order book to find the market big and ask price.

## Introduction

This project contains a verilog implementation of an order book based of Nasdaq TotalView-ITCH 5.0 protocol (NQTVITCHspecification.pdf (nasdaqtrader.com))

The implementation supports the bare minimum opcodes "S","R","A","F","E","C","X","D","U" from the ITCH spec needed to build the order book. For simplicity the test vector data is derived from the 08302019.NASDAQ_ITCH50.gz file from Nasdaq archives for sample data and contains 10,000 orders belong to the above opcodes.

Additionally for demonstration I have filtered out the data pertaining to only the following 12 tickers as this is a hardware implementation constrained by the BRAM resources.

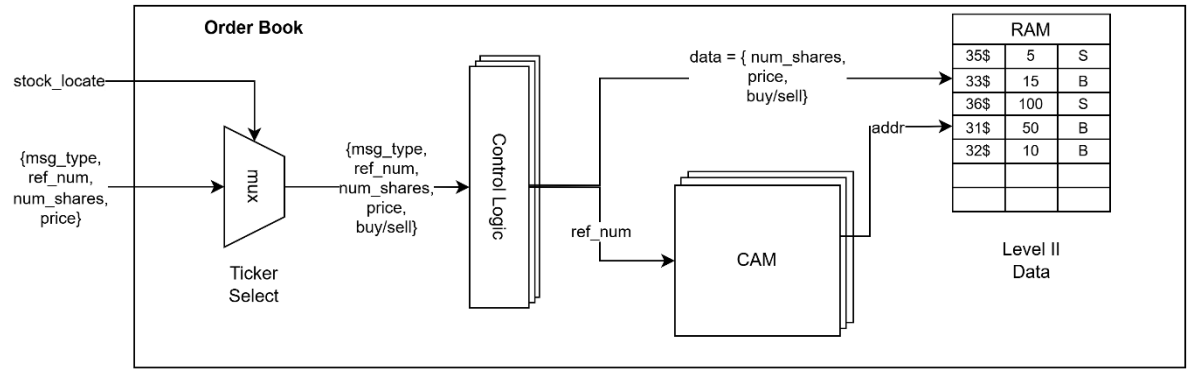
{“AAPL", "MSFT", "NVDA", "GOOG", "AMZN", "FB", "NFLX", "AMD", "INTC", "COST", "JPM", "TSLA"}

The design is parameterized to track from either 0 to all of the 12 tickers, although I am not quite sure how the resource utilization would blow up in case all tickers are set as active!!!

Also, the depth of the order book is parameterized to be any power of 2. I see that for the test sample I used that during heavy volume, the depth does go to 500+ pending orders so I think 1024 is a safe number to start with.

## Microarchitecture

Here is the 20,000 ft view of the order book module.




For simplicity only the datapath is shown and the control signals are not shown. The control logic has the information about “S” type messages and keeps a track of start of day, start of market hours etc.

Also, the “R” type messages are decoded in control logic to match any of the above mentioned 12 tickers and accordingly the stock locate is assigned for further decoding for the day.

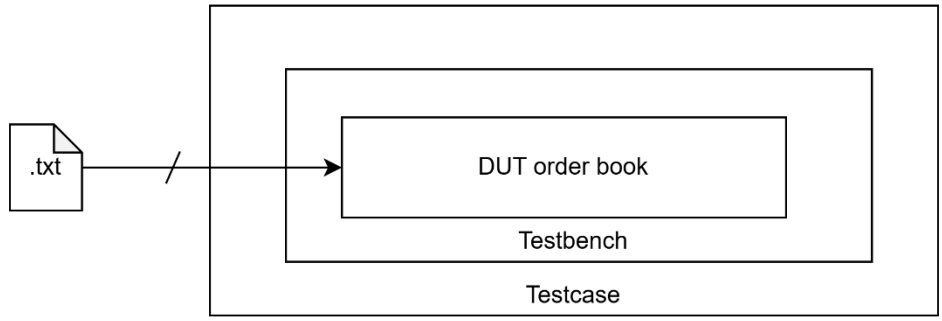
The order book needs to perform a 64 bit search and lookup every cycle in order to execute, cancel, modify etc orders. The design implements a Content Addressable Memory (CAM) which provides a unique address depending on the depth of the order book.

This address is used to locate, store or modify the level 2 market data depending on the order type.

