An HFT (High Frequency Trading) Accelerator

Endrias, Tony, Natnael

Terminology

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

Order: a request to buy or sell a stock.

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.

Overview

- 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.

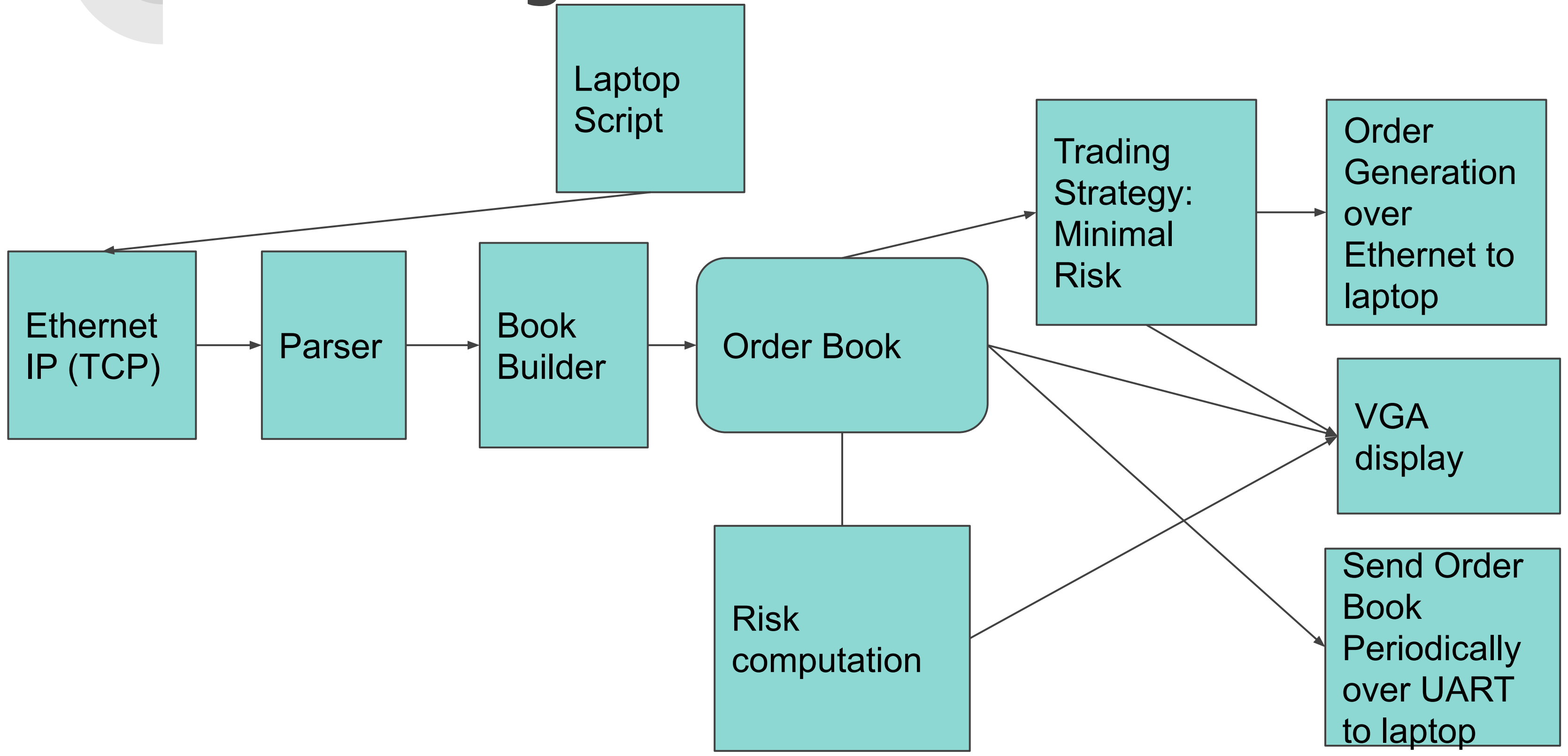
Our Objective

- HFT companies are currently moving in the direction of using FPGAs to replace their current software system.
- 3rd party companies offer latencies of around 300 ns for each of step 1 and step 2. 3 depends on the complexity of the trading strategy.
- We will build a FPGA implementation of HFT for low latency and high throughput and see how close we can get to state of the art taking into account our constrained resources.

