Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2022

Project Title:	**Caching on FPGAs for the speedup of algorithmic trading**
Student:	**Maëlle Guerre**
CID:	**014960401**
Course:	**EIE4**
Project Supervisor:	**Dr John Wickerson**
Second Marker:	**Dr Edward Stott**

Abstract

The demand for speed and competition drives technological innovation in financial sectors such as algorithmic trading. In the last two decades, the time taken to respond to incoming market data - tick-to-trade - and write it in limit order books (LOBs) has increased exponentially; the recent introduction of FPGAs promises even more progress in the forthcoming years.

In this project, 96-bit write-back caches were used to address the significant hardware delay caused by large client data stored in the FPGA chip DRAMs. By caching a subset of accumulated and cancelled orders, LOB's tick-to-trade for groups of 50 to 250 frequently traded stocks on the exchange accelerated by a factor greater than two.

The main research and development (R&D) challenge in algorithmic trading is demonstrating a system's robustness and dependability prior to its release to assess its benefits. Coverage is crucial to the project since an error or an attack on the system would result in billions of dollars of losses within seconds.

Constrained random testing, a common strategy used in conjunction with SystemVerilog Assertions, was implemented to ensure that adding caches had no impact on the system's behaviour for the vast majority of input combinations.

This proof of concept was suggested and reviewed by the High-Performance Computing team of Morgan Stanley. They found it to be "a successful and encouraging work that we welcome [the author of this project] to implement next year. We hope this will give the firm a competitive edge. "

Acknowledgements

Firstly, I would like to express my gratitude to Dr. John Wickerson for his consistent assistance, feedback, and willingness to learn about trading and supervise a project involving topics unrelated to his field. Not only did he help with the content of this project, but he also paid attention and gave great advice on general report writing.

I also thank Morgan Stanley, especially the High-Performance Computing team, for providing me with this project. In particular, Yousin Park and Hilda Xue, for spending several of their work hours explaining the structure of equity markets and FPGA order books within the firm. A special thank you to Changhoan Kim, Vice-President of the AMM team, for his enthusiasm for the subject and for providing me with critical metrics to calibrate my data.

Furthermore, I would like to thank the lecturers and students for helping with the technical set-up of the project, especially Yann Herklotz Grave for setting up a server for me and Jinge He for the advice and support on technical writing.

Lastly, I wish to state my gratitude to Keith O'Donnell, my mentor and previous manager, for his career guidance, teachings on project management, and planning. His priority was always to open opportunities at work related to the field of study I am the most passionate about.

Contents

Abstract **i**

Acknowledgements **i**

1 Introduction **1**

1.1 Motivation and Objectives 1
1.2 Related Work 3
1.2.1 LOBs on FPGAs 3
1.2.2 Caching 4
1.2.3 FPGA Bandwidth 5
1.2.4 Data validity 6

2 Background Theory **9**

2.1 Algorithmic Trading 9
2.1.1 The competition for speed 9
2.1.2 Order Book 11
2.1.3 Order types 12
2.2 Trading Model 14
2.2.1 Field Programmable Gate Arrays (FPGAs) 14
2.2.2 FPGA trading model 15
2.2.3 Storing Orders 16
2.2.4 FPGA operations on the LOB 19
2.2.5 A good metric: tick-to-trade 20
2.3 Caches 21
2.3.1 Write-back and write-through 21
2.3.2 Write allocate methods 22
2.3.3 Snooping protocol 22
2.3.4 MESI protocol 22

2.4 Finite-state machine (FSM) . . . 23
2.4.1 Blocking and non-blocking assignments . . . 23
2.4.2 Synchronous and asynchronous models . . . 24
2.4.3 Moore and Mealy designs . . . 26
2.4.4 Race conditions . . . 27
2.5 Language and Tools . . . 27
2.5.1 SystemVerilog . . . 27
2.5.2 Icarus Verilog . . . 28
2.5.3 Questasim . . . 28
2.6 Verification . . . 29
2.6.1 Verification testing . . . 29
2.6.2 Coverage . . . 30
2.6.3 Directed versus Random testing . . . 31

