High Frequency Trade Book Builder using FPGA

Choka Thenappan[ct3185]
Shivam Shekhar[ss6960]
Ameya Keshava Mallya[am6024]

Columbia University
Fu Foundation of Engineering and Applied Science
CSEE 4840 Embedded System Design

Abstract. High-frequency trading (HFT) is a form of algorithmic trading that involves the rapid execution of a large number of orders at extremely high speeds. Key characteristics include: Speed, Low Latency, High Order-to-Trade Ratios, Market Making and Accuracy. The HFT engine is responsible for receiving this data, parsing it with the protocol provided by the exchange, updating its internal state about the market, submitting orders back to the exchange in reaction to market updates, and storing the buy and trade in a sorted order book that keeps records per share. In this project, we to build a Low-Latency FPGA based Order book for High Frequency Trading.

1 Introduction

High-frequency trading (HFT) refers to the practice of using powerful computers and algorithms to execute trades at extremely high speeds in financial markets. These trades are typically executed in fractions of a second, taking advantage of small price discrepancies or market inefficiencies.

- **Low-Latency Infrastructure**: HFT firms invest significant resources in building ultra-fast trading infrastructure to minimize latency—the time it takes for data to travel from one point to another.
- **Market Data Feeds**: HFT firms subscribe to direct market data feeds from exchanges, which provide real-time information about prices, trade volumes, order book changes, and other relevant market data. These data feeds are crucial for making split-second trading decisions.
- **Execution Platforms**: HFT firms use advanced trading platforms and execution systems capable of processing large volumes of orders rapidly. These platforms often feature co-location services, which allow firms to place their trading servers in close proximity to exchange servers for further latency reduction.

While it's possible to implement high-frequency trading (HFT) strategies using software running on traditional CPUs, there are several limitations that make FPGAs a preferred choice for many HFT applications:

- **Latency**: FPGAs offer significantly lower latency compared to software running on CPUs as they allow for the implementation of trading algorithms directly in hardware, bypassing the overhead associated with executing instructions in software.
- **Parallelism**: FPGAs excel at parallel processing, allowing multiple tasks to be executed simultaneously that can be harnessed to process large volumes of market data and make numerous trade decisions concurrently, improving overall throughput and responsiveness.
- **Low-Level Access to Hardware**: FPGAs provide direct access to hardware resources, allowing for fine-grained control and optimization of the underlying hardware architecture, whereas software running on CPUs operates at a higher level of abstraction and may be subject to limitations imposed by the operating system or underlying hardware platform.

2 System Block Diagram

For the Implementation we will simulate an real-world situation based on how these accelerators will be deployed. For this we will use the following components:

- FPGA (To run the book keeping algorithm)
- Personal Computer (To simulate the market)
- VGA Monitor (To display the market state)

We will connect the FPGA to our micro PC and have a program that sends market information to the FPGA, this setup will help us simulate the exchange. This data inside the FPGA will go to the parser module which will then process this information and send update requests to the order book to keep track of the current state of the market.

The complexity arises from the sequential nature of the order book, making it challenging to efficiently implement on an FPGA, which is a parallel processing unit.

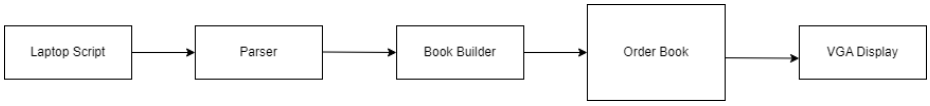

Fig. 1. Block Diagram for HFT Book Keeping

Laptop Script: The Peripheral connected for this stage will be the Personal Computer which will relay the simulated market data.

Parser: Processes the raw market data received from the Laptop Script and sends it to the Book Builder. The main task of the parser is to pass these messages to the order book while filtering out every other message. Given that the order book supports both add and cancel messages, it is feasible to create a module that is responsible with. This component is written in Verilog.

```
module parser_module (
    input logic clk,
    input logic reset_n,
    input logic data_valid, // Indicates when data_in is valid
    input logic [7:0] data_in, // Incoming data byte
    output logic packet_valid, // Indicates when a complete packet has been parsed
    output logic [7:0] parsed_data [7:0], // Parsed data bytes
    output logic error // Indicates a parsing error
);
endmodule
```

Book Builder: Book Builder accesses the book data parsed from the hardware and sends it to order book to store it. This is the linking point between the software and hardware component.