Planning / Setup

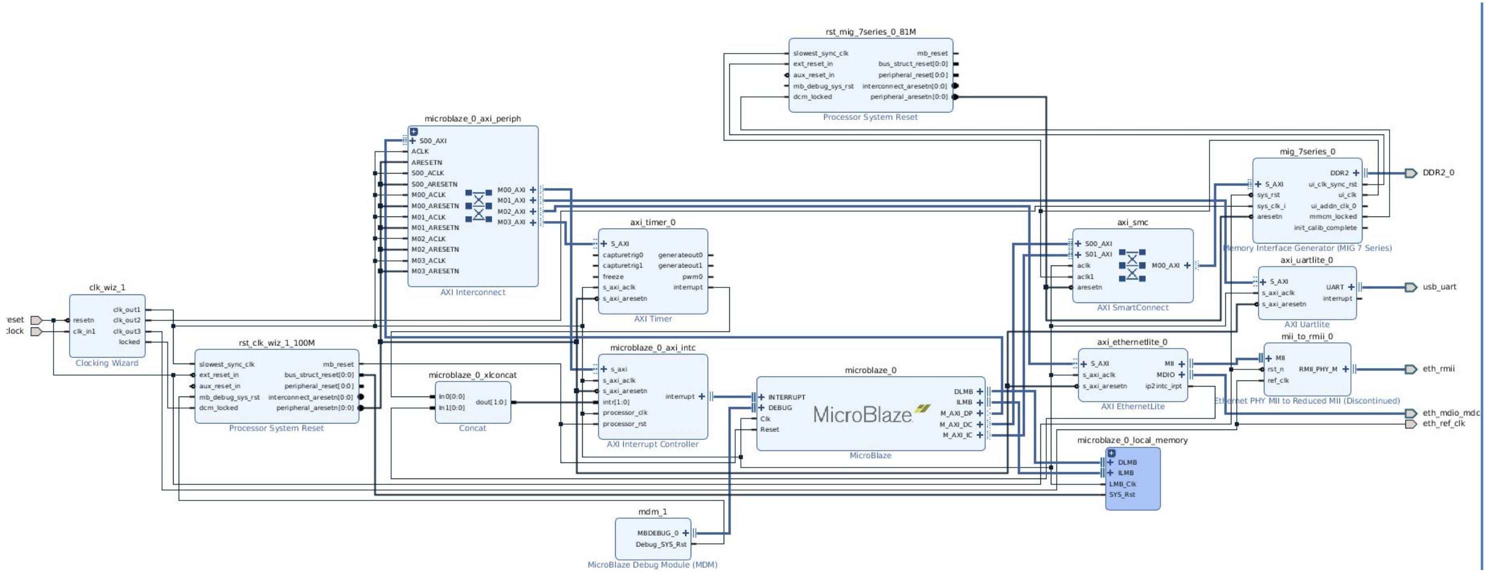
- We will connect the FPGA with our laptop over ethernet, and have a script that sends market data using the exchange protocol over TCP.
- The FPGA will use the ethernet IP stack to process the packets send from our laptop.
- It will output the orders it wants to put on the market, and will send the state of the order book periodically over UART to our laptop.
- For our basic implementation, we will use UART instead of ethernet / TCP.

Block diagram


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

Ethernet IP

- Microblaze IP provided by Xilinx.
- Provides soft core processor to interface over AXI protocol.
- Implements tcp/ip stack and netty gritty details of network communication over ethernet.



Parser

- Parses Nasdaq ITCH protocol based messages. Messages detail the time evolution of the state of the exchange.
- Messages consumed through communicating with Microblaze IP over AXI protocol.
- Passes relevant information to book builder module.