3 Requirements **34**

3.1 Executive Summary . . . 34
3.2 Specific requirements for this project . . . 35
3.3 Projects objectives overview . . . 35
3.4 Deliverable: Software Simulation . . . 37
3.4.1 Realistic Baseline Model . . . 37
3.4.2 Trading Simulation . . . 37
3.5 Ethical, Legal and Safety Considerations . . . 39

4 Analysis and Design **42**

4.1 General design choices . . . 42
4.1.1 Simplifications and Assumptions . . . 42
4.1.2 Risk Checking . . . 47
4.2 Data type design . . . 48
4.3 Logic design . . . 50
4.3.1 Logic blocks . . . 50
4.3.2 Finite-state machines . . . 51
4.3.3 Removing states . . . 53
4.3.4 Removing FSM . . . 56
4.4 Memory design . . . 57
4.5 Cache design . . . 58
4.6 Timing design . . . 60

5 Implementation **63**

5.1 General implementation 63
5.2 Data type implementation 65
5.3 Logic implementation 67
5.3.1 Downstream processor 67
5.3.2 Upstream processor 68
5.3.3 Challenges 70
5.4 Memory implementation 70
5.5 Cache implementation 72
5.5.1 Cache Components 72
5.5.2 Cache FSM 74
5.6 Timing implementation 76

6 Testing **79**

6.1 Risk Assessment 79
6.2 Testing Coverage 82
6.2.1 Testbench Architecture 83
6.2.2 Input bounds 85
6.2.3 Output generation 86
6.3 Testing logic 87
6.3.1 Basic checks 87
6.3.2 SystemVerilog Assertions 87

7 Results and Evaluation **89**

7.1 Threats to validity 89
7.2 Baseline model performance 91
7.2.1 Coverage 91
7.2.2 Functional performance 91
7.2.3 Model efficiency 92
7.2.4 Computing the penalty coefficient 94
7.3 Write-back cache examples 94
7.3.1 Coverage and Functional performance 94
7.3.2 32-bit cache, 50 clients 95
7.3.3 12-bit cache, 250 clients 96
7.3.4 12-bit cache, 20 clients 97

7.3.5 Write-back cache performance 98
7.4 Write-through cache performance 101
7.5 Parameters studied 104
7.6 Best performing model 105
8 Conclusion & Further Work **106**
8.1 Summary of Thesis Achievements 106
8.2 Applications 106
8.3 Future Work 107
9 User Guide **110**
9.1 Prerequisites 110
9.2 Accessing the project 111
9.3 Running the project 111
10 Appendix **113**
10.1 Project development 116

Acronyms

BRAM Block Random Access Memory

CPU Central Processing Unit

DPBRAM Dual Port Block Random Access Memory

FPGA Field Programmable Gate Array

FSM Finite-state machine

GUI Graphic User Interface

HDL hardware description language

HFT high-frequency trading

HPC high-performance computing

HPE high-performance engineering

LOB Limit Order Book

NYSE New York Stock Exchange

OOP Object-Oriented Programming

R&D research and development

SEATS Stock Exchange Automated Trading System

SEC Securities and Exchange Commission

SVA SystemVerilog Assertions

TCP Transmission Control Protocol

UDP User Datagram Protocol

Chapter 1

Introduction

1.1 Motivation and Objectives

On May 6^{th}, 2010, a computer-driven sale worth $4.1 billion triggered the May Flash Crash [1]. The Dow Jones Industrial Average – a price-weighted measurement stock market index of 30 prominent companies listed on stock exchanges in the United States – plummeted 1000 points within a single trading day. One trillion dollars was removed from the market value within five minutes. Since 70% of equity stocks in the US are traded by high-frequency trading (HFT) models [2], it is crucial that the decision engine is never allowed to make mistakes and always chooses the safest option [3]. The advantages of HFT over conventional human trades are the following:

1. *Speed*: computers trade orders in matters of nanoseconds instead of $1/10^{th}$ of a second – human reaction speed.

2. *Volume*: the software does not need to recover after sending a high volume of trades, some operations can be done in parallel.

3. *Safety*: when properly verified, software tends to make fewer mistakes than humans.

