HFT Accelerator

Terminology:

Stock: a virtual object which someone can sell or buy.

Order: a request to buy or sell a stock.

Bids: a request to buy a stock at a particular price

Ask: a request to sell a stock at a particular price

Exchange: a third party entity that matches groups interested in buying / selling stocks. Sends market updates to market participants over network.

Automated trading: a computer strategy that connects to an exchange and submits orders with the objective of making money.

HFT: High frequency trading, a subset of automated trading where the objective is to react quickly to changes in the market, and submit orders with ultra low latency.

Objective:

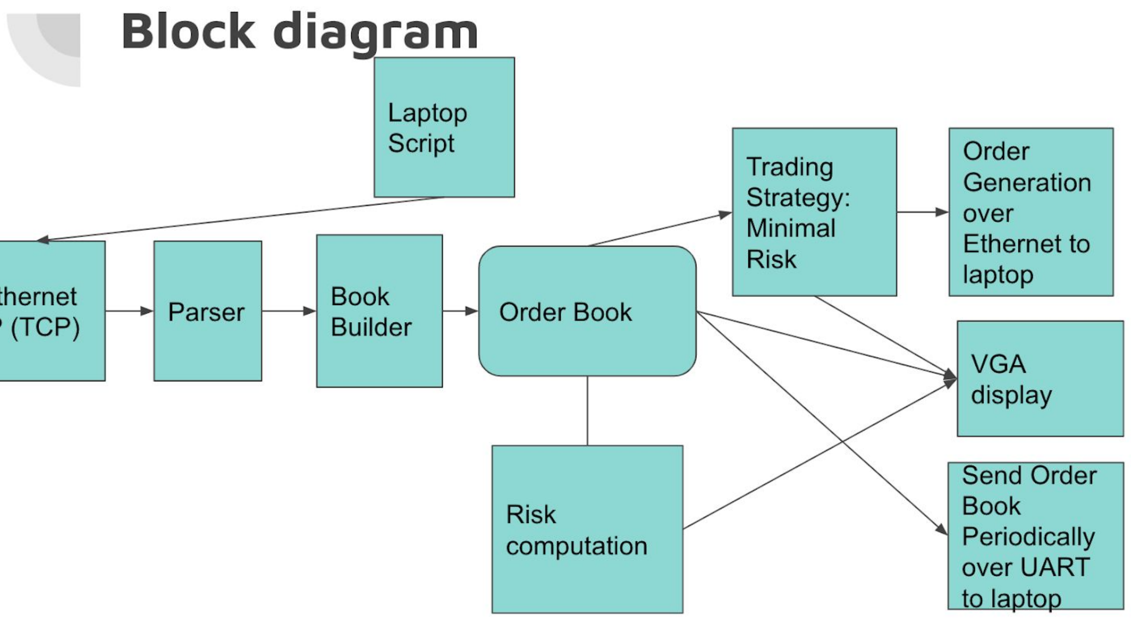
HFT or High frequency trading, a subset of automated trading where the objective is to react quickly to changes in the market, and submit trades. The flow of information is as follows: the exchange sends market data through ethernet (describing the current price of the stock, who is interested in buying and selling). The HFT engine is responsible for receiving this data, parsing it with the protocol provided by the exchange, updating is internal state about the market, and submitting orders back to the exchange in reaction to market updates.
In summary:
HFT firms and market makers need ultra-low latency solutions to quickly:

1) filter the market datastream from the exchange.

2) update their knowledge of the market to keep track of best prices for stocks (building an order book).

3) Submit trades to the exchange based on the information.

For our 6.111 project, we are interested in building an HFT (High Frequency Trading) accelerator in FPGA.


Block diagram
Laptop Script
Ethernet IP (TCP)
Parser
Book Builder
Order Book
Trading Strategy: Minimal Risk
Order Generation over Ethernet to laptop
VGA display
Send Order Book Periodically over UART to laptop
Risk computation

Block Diagram for HFT Accelerator

We will connect the FPGA with our laptop over ethernet, and have a script that sends market data using the exchange protocol over TCP (to simulate the exchange). The FPGA will have an ethernet stack (TCP), which will receive the market data from the laptop. It will forward to the parser module, which is responsible for implementing the exchange protocol and parsing what orders were submitted / canceled, or what trades happened on the market.