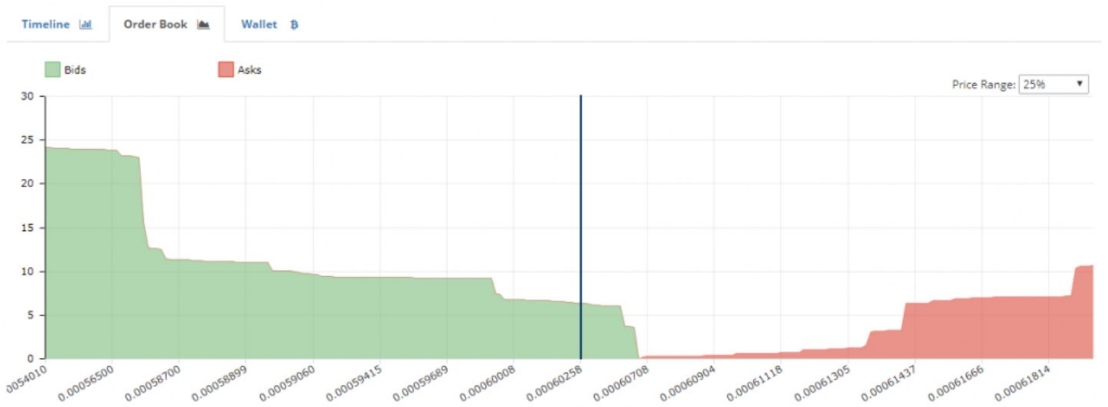
Length in Bytes	Type	Value	Meaning
1	Message	8'hA	Add order
4	Timestamp	32'h0300	Time that order happened
4	Order number	32'h03BA	Unique value to distinguish order
1/8	Buy or sell	1'b1	A Buy order
4	Shares	32'h01BB	The total number of shares
8	Stock Symbol	64"h0AAB_2341	Which stock the order concerns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

```
module parser #(parameter PRICE_WIDTH=15;
                parameter ID_WIDTH=15;
                parameter QUANT_WIDTH=7;
                parameter STOCK_WIDTH=7;)
   (
    input                       clk_in,
    input [7,0]                 data_in,
    input                       enable_in,

    output [2,0]                operation,
    output [STOCK_WIDTH, 0]     stock_symbol,
    output [ID_WIDTH, 0]        order_id,
    output [PRICE_WIDTH, 0]     price,
    output [QUANT_WIDTH, 0]     quantity,
    output                      ready_out
    );
  endmodule
```

Order Book Building

- Want FPGA to maintain the current state of the market -- a list of best prices for buying (bid) / selling (asks) for multiple stocks.
- Maintains price ladders (from lowest to highest price for the stock).
- Receives updates from parser about new orders, and cancels.
- Needs to be low latency and have low space storage (<500 kb) to fit in the constraints of the nexus FPGA.