Order Book: 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, and a per stock data structure. Order book creates a linked list to store the received books and sends it back to the hardware to display it to the VGA Display. To build and maintain an order book, we have messages of the following types:

Message Category	Type	Message Value
Add Orders	add order	"A" = 0x41
Add Orders	add order with mipid attribution	"F" = 0x46
Cancel Orders	order executed	”E” = 0x45
Cancel Orders	order executed with price message	”X” = 0x58
Cancel Orders	order cancel	”C” = 0x43
Cancel Orders	order delete	”D” = 0x44
Cancel Orders	order replace	”U” = 0x55

VGA Display: Displays the output created by the Order Book which essentially displays the list of all the shares and their relevant data such as price, quantity, etc.

3 Algorithms

For the project we are making few assumtions to control the scope of the project. We will be keeping track of 4 stocks values in parallel and each stock value will be a 16-bit value. These values are chosen based on the memory and speed of the DE1-SoC in the FPGA.

The ORDER_BOOK module will be designed in SystemVerilog. The module will be instantiated 4 times as we want to keep track of 4 stock prices in our design. These 4 modules will have a wrapper, which will pass the instructions received from the market to corresponding module based on the STOCK_ID signal.

Each Order_Book module will have 3 sub-modules for Cancel, Execute and Add_New, whose functions are explained below.

We assume there will be three kinds of commands the Order_Book module receives from the exchange:

- Cancel : The exchange specifies an order id, and the order should be removed from the market.
- Execute : The exchange specifies an order id, and a quantity to decrease the order by
- Add_New : The exchange specifies the order id, the price, and quantity of an order. The order book should accept this as a new order and appropriately update the size and best price on the market for that stock.

We plan on using READY signal, which is a output of the Order_Book. This signal when high means that the order book is serving a message and can't currently take any messages.

4 Resource Budgets

In the FPGA we store order entries. As we plan on using SystemVerilog, each order entry will be structure with price, order id and quantity.

For scalability lets assume that the price and quantity are 16-bits each and the order id is 8 bits. So a given Order Entry is 40 bits.

An order entry is 5 Bytes, and an real-world stock can have requests ranging from 1000 to 1000,000,000 a day. For our design we will simulate 10,000 order entries which are canceled and executed. Hence we will need about 50 KB of memory for maintaining the order book.

Block RAM (BRAM) is a type of random access memory embedded throughout an FPGA for data storage. We look to use this BRAM for storing our Order_Book.

5 Hardware/Software Interface

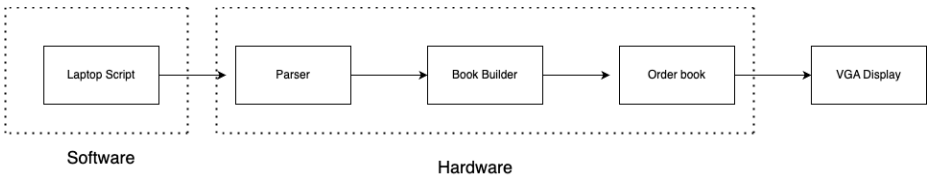

Fig. 2. Hardware-Software Distinguish in the system block Diagram

The Hardware Software Integration start with the C program relaying the simulated market data to the FPGA using the USB connector cable. To implement a program in C that sends data to an FPGA using a USB cable, you would typically follow these steps:

- Open a Connection to the USB Device: Use platform-specific APIs to open a connection to the USB device. On Windows, this involves using the CreateFile function to get a handle to the device.
- Set Up the USB Device for Communication: Configure the USB device for communication. This might involve setting up the correct configuration, claiming the interface, and ensuring that the device is ready to send and receive data.
- Send Data to the USB Device: Use the appropriate function to write data to the USB device. On Windows, this would be the WriteFile function.
- Close the Connection: After the data transfer is complete, close the connection to the USB device to free up resources.

The implementation would be something similar to as follows:

```
    #include <windows.h>
#include <stdio.h>

int main() {
    HANDLE hDevice;
    DWORD bytesWritten;
    char dataToSend[] = "Hello, FPGA!";

    // Step 1: Open a connection to the USB device
    hDevice = CreateFile(
        "\\\\.\\YourDevicePath", // Replace YourDevicePath with the actual device path
        GENERIC_WRITE,
        FILE_SHARE_WRITE,
        NULL,
        OPEN_EXISTING,
        0,
        NULL
    );

    if (hDevice == INVALID_HANDLE_VALUE) {
        printf("Failed to open device\n");
        return 1;
    }

    // Step 2: Set up the USB device for communication
    // This step is highly device-specific and may involve sending control requests
    // to configure the device. This example assumes the device is ready to use.
```

```
    // Step 3: Send data to the USB device
    if (!WriteFile(hDevice, dataToSend, sizeof(dataToSend), &bytesWritten, NULL)) {
        printf("Failed to write data to device\n");
    } else {
        printf("Data sent successfully\n");
    }

    // Step 4: Close the connection
    CloseHandle(hDevice);

    return 0;
}
```