The parser module will forward the parsed information into the order book module, which is used to represent the current state of the market (outstanding bids, and asks).

The order book module will forward the best price for each stock to the trading module. It will periodically send a snapshot of the order book to a laptop so that we can display it. The trading module will do calculations based on the best price of the stocks at this time step, as well as in the previous time step to decide what orders to submit to the market.

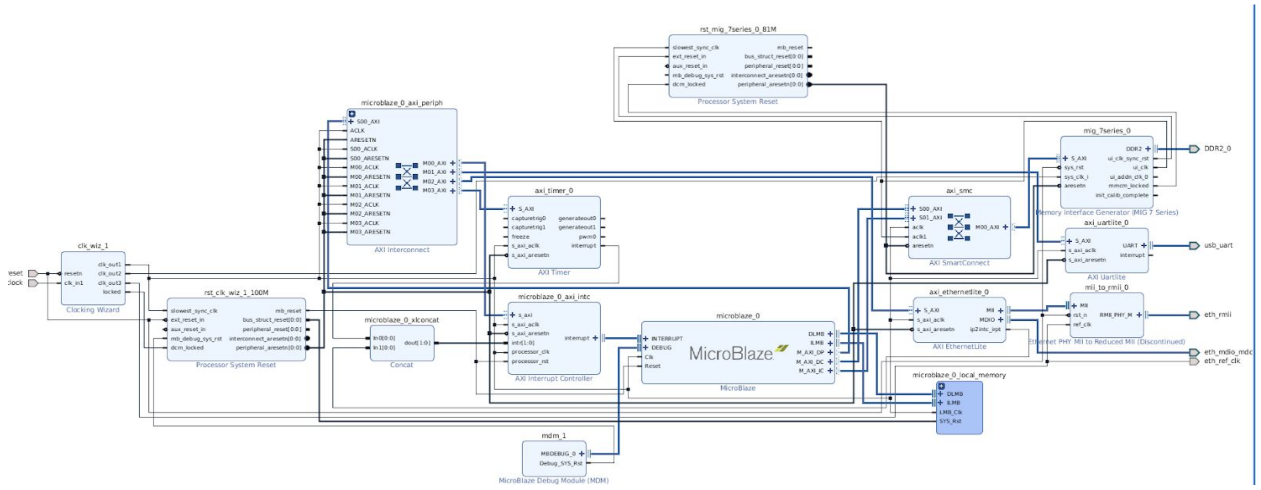
The trading module outputs orders over ethernet to the laptop, and the laptop displays the orders from the FPGA.

In more detail, here are the different components:

Ethernet IP Stack:

The IP stack is based heavily on Vivado's AXI IPs and Microblaze IP. The FPGA that we have have a PHY (physical chip for Ethernet). We plan to use AXI ethernet lite to interface with the physical layer Ethernet protocol and the Microblaze to handle the higher level TCP/IP protocol. The Microblaze and the Ethernet lite IP will interface through the AXI protocol. The relevant packets then would appear in the Microblaze's local memory. We will need to figure out a way to interface with the Microblaze (probably have to be through AXI) and get its memory (job of parsing stack). Not sure what throughput we can ultimately end up with this way.

There is much uncertainty surrounding whether or not we can get the stack to work even with heavy usage of xilinx ips.



Ethernet IP Stack

Parsing:

On a high level, the Parser is posed to communicate with the Mircoblaze softcore processor, get and process the structured data being sent to it in a wise manner. It is posed to receive structured data in the format specified by [1] and extract relevant information that can be used to construct the order book. The structured data take the form of a string of Bytes in Big endian format that are structured per specification. For initial implementation, We will be receiving structured data and sending parsed information back through UART. Then We would implement the AXI protocol needed to communicate with the Mircoblaze IP and retrieve structured data from its internal memory.

An example string of inputs is the following.