This project aims to provide another speedup for algorithmic trading while increasing safety through design decisions based on the risk-check model.

When sending mass orders, the tick-to-trade time between obtaining market data – tick – and acting on this new information – trade – is critical. Microseconds of trading can result in millions of pounds in gains or losses: cancelling an order too late decreases its value as the market falls. It is also pointless for brokers to place orders after their competitors because the price of stock changes after each order. Investment banks spent time and money refining their trading strategies. For example, they could act on trades faster using C++ models than Python ones because of additional control over the software; parallelism enabled multi-threading, and so on. Every aspect of the process is now supervised and accelerated. What further steps can be taken to improve efficiency?

Once they had reached the software limit, banks started investing in programmable hardware: Field Programmable Gate Array (FPGA). "It is not just (...) high-frequency traders that are looking to more flexible, agile versions of hardware. So are prime brokers, broader quant investors, crypto-currency miners, and banks' trading floors" [4].

Today, this technical breakthrough offers banks a great competitive advantage. For example, Nexus, a low latency software firm, enables trade latencies as low as 31 nanoseconds, "a 10-times speed improvement by moving software from CPUs to FPGA firmware." – Matthew Grosvenor, Sydney-based senior vice president of technology at Nexus.

Critical trading information is stored in on-chip Dual Port Block Random Access Memory (DPBRAM) blocks. DPBRAMs are single-cycle read and write on-chip memory. They were chosen for their size and speed. Because they live on an FPGA chip, they are usually easy to access; data is written/read in each line in a single cycle as long as it fits the FPGA bandwidth during the time to perform a clock cycle. With tick size decreasing, large floating points are stored in BRAMs along with other crucial information, taking 128-byte long memory lines to store a single client's data. The data size exceeds the data bus capacity, DPBRAMs cause hardware delays when cancelling and sending orders. Would it be possible to avoid such delay and speed up the model?

Orders and cancellations tend to be repetitive. The stock of only a few clients will vary for a limited time due to sociopolitical factors influencing the market. The repetitive pattern of orders motivates this project: caching could improve the bottleneck caused by the large DPBRAMs. Cache hits should be expected for repeated orders. Caching the memory line corresponding to this subset of clients should therefore speed up the data fetching phase. The complex address will not have to be computed, avoiding the bottleneck described.

Since the May Flash crash, banks are especially careful about testing and verification when it comes to front-end technology that can directly communicate with the exchange and forward orders. Therefore, one of the key aims of this project is to formally verify the design and provide confidence in the model.

The technique of ensuring that a logic design conforms to its specification is referred to as functional verification. At this stage, code in a hardware description language (HDL) defines the component's organisation (hierarchy, pin interface, modules), state variable allocation, and behaviour down to the binary function driving each electronic signal. This HDL code can be simulated with commercial software tools and synthesized into gate-level circuits on FPGAs. Functional verification is widely recognised as a bottleneck in the hardware design cycle, and it has become increasingly difficult as the need for higher performance and shorter time to market grows.

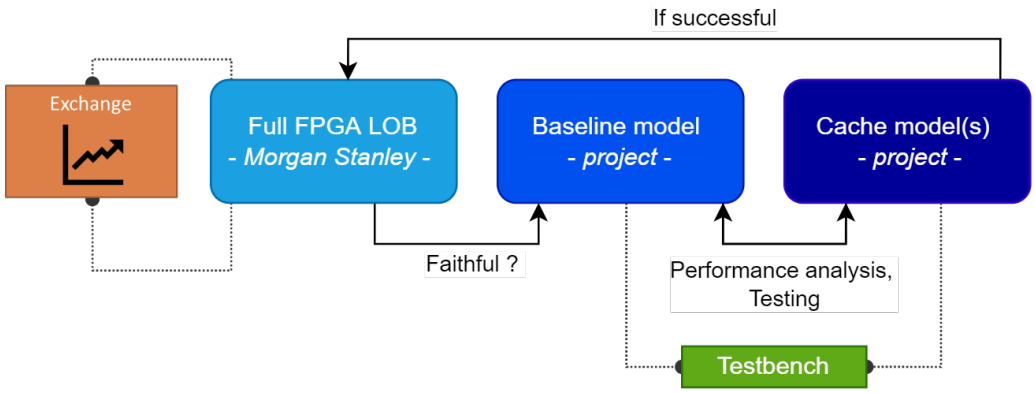
This project aims to simulate a realistic cache and memory Limit Order Book (LOB) to send and cancel mass orders, then to use verification to check the model's coverage and compare performance for different configurations, as displayed in figure 1.1.