Operations

```
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
```

Operations

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

    output                  ready_out
      );
    endmodule
```

Operations

```
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
```

Implementation

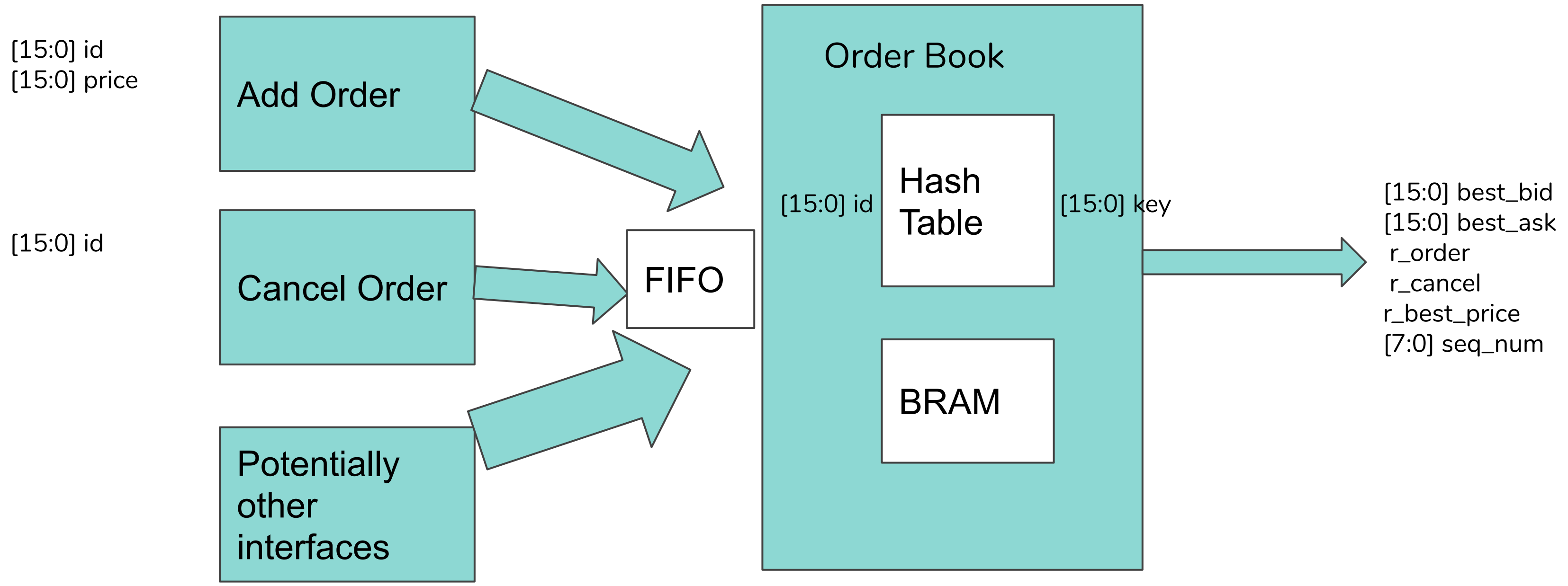
Code will be parametrized by number of stocks, and price ladders.

An array to represent the different price levels, and hashtable of all the orders to support quick removal from a level given an order id.

The current state of the order book will stored in BRAM.

Output best buy / sell price for each stock to the trading Module.

Block Diagram for single symbol



Challenges

Implementing a fast hash table and queue in FPGA -- will have to be pipelined as it will require multiple memory reads. Possibilities: cuckoo hashing, quadratic probing.

Minimizing space utilization -- the exchange sends 8 bytes for order id and price, and we have to map that to an 16 bit space so that we can fit it on FPGA.

(The order book max size is 2^16 * 32 bits which is 262 kb, we will try increasing this depending on resource usage).

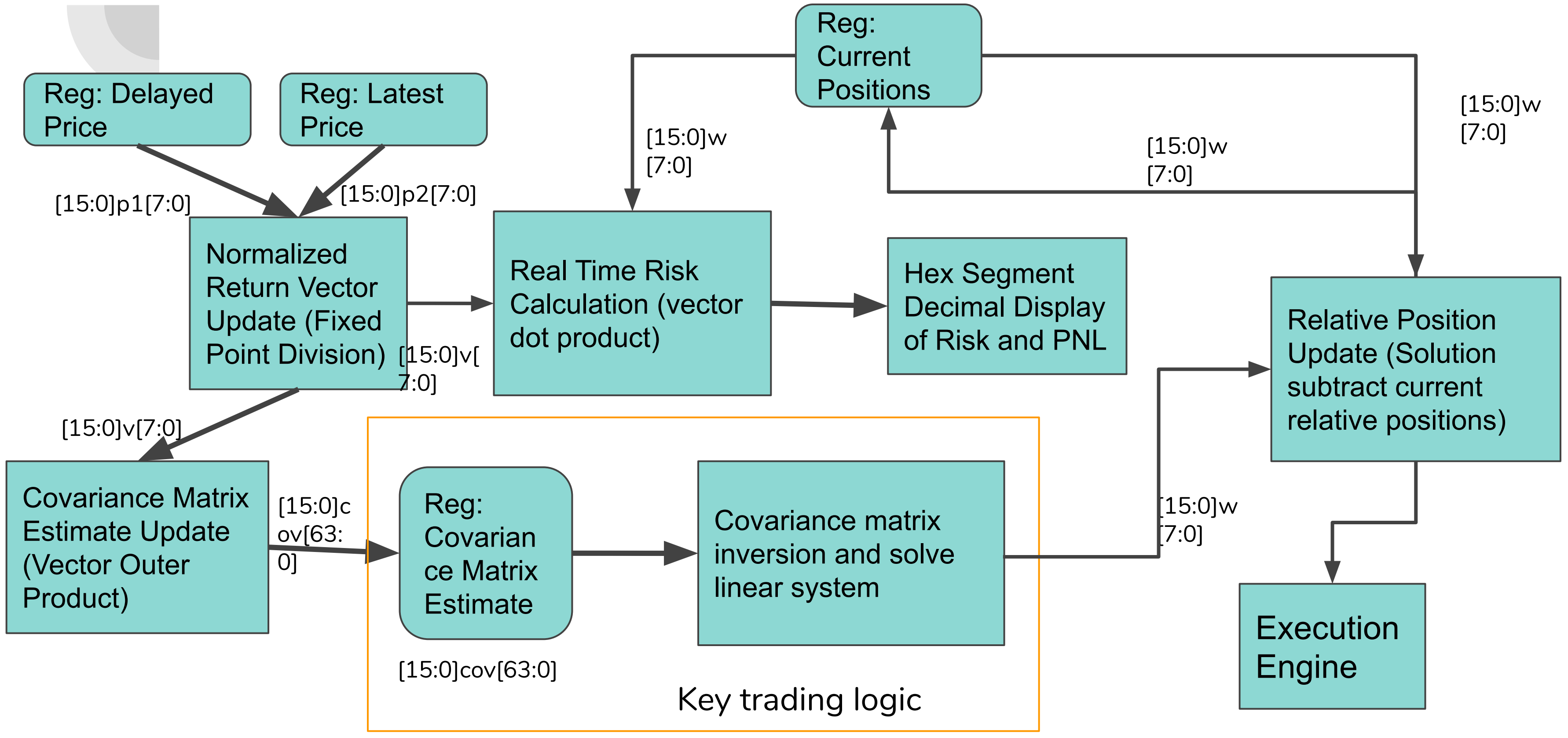
Multiple companies specialize in making a fast order book, so a lot of room to optimize latency.

Trading Logic

- Want FPGA to quickly update risk (for compliance and other reasons). Latency sensitive: w∑w, which is effectively give new observation v: 2(v^Tw).

- Might want FPGA to quickly rebalance the portfolio to minimize risk. Less latency sensitive, but still preferably fast. **Cannot take up too much area.**

Trading Logic Block Diagram


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

Inverting an 8 by 8 matrix

1							
2	3						
1	2	7					
3	5	6	8				
1	4	5	7	9			
2	3	4	6	8	10		
1	2	3	5	7	9	11	

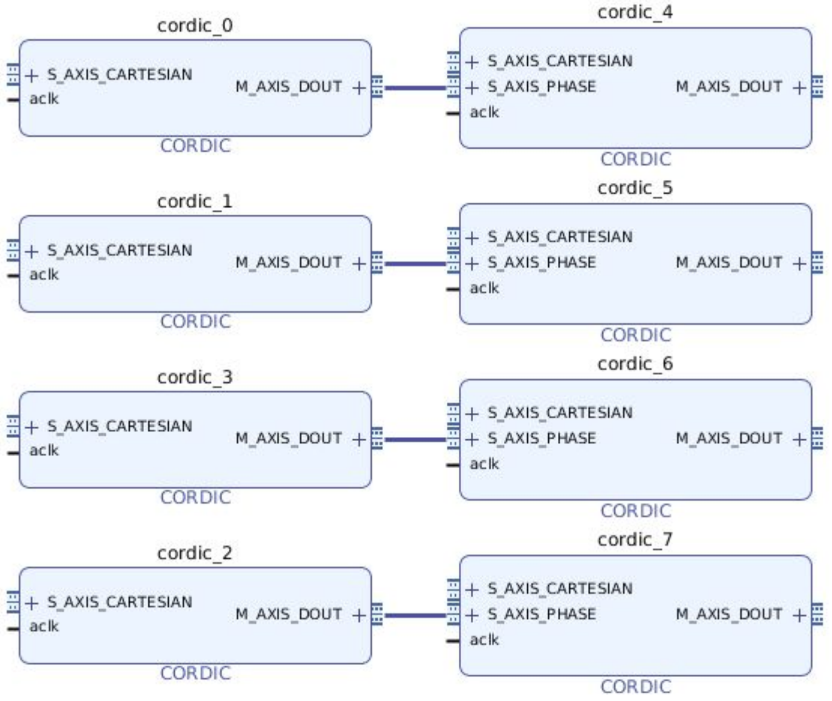
- We would like to obtain inv(**Σ**)**1**
- This is equivalent to inv(**R**)**1**, where **R** as in **Σ=QR**. In other words, we would like to do fast QR decomposition.
- We will use Givens rotation method, which minimizes DSP usage (given the Artix only has 240 DSPs)