Length in Bytes	Type	Value	Meaning
1	Message	8’hA	Add order
4	Timestamp	32’h0300	Time that order happened
4	Order number	32’h03BA	Unique value to distinguish order
1/8	Buy or sell	1’b1	A Buy order
4	Shares	32’h01BB	The total number of shares
8	Stock Symbol	64”h0AAB_2341	Which stock the order concerns
4	Price	32’hBABB	The price offered to buy

1. http://nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/NQTV-ITCH-V4_1.pdf

Trading strategy:

The high level idea is to keep track of an estimate of the covariance matrix between the one-minute normalized returns of all our securities. Then we will construct a min-risk portfolio by inverting the covariance matrix. We can then generate orders to update our positions to hit the target portfolio.

The trading logic module is composed of various submodules as indicated in the diagram. We will assume that the order book builder can provide us with the latest bid/ask price vector of all the securities we are tracking and a delayed bid/ask price, perhaps from 1 minute ago. (Note that we do not use any other information of the order book to decouple our modules for ease of testing) From those two price vectors, we can calculate the normalized return for each security using fixed point division. This will give us an observation of the latest normalized return vector for the securities we are tracking, which we will call *v*.

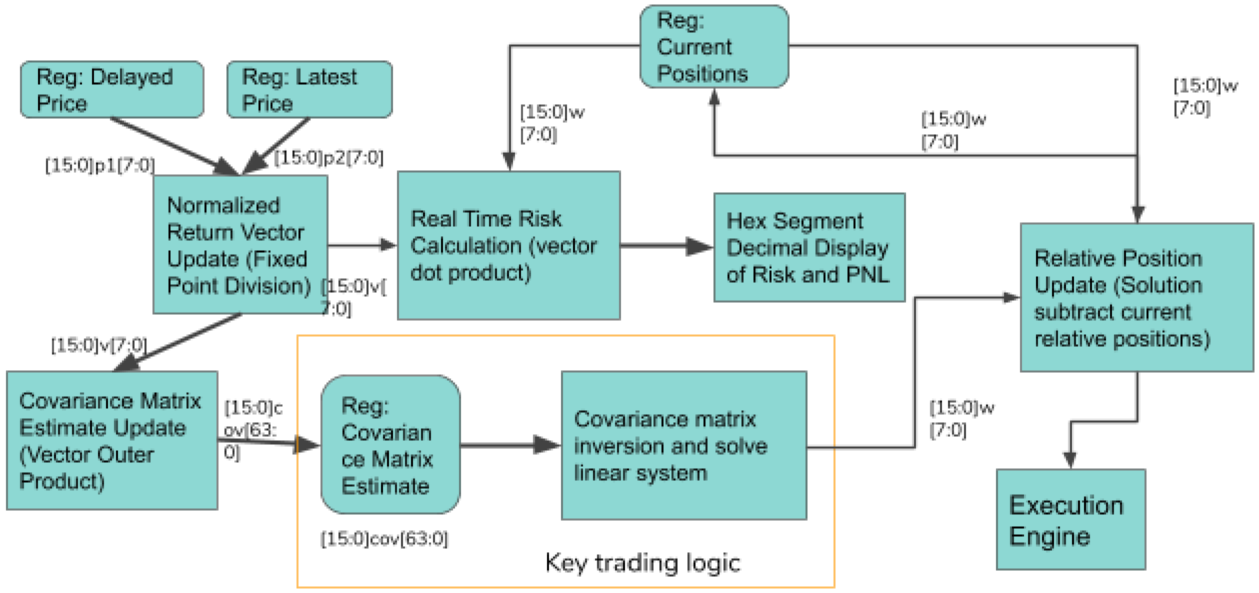
From *v*, we can update our covariance matrix Σ. If the current covariance matrix is obtained from n observations, then we simply multiply it by n, add to it the outer product of our new observation, then divide the whole thing by n+1. The formula is this: $(n\Sigma + vv^T) \div (n+1)$.

Now the calculation of the risk is just $w^T \Sigma w$ where *w* is the current relative positions. We don’t have to do the matrix vector product however, and do something smart. We only have to incrementally update this value, which reduces to adding the contribution of the new observation, which is $w^T vv^T w = 2(v^T w)$. This means we just have to do a dot product, greatly reducing the latency for risk computations. This will be realized through an adder tree. The updated risk will be displayed in decimal on the hex display. The latency here is on the order of tens of cycles (i.e. nanoseconds).