## Testing

The collab notebook added to the git unzips(08302019.NASDAQ_ITCH50.gz) and creates the necessary .txt file which is human readable file needed to make the debug easier. This .txt file serves as the driver input for the testcase which drives the data into the DUT which 1 request per cycle.

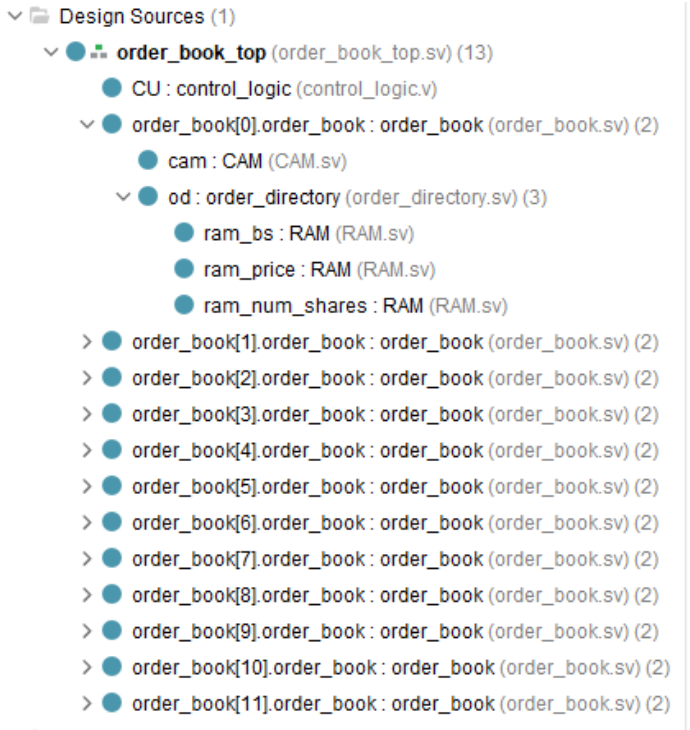
This scenario is stressful enough such that it drives 1 request per cycle which might be close enough to mimic the real system where the fronted working over UDP protocol.




The unit level verification of each of the sub modules is limited to eye-balling at this moment.

Since the test vector is extremely large, the long term idea after the other sub modules are ready is to construct a C/Python based model to maintain and verify the state of the Verilog order book periodically for individual rows for buy/sell, price and stock quantity fields in the RAM.

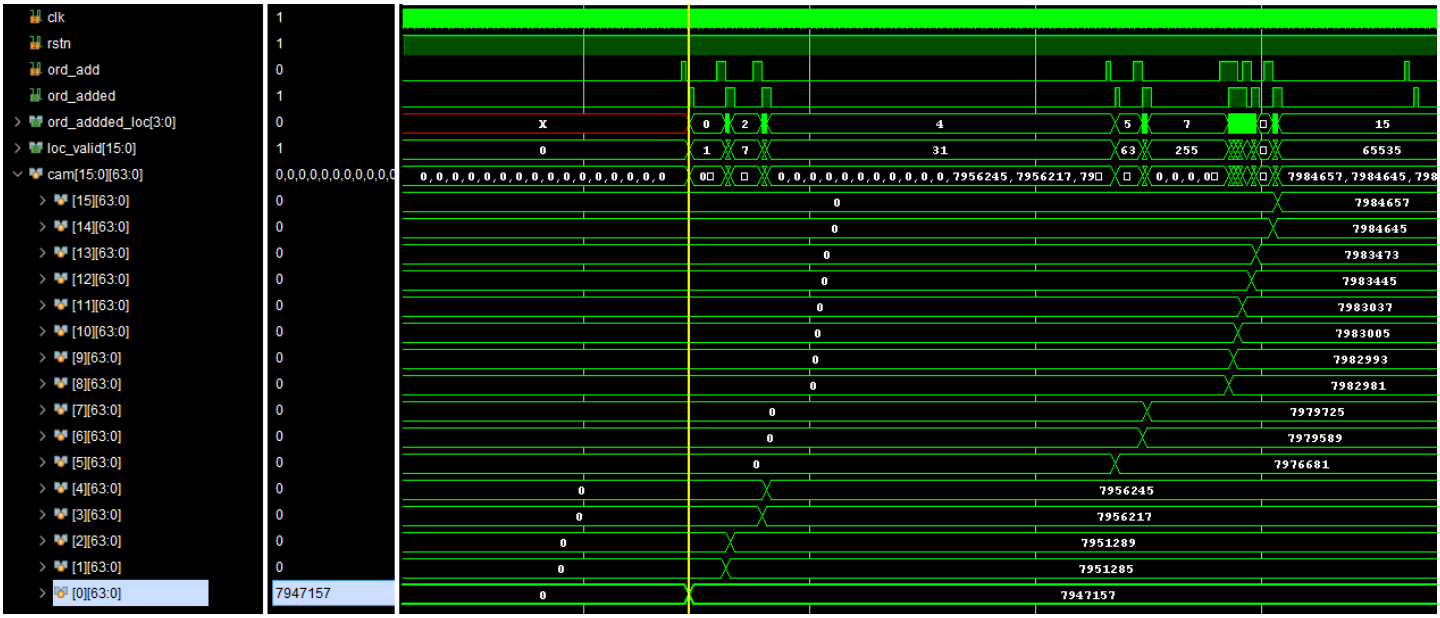
Hierarchically, the source code is organized as follows when enabled for each of the 12 tickers.