1.2 Related Work

1.2.1 LOBs on FPGAs

Several studies from 2014 to 2022 promote the implementation of a LOB on FPGAs, in whole or in part, such as "Low latency book handling in FPGA for high frequency trading" [5] and

Figure 1.1: Overview of the project's aims.



"Exploring the Potential of Reconfigurable Platforms for Order Book Update" [6]. Combining the software with some hardware has been proven efficient on several occasions.

However, due to the secretive nature of trading, all studies were based on assumptions and hypothetical models that are not used in practice.

1.2.2 Caching

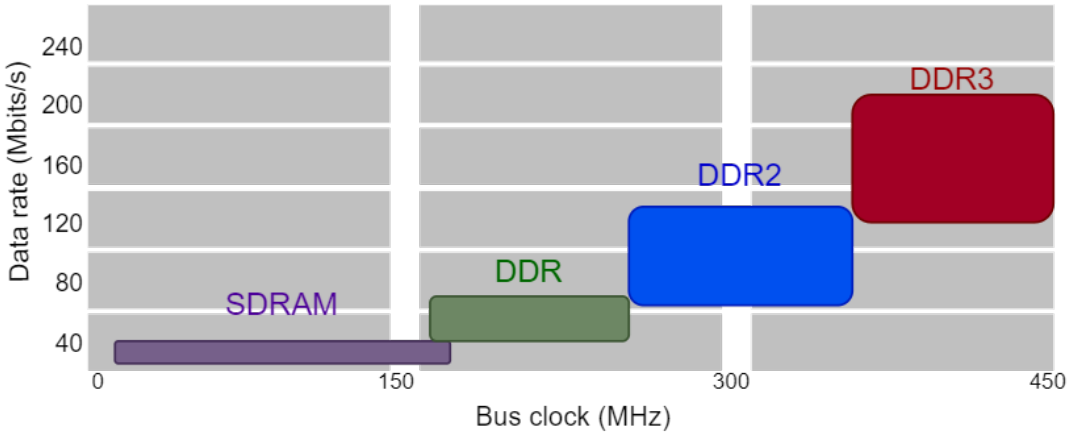
Caching to improve memory latency in hardware to implement a trading model is a new field with no precedent. Caching was studied in theoretical contexts, such as graphics processing [7] and studies on memory sub-types [8]. As a result, solving this problem is particularly intriguing, novel, and challenging.

It was argued that the increased complexity of having to retrieve the cache block can take longer than retrieving it from the shared memory [9]. However, this only happens when the processors are on separate chips. Write-back caches have a lower requirement for memory bandwidth, hence supporting a larger number of processors. This was useful for an on-chip cache, giving the developer freedom to design and develop a more complex model with more processors to access these memory blocks [10] [11].

Essentially, some components of this project have been studied in other contexts: caching on FPGA and implementing LOBs. The conclusions of these studies were encouraging since they were in line with this project's aims and results.

1.2.3 FPGA Bandwidth

Figure 1.2: Effect of Bus width on data rate. Adapted from Benjamin F. Harding and R. C. Cofer [12]



In the real trading model, a large BRAM was created inside the FPGA and changes the connection of the logic elements on the board. Since the data is too large to fit into a single block, the memory cannot fetch data in a single cycle. Figure 1.2 gives an idea of how the internal timing path and data rate can be slowed down by the data bus size. The maximum size on any FPGA board of a data bus is **128 bytes**, as detailed by Kevin E. Murray [13]. The logic is clocked at 500MHz [1] as per 2020 studies on FPGA multi-buses [14] and IBM FPGA bandwidth [15]. The maximum clock rate for memory is lower, at 100MHz, defined by Xilinx [16] and studied in 2020 [8] for accessing Block Random Access Memory (BRAM)s. Tables and graphs for data compression, access and partition are all displayed in Appendix 10.1 and 10.2. In reality, the memory clock frequency may be lower than 100MHz. The data must be extracted by and partitioned from memory and written into a single register. The exact on-chip bus structure used by Xillinx can be found in Appendix 10.3.