This covariance matrix will be used in the next step, which is construction of the minimum risk portfolio. This amounts to calculating the following quantity: $x = \Sigma^{-1}1$, and then normalizing it. In effect, we have to solve a linear system with the covariance matrix as the coefficients of the unknowns. In linear algebra, it is typical to first perform QR decomposition and then solve the resulting upper triangular system for better numerical stability.

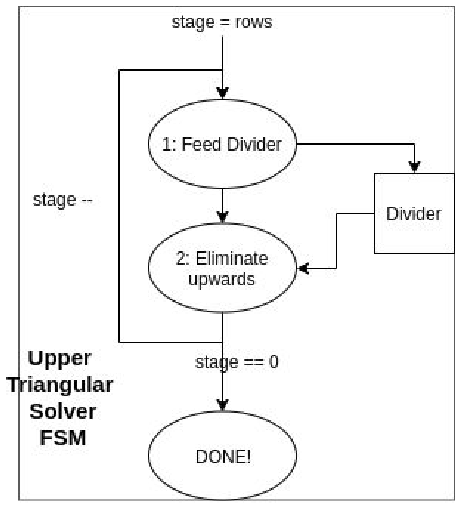
There are typically three ways to perform QR decomp. Householder reflections and gram-schmidt are well suited for cases where we have access to many parallel multipliers, which is not the case on the Artix-7. We will perform the QR decomposition using the Givens rotation method using Cordic processors. [2]

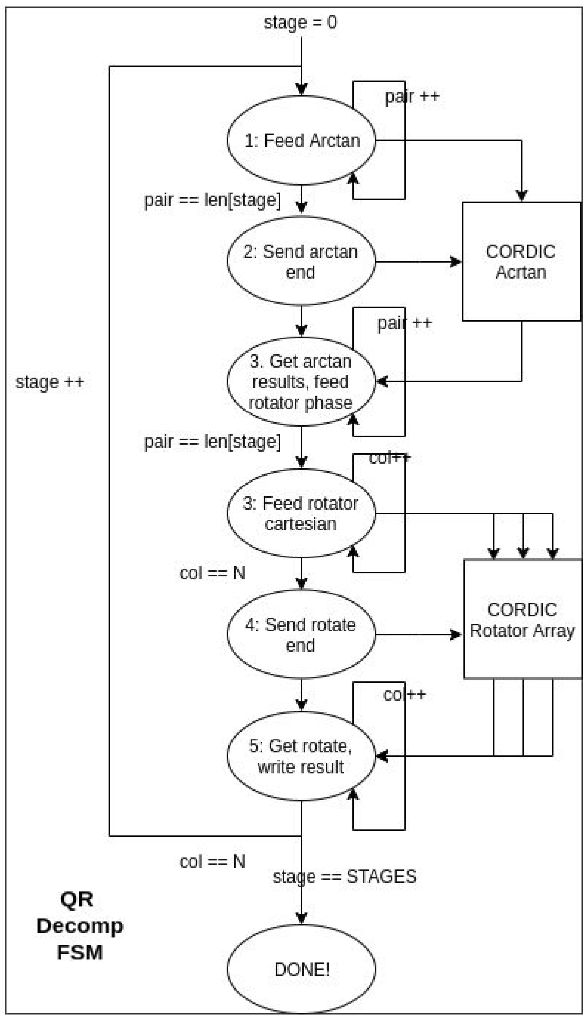
We present the overall system diagram here:

