HFT_on_a_Chip

Code Description

2024.3

Table of Contents

FIX, FAST, and HFT

The Financial Information eXchange (FIX) and FAST (FIX Adapted for STreaming) protocols are standards used in the world of finance, particularly in high-frequency trading (HFT), to facilitate communication and data exchange between entities involved in financial transactions.

FIX Protocol:

- The FIX protocol is an industry-standard messaging protocol designed for the real-time electronic exchange of securities transactions. It is widely used by banks, broker-dealers, exchanges, and institutional investors.
- FIX is used for a variety of financial transactions, including order submission, order cancellation, trade execution reports, and market data transmission.
- The protocol defines a set of standardized messages that allow participants to communicate in a clear and structured format. It supports different asset classes, such as equities, fixed income, foreign exchange, and derivatives.

FAST Protocol:

- FAST (FIX Adapted for STreaming) is a protocol designed to optimize the performance of the FIX protocol by reducing the bandwidth required to transmit FIX messages, thereby increasing the speed of message transmission.
- It achieves this by applying compression techniques to the data, allowing the representation of FIX messages in a more compact form without losing any information. This is particularly useful for transmitting large volumes of market data.
- The FAST protocol is commonly used in conjunction with the FIX protocol to enhance the efficiency of data transmission, especially in market environments where low latency is crucial, such as HFT.

FAST helps to reduce the amount of data that needs to be sent over the network, which is vital for HFT strategies that rely on analyzing market data and executing trades in milliseconds or microseconds.

HFT firms use these protocols to receive market data, submit orders, and manage their trading strategies in real-time, enabling them to exploit price differences and market inefficiencies quickly.

Trading Logic

This part defines a trading logic system that uses a combination of technical analysis indicators to make trading decisions. Here's a breakdown of each part of the code:

Initialization of Constants and Variables:

The code begins by defining constants for the periods of the short-term and long-term Simple Moving Averages (SMA) and the Relative Strength Index (RSI). It also defines thresholds for determining overbought and oversold conditions using the RSI. Variables for storing the values of the SMAs, RSI, and a volatility index are initialized.

Update Moving Averages Function (**updateMovingAverages**):

This function updates the short-term and long-term SMAs based on the latest price. It maintains arrays to store recent prices for each SMA period and calculates the average values.

Update RSI Function (**updateRSI**):

This function calculates the RSI, a momentum indicator, using price changes. It distinguishes between gains and losses to calculate the average gains and losses over the RSI period. The RSI is then derived from these average values.

Calculate Volatility Index Function (**calculateVolatilityIndex**):

The volatility index is calculated as the absolute difference between the short-term and long-term SMAs, normalized by the long-term SMA. This index can help gauge the market's volatility.

Create Order Function (**createOrder**):

A utility function to create a new order based on the bid or ask order details and whether it's a buy or sell order.

Trading Logic Function (**trading_logic**):

This is the main function where trading decisions are made. It reads incoming bid and asks orders from streams and updates the SMAs and RSI using the average of the bid and ask prices.

Trading Strategy:

The strategy uses two RSI thresholds to define overbought (above 70) and oversold (below 30) conditions. An asset is considered overbought when the RSI is above 70, which might indicate a potential sell signal. Conversely, an asset is considered oversold when the RSI is below 30, potentially signaling a buying opportunity.

When the short-term SMA is above the long-term SMA, it suggests an upward trend, signaling a potential buying opportunity. Conversely, it might indicate a downward trend, suggesting a potential selling opportunity.

Order Book

The order book manages and ranks orders using a heap data structure for both bid (buy) and ask (sell) orders.

The bid orders are organized in a max heap, where the top of the heap (root node) represents the highest bid price available in the market. Ask orders are organized in a min heap, where the top of the heap represents the lowest ask price in the market.

Both heaps prioritize orders based on price. For bids, higher prices take precedence, while for asks, lower prices are prioritized. When two orders have the same price, the order ID can serve as a tiebreaker.

Utility Functions:

log_base_2: This function calculates the base-2 logarithm of a given index. It's optimized for quick calculations, essential for fast indexing in heap structures.

pow2: Computes the power of 2 for a given level. It's used for determining positions within the binary heap efficiently, leveraging bitwise operations for speed.

find_path: Determines the path in the heap where a new node should be inserted, or an existing node should be removed. It utilizes the current state of the heap to find the optimal position, considering any gaps (holes) left by removed orders.

calculate_index: Calculates the child node's index in the binary heap based on the current node's path and level. This function is crucial for navigating through the heap.

Order Management Functions:

swapOrders: Exchanges the properties of two orders, including price, size, order ID, and direction. This function is vital for maintaining the heap's order during insertions and deletions.

add_bid and add_ask: These functions add new bid and ask orders to the respective heaps. They locate the correct position for the new order and ensure the heap's properties are preserved after insertion.

remove_bid and remove_ask: Responsible for removing orders from the bid or ask heaps. They handle the direct removal of orders and adjust the heap to maintain its structure and order properties.

order_book:

The central function is that orchestrates the order book's operations. It processes incoming orders, timestamps, and metadata, updating the bid and asking heaps accordingly. This function ensures that the top of each heap reflects the current best bid and ask prices and manages the data flow between different streams for synchronization and consistency.

FAST Processor

The FAST processor decodes incoming order, and encodes outgoing packets, by FAST protocols, it contains protocol processor, encoder, and decoder.