Data Reception on the FPGA would be something similar to as follows:

```
module usb_interface (
input logic clk,
input logic reset_n,

// USB signals
inout logic usb_dp, // USB D+ line
inout logic usb_dn, // USB D- line

// Interface with the rest of the FPGA
output logic [7:0] received_data,
input logic [7:0] data_to_send,
input logic send_data,
output logic data_received,
output logic data_sent
);
```

Implementing a parser module within an FPGA using SystemVerilog involves creating a design that can interpret incoming data according to a predefined protocol or data format. This parser module would be responsible for extracting meaningful information from the raw data stream received by the FPGA. For the sake of illustration, let's assume we're parsing a simple protocol where messages are sent in packets that start with a start byte (e.g., 0xAA), followed by a payload of fixed length (e.g., 8 bytes), and end with an end byte (e.g., 0x55).

– **Step 1:** Define the Module Interface
  Defining the interface for the parser module. This includes inputs for the incoming data stream and control signals, and outputs for the parsed data and status signals.

```
        module parser_module (
    input logic clk,
    input logic reset_n,
    input logic data_valid, // Indicates when data_in is valid
```

```
    input logic [7:0] data_in, // Incoming data byte
    output logic packet_valid, // Indicates when a complete packet has been parsed
    output logic [7:0] parsed_data [7:0], // Parsed data bytes
    output logic error // Indicates a parsing error
);
```

– **Step 2:** Implement the State Machine
A common approach to parsing is to use a finite state machine (FSM). The FSM will have states corresponding to the different stages of parsing the packet.

```
        typedef enum logic [2:0] {
    WAIT_START,
    READ_DATA,
    WAIT_END
} parser_state_t;

parser_state_t current_state, next_state;
logic [2:0] byte_count; // To keep track of the number of bytes read

// State transition logic
always_ff @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        current_state <= WAIT_START;
    end else begin
        current_state <= next_state;
    end
end

// Next state logic
always_comb begin
    next_state = current_state; // Default to staying in the current state
    case (current_state)
        WAIT_START: begin
            if (data_valid && data_in == 8'hAA) begin
                next_state = READ_DATA;
            end
        end
        READ_DATA: begin
            if (data_valid) begin
                if (byte_count == 7) begin // Last byte of payload
                    next_state = WAIT_END;
                end
            end
        end
        WAIT_END: begin
```

```
                if (data_valid && data_in == 8'h55) begin
                    next_state = WAIT_START;
                end else if (data_valid) begin // Error condition
                    next_state = WAIT_START;
                    error = 1;
                end
            end
        endcase
    end

    // Output logic and byte counting
    always_ff @(posedge clk) begin
        if (current_state == READ_DATA && data_valid) begin
            parsed_data[byte_count] <= data_in;
            byte_count <= byte_count + 1;
        end else if (current_state == WAIT_START) begin
            byte_count <= 0;
            packet_valid <= 0;
            error <= 0;
        end else if (current_state == WAIT_END && data_valid && data_in == 8'h55) begin
            packet_valid <= 1;
        end
    end
```

To programmatically implement a VGA display of an order book on an FPGA using SystemVerilog, we create a module that generates the necessary VGA signals and displays the order book information on a monitor. The VGA interface typically requires generating horizontal and vertical synchronization signals, as well as RGB color signals for the pixels.

```
    module vga_display (
    input logic clk, // Pixel clock for VGA
    input logic reset_n,
    input logic [7:0] order_book_data [0:1023], // Example order book data
    output logic hsync, // Horizontal sync signal
    output logic vsync, // Vertical sync signal
    output logic [3:0] red, // 4-bit red channel
    output logic [3:0] green, // 4-bit green channel
    output logic [3:0] blue // 4-bit blue channel
);
```

References

1. Endrias, Tony, Natnael : An HFT (High Frequency Trading) Accelerator https://web.mit.edu/6.111/volume2/www/f2019/projects/endrias$_{P}roject_{D}esign_{P}resentation.pdf$

2. Endrias, Tony, Natnael : HFT Accelerator
https://web.mit.edu/6.111/volume2/www/f2019/projects/endrias$_{P}roject_{P}roposal_{R}evision.pdf$
3. Christian Leber, Benjamin Geib, Heiner Litz : High Frequency Trading Acceleration using FPGAs
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=6044837
4. Orderbook for High Frequency Trading (HFT) with FPGA
https://logictronix.com/wp-content/uploads/2021/10/OrderBook$_{H}FT_{2}O23_{2}021.pdf$