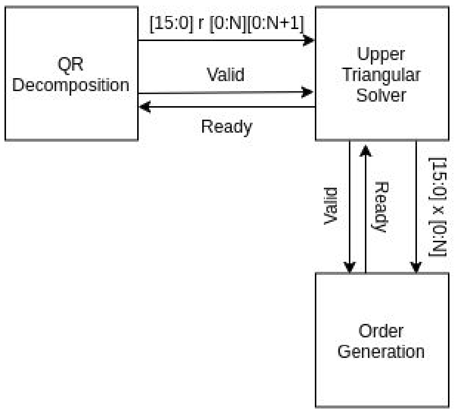


Reg: Current Positions
Reg: Delayed Price
Reg: Latest Price
[15:0]w[7:0]
[15:0]w[7:0]
[15:0]w[7:0]
[15:0]p1[7:0]
[15:0]p2[7:0]
Normalized Return Vector Update (Fixed Point Division)
Real Time Risk Calculation (vector dot product)
Hex Segment Decimal Display of Risk and PNL
Relative Position Update (Solution subtract current relative positions)
[15:0]v[7:0]
[15:0]v[7:0]
Covariance Matrix Estimate Update (Vector Outer Product)
[15:0]cov[63:0]
Reg: Covariance Matrix Estimate
Covariance matrix inversion and solve linear system
[15:0]w[7:0]
[15:0]cov[63:0]
Key trading logic
Execution Engine

We also present the FSM diagram for the key trading logic in the following figure. The QR module does the Givens rotation in stages, where in each stage some columns are rotated in parallel. Afterwards, the result is fed to an upper triangular solver module. We implement a folded design where each stage reuse the same arctan and rotate CORDIC modules, since the Artix-7 does not have that much resources on the chip. **These modules have been implemented and work in simulation for a 4 by 4 linear system.**







1. Cordic citation: https://web.njit.edu/~akansu/PAPERS/Torun-Yilmaz-Akansu-FPGAPortfolioRisk-ICASSP2013.pdf
2. https://ieeexplore.ieee.org/document/7110554

Book Building:

One of the most important aspects of a trading system is having an internal succinct representation of what is happening on the market: this data structure is typically referred to as an Order Book. The Order Book captures the current range of price for each stock including what the “price ladder” is. There is two sides in an order book: “ask” and “bid”. An ask is a request to sell a stock at a particular price, and a “bid” is a request to buy a stock. An order book maintains both of these information per stock. Each side is sorted by price, and for each price, there is a list of orders with that price.
This reflects that people want to trade with the best price on the market, while giving priority

within the same price to orders that arrived earlier.

An order book is represented as a binary search tree on the price, where each price has a linked list internally sorted by the time of arrival. There is also the concept of “depth” of a book which represents how many distinct prices its keeping track of. This data structure is highly non trivial to implement efficiently on FPGA because of the sequential nature of the order book, and the algorithm complexity of implementing tree rotations on FPGA in a low latency manner.

Each order is uniquely associated with an order_id assigned by the exchange, and there is a “ticker” which determines the offsets between the different prices. For example a ticker of “0.1” means the price ladders goes from 90 to 90.1 to 90.2 and so on. The state of the order book will be stored in BRAM.

Supported Operations (Input / Output):

The order book supports three operations:

AddOrder:

```
module addOrder #(parameter PRICE_WIDTH=15;
                  parameter ID_WIDTH=15;
                  parameter QUANT_WIDTH=7;
                  parameter STOCK_WIDTH=7;)
   (
    input                   clk_in,
    input                   enable_in,
    input [STOCK_WIDTH, 0] stock_symbol,
    input [ID_WIDTH, 0]    order_id,
    input [PRICE_WIDTH, 0] price,
    input [QUANT_WIDTH, 0] quantity,

    output                  ready_out
      );
    endmodule
```

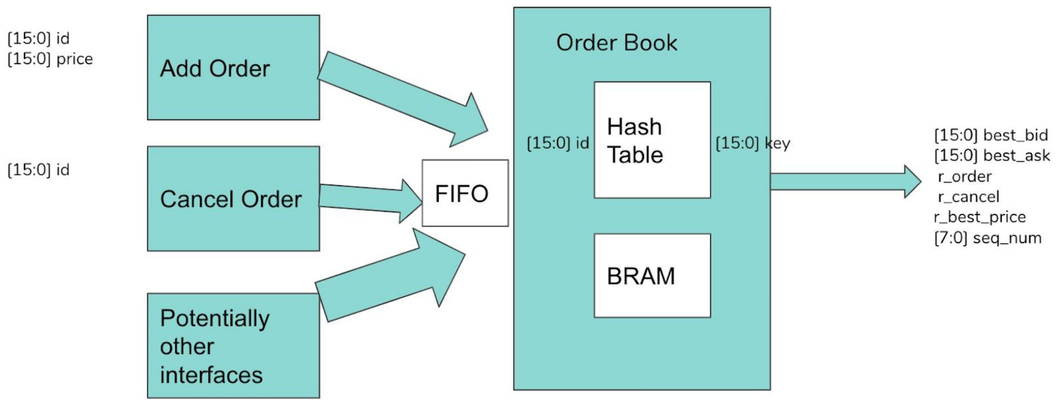
CancelOrder:

```
 module cancelOrder #(parameter ID_WIDTH=15; )
    (
     input                clk_in,
     input                enable_in,
     input [ID_WIDTH, 0] order_id,

     output               ready_out
       );
     endmodule
```