Decoder

The **Fast_Decoder::decode_fast_message** function is designed to decode a FAST (FIX Adapted for STreaming) encoded message into a structured order format.

Initialization: The function starts by initializing an array encoded_message that will hold the encoded data extracted from the first_packet and second_packet. It also initializes several buffer variables to hold the decoded values for price, size, order ID, and order type.

Extracting Encoded Data: It loops through the bytes in the input packets (first_packet and second_packet), extracting the encoded data and populating the encoded_message array.

Decoding Price: The function begins decoding the price from the third byte of the encoded message. It first decodes the exponent and then the mantissa. The final price is calculated by multiplying the mantissa by 10 raised to the power of the exponent. This process utilizes a simple form of floating-point representation tailored for fast decoding.

Decoding Size: Next, the function decodes the size, which is a simpler integer value compared to the price. It reads the bytes until it finds the stop bit and constructs the size value from these bytes.

Decoding Order ID: The order ID is decoded in a similar manner to the size, reading the bytes until the stop bit is encountered and assembling the order ID from the extracted data.

Decoding Order Type: Finally, the order type is decoded. Since the order type is a smaller integer (represented in the code as uint3), it can be directly extracted from the next byte.

The function updates the **decoded_message** object with these extracted values, translating the encoded FAST message into a readable and structured order format.

The additional helper functions **decode_decimal_to_fix16, decode_uint_to_uint8, decode_uint_to_uint32**, and **decode_uint_to_uint2** are specialized decoders for different data types within the FAST message. They follow a similar pattern of extracting the bytes while considering the stop bit and assembling the final value.

decode_decimal_to_fix16: Decodes a decimal value to a fix16 format, handling the exponent and mantissa separately.

decode_uint_to_uint8: Decodes an unsigned integer into a uint8 format.

decode_uint_to_uint32: Decodes an unsigned integer into a uint32 format.

decode_uint_to_uint2: Decodes a 2-bit unsigned integer.

Encoder:

Networking Interface:

Defines constants REQUEST and REPLY for network operations.

replyTimeOut, MY_MAC_ADDR, and BROADCAST_MAC are set for network communication timing and addressing.

axiWord struct represents a network data packet with data, keep, and last indicators.

sockaddr_in struct holds network address information, such as port and IP address.

metadata struct combines source and destination network socket information.

Encoder/Decoder Interface:

Defines message size and field order constants for the market data template. This includes sizes and order numbers for various fields like price, size, orderID, and order type.

Defines data types (fix16, bit, uint3, uint8, uint32, uint64, uint16) using ap_int.h for different precisions and sizes to represent numeric values efficiently in hardware.

The order struct represents a trading order with price, size, orderID, and type.

fast_protocol Function:

This is a primary function signature designed to handle FAST protocol encoding/decoding.

It takes in multiple hls::stream parameters which are specialized data types for streaming data in hardware. These streams handle network data, metadata, time tags, and order information.

lbRxDataIn and lbRxMetadataIn likely represent input streams for received network data and metadata.

lbRequestPortOpenOut and lbPortOpenReplyIn are streams for handling network port request and reply to mechanisms.

lbTxDataOut, lbTxMetadataOut, and lbTxLengthOut are output streams for sending out encoded data, metadata, and data length.

tagsIn and tagsOut might be used for timestamping or tagging data packets.

metadata_to_book and metadata_from_book, along with time_to_book and time_from_book, are streams for handling order book-related metadata and timing information.

order_to_book and order_from_book are streams for sending and receiving order data to and from the order book.

Protocol Processor:

This part implements the FAST (FIX Adapted for STreaming) protocol, which is used for high-speed data encoding and decoding in financial applications, particularly for market data and order handling. Here's a breakdown of the main components and functionalities of the code:

Header Includes and Type Definitions: The code starts with the inclusion of necessary header files and definitions of custom data types. These types, such as fix16, uint3, uint8, uint32, etc., are defined for various bit-widths to represent different pieces of data accurately.

Networking Interface: Constants and structures are defined for handling networking, including MAC addresses, port numbers, and data structures (axiWord and metadata) for handling network packets and associated metadata.

FAST Protocol Message Definition: Defines the structure and size of a FAST protocol message, detailing the contents of each field within a message. It also outlines the expected sizes of messages.

Order Structure: Defines an order structure that holds information about a market order, including price, size, order ID, and order type.

rxPath Function: This function handles the reception (RX) path for network data. It reads incoming data packets, processes them based on the current state of a finite state machine (FSM), and eventually decodes the FAST-encoded message into an order structure. The FSM transitions through various states, such as opening a port, reading packets, and decoding data.

txPath Function: This function manages the transmission (TX) path. It encodes orders into the FAST format, divides the encoded data into packets, and sends them over the network. Similar to rxPath, it uses an FSM to manage its operation, moving through states like reading orders, encoding them, and writing out the network packets.

fast_protocol Function: Acts as the main interface that ties the RX and TX paths together. It uses HLS pragmas to optimize the function for hardware implementation, indicating that the function should not return (it acts as a continuous process in hardware) and that it should use a dataflow model to allow concurrent operation of the rxPath and txPath functions.

Finite State Machines (FSMs): Both rxPath and txPath functions implement FSMs to control the flow of their operations, transitioning between states based on the availability of data and the completion of specific tasks like reading a packet or encoding a message.