The systemverilog test provides a task to scan the text file and send the data to the DUT to mimic the output of the parser.

Eg. The order book looks for the “S” start of day message and waits for the “R” type message to stores the unique stock locate for the desired ticker. After the “S” start of market hours the order book starts accepting orders of the type "A","F","E","C","X","D","U". This sequence is verified by running the testcase with the file nsdaq_itch50_08302019_0.txt.

Following is the CAM contents when for “AAPL” for the first few orders added after the markets opened on 08/30/2019. The order book depth is limited to 16 entries for the sake of easy visibility.




```
1 S 0 0 11046036981912 O
2 R 14 0 11316046568183 AAPL
3 S 0 0 14400000191772 S
4 S 0 0 34200000081775 Q
5 U 14 0 34200009297023 7726541 7947157 100 2116000
6 A 14 0 34200013883023 7951285 S 100 AAPL 2122700
7 A 14 0 34200013885453 7951289 B 100 AAPL 2078400
8 F 14 0 34200019027721 7956217 B 100 AAPL 1764300 CDRG
9 F 14 0 34200019050025 7956245 S 100 AAPL 2437900 CDRG
10 A 14 0 34200055132751 7976681 B 100 AAPL 2100300
11 F 14 0 34200061618043 7979589 B 100 AAPL 1974300 SGAS
12 F 14 0 34200061820505 7979725 S 100 AAPL 2227800 SGAS
13 F 14 0 34200067932909 7982981 B 100 AAPL 1995300 CTDL
14 F 14 0 34200068009257 7982993 S 100 AAPL 2206700 CTDL
15 A 14 0 34200068016961 7983005 B 100 AAPL 2058300
16 A 14 0 34200068041684 7983037 S 100 AAPL 2143700
17 A 14 0 34200068630866 7983445 B 100 AAPL 2058300
18 A 14 0 34200068645528 7983473 S 100 AAPL 2143700
19 F 14 0 34200070637798 7984645 B 100 AAPL 1995300 CDRG
20 F 14 0 34200070644851 7984657 S 100 AAPL 2206700 CDRG
21 A 14 0 34200078602152 7988217 S 45 AAPL 2103300
22 F 14 0 34200085905205 7991213 B 100 AAPL 1974300 SGAS
23 F 14 0 34200085923228 7991241 S 100 AAPL 2227700 SGAS
24 U 14 0 34200097457220 7928225 7995737 100 2164800
25 A 14 0 34200099554996 7996953 S 50 AAPL 2200000
26 A 14 0 34200123054581 8008789 S 48 AAPL 2102000
27 A 14 0 34200123090043 8008821 S 100 AAPL 2123000
28 A 14 0 34200123095230 8008825 B 100 AAPL 2079300
```

As can be seen from the waveforms and the following data from Nasdaq (filtered for “AAPL” from the .gz file), we can see all the add type orders being added successfully to the order book.

Additionally, from the same test input, each of the "A","F","E","C","X","D","U" were verified to be working correctly by adding and removing the orders from the order book.

The RAM contents were similarly verified for holding the correct price, volume and buy/sell information for each of the associated unique order numbers.

## Future work

- Additional stress testing to be conducted after the parser and ethernet frontend is completed