getBestPrice:

```
module getBestPrice #(parameter PRICE_WIDTH=15;
                  parameter STOCK_WIDTH=7;)
   (
    input                       clk_in,
    input                       enable_in,
    input [STOCK_WIDTH, 0]  stock_symbol,

    output                      ready_out,
    output [PRICE_WIDTH, 0] best_sell,
    output [PRICE_WIDTH, 0] best_buy
    output                      ready_output
      );
    endmodule
```

Block Diagram for single symbol



Block diagram for Order Book

Targets:

Basic:
Keeps track of best price and aggregate quantity at that price, per stock (only one price ladder). This information is stored in an array indexed by stock. Can calculate risk in real time.

Target:
Fully functional order book with multiple levels, and support for multiple tickers, and some basic performance optimizations to reduce latency. No support for concurrent modifications. Can calculate risk in real time and do some form of eigendecomposition.

Stretch:
Massive reduction of latency in accessing and modifying the order book by taking advantage of the parallelism of the FPGA.
We will be following “Exploring the Potential of Reconfigurable Platforms for Order Book Update” https://www.doc.ic.ac.uk/~wl/papers/17/fpl17ch.pdf which describes a highly optimized Order Book.
Fully functional trading strategy and 200 ns risk computation.

Challenges:

Order Book:

The orderbook will likely use up a lot of resources on the FPGA because the depth on exchanges is large, and implementing a fast order book requires a lot of physical resources. All the functions declared share the same underlying BRAM, which is used to store the actual order book and it will be difficult to synchronize the usage of this structure while also being fast. Its likely that the order book update has to be deeply pipelined and making sure its doesn't bottleneck our system will be crucial.

Trading Strategy Challenges:
The key technical challenge here is 1) how to efficiently implement the risk update, (which needs to execute in less than ~20 cycles assuming 100 MHz clock to be competitive with state of the art implementations). The operations involved here are relatively simple fortunately, just involving division, vector dot product and multiplication. This provides interesting exercise in lower level performance engineering. (e.g. where/if you pipeline etc.) 2) Implementing the systolic array of CORDIC processors to perform the eigen-decomposition. The operations involved here are quite complex, but the timing constraints are not as dire. (We do not want to trade excessively in fear of commission costs etc.) This provides good practice in designing complex systems and minimizing area usage.

Combination
Each part will have different demands on compute resources. Will be hard to balance the needs of order book vs trading strategy etc.

Timeline:

	Week 1 (Nov 4)	**Week 2 (Nov 11)**	**Week 3 (Nov 18)**	**Week 4 (Nov 25)**	**Week 5 (Dec 2)**
Ethernet/Parser	detailed FSM diagram of how the parser works, have testbench over	implement parser module with test bench	debug parser module and start performance optimization	Explicit Integration.	Look into stretch goals, and performance optimizations.
Book building	implement order book software implementation, complete high level microarchitecture design with resource estimates.	implement first-pass verilog module with test bench	debug order book verilog implementation, start looking at performance optimization	Explicit Integration.	Look into stretch goals, and performance optimizations.

Trading logic	Implement matrix inversion module and test bench in verilog to get resource estimate.	functionally debug matrix inversion module and get updated resource estimates	implement the rest of the risk computations, trade updates, etc.	Explicit Integration.	Look into stretch goals, and performance optimizations.