- This method zeros out an entry of the matrix at a time to produce an upper triangular matrix (**R**), by rotating two rows in conjunction.
- We can parallelize parts of it by picking disjoint two rows, as indicated in the table, where each number corresponds to a stage of the rotation process.

Cordic for matrix inversion

8 by 8 register array to store the updated covariance matrix

FSM Controller (basically a programmable 64 by 12 16bit crossbar, programs accesses to covariance matrix depending on algorithm stage)

AXI cartesian I/O (12 * 16bits)



Latency estimates

- Fully pipelined CORDIC IP has a latency of 16 cycles for 16 bit input, and a throughput of one output per cycle
- The rotation cannot begin until the arctan is done. This suggests a latency of 16 + 16 + x for each stage, where x is the length of the longest row to be rotated in the stage. ($x < 8$)
- Therefore, it takes about 400 cycles to perform the matrix inversion (assuming no additional pipelining in the control FSM), which suggests a latency of around 4 microseconds.
- Then we will need to solve the resulting linear system, which is upper triangular. This can be done quite simply. The latency is 8 times the latency of the divider.

Timeline

11/12

Tony: implement matrix inversion module and test bench in verilog to get resource estimate

Endrias: implement order book software implementation, complete high level microarchitecture design with resource estimates

Natnael: detailed FSM diagram of how the parser works, have testbench over UART

11/19

Tony: functionally debug matrix inversion module and get updated resource estimates

Endrias: implement first-pass verilog module with test bench

Natnael: implement parser module with test bench

11/26

Tony: implement rest of the risk computations, trade updates, etc.

Endrias: debug order book verilog implementation, start looking at performance optimization

Natnael: debug parser module and start performance optimization

(Implicit Integration)

12/05

Finish performance optimization.