Based on these values, the maximum memory bandwidth is 12.8 Gbytes/s. Taking into account the size of an order – 128 bytes – the memory can process a maximum of requests of around 100 mega from equation 1.1. For the tick-to-trade value of **1ns** since 2016, the memory processes less than a fifth of the total orders sent in one second.

[1] "The working frequency of current FPGAs can reach several hundreds of MHz, however, it is unusual to see any complex FPGA logic running at more than 500 MHz [14]"

$$R_{mem} = \frac{128\ bytes \times 100\ Mhz}{128\ Ghz} = 99.7 \times 10^6\ requests/s \tag{1.1}$$

For the cache not to be impacted by the FPGA bandwidth limit, its size must be smaller than the maximum amount that can be processed for the 24-byte bandwidth, given the logic rate of 500MHz. The data length of 128 bytes does not impact the computation as it is already loaded in the cache registers.

$$Max_{cache} = \frac{128bytes \times 500MHz}{1GHz} = 64\ bytes = 512\ bits. \tag{1.2}$$

Cache design and analysis were based on this value. Any cache of a size greater than 64 bytes was modelled taking into account a hardware delay, reducing the benefits outlined above.

In this model, one cache tag was only 8 bits to simplify the implementation. From this information, the maximum cache size is $64 * 8\ bits = 64bytes$. For a cache size greater than **64**, the hardware delay was computed linearly with a penalty score equal to the memory penalty of 7 cycles.

1.2.4 Data validity

The collection of testing data was the greatest source of errors in this project. Previous information about bids, asks, and LOB data is highly protected and secret as it gives firms a competitive advantage. This allows them to train models on more data and learn about example LOB order strategies. For this reason, most companies only use their own recorded orders and do not share them.

Some data statistics and small samples can be found on the Nasdaq Book Viewer [17]. The daily share volume on the 25^{th} of March 2022 was $4,502,377,462$. Details about the clients come at an additional cost because knowledge provides companies with opportunities for wealth, and advantage. The cost of obtaining general information about trade statistics for a single day

is over £400. Brokers and institutional investors have access to the order-book information in detail, whereas the aggregate version is representative of what online traders would see [18].

$$Share\ Frequency = \frac{4,502,377,462}{24 * 60 * 60} \approx 52,111\ shares/second \tag{1.3}$$

In *The information content of an open LOB*, Charles Cao et al(2012) provide some statistical data used in their mathematical model [18]. Using data from the Stock Exchange Automated Trading System (SEATS) Trading Screen, they were able to view details on individual buy and sell orders.

Using a similar technique to these authors and to Bessembinder and Venkataram (2004) – who built a LOB from the Paris Bourse, – deductions can be drawn [19]. Table 1.1 shows the cross-sectional mean, standard deviation, minimum and maximum daily share volume, trade size, and the number of trades for 100 sample stocks in the Australian Stock Exchange, which behaved similarly to the US Stock [19] from March 1 to March 31, 2000. On the same day, the most active stocks had 1,323 transactions and the least active 63 out of the 1,842,371 transactions every day. From this data, it seemed reasonable to consider periods of trades where **50-250** clients' shares were solely traded by a given broker. The trade size and share volume standard deviation between clients allow the author of this project to reduce the testing space to a small number of clients who took up most of the trading volume for a time frame of up to a second.

	Share Volume (Million)	*Trade Size (Share)*	Nb. of Trades
Mean	1.783	5,279	349
Standard Deviation	1.967	5,158	266
Minimum	0.127	1,186	63
Maximum	12.791	37,532	1,323

Table 1.1: Daily characteristics of the 100 Most Actively Traded Stocks on the Australian Stock Exchange [18]

The distribution of the size of market orders has been found by Marce Avellaneda and Sasha Soikov to obey a power-law [20]. The density of the market size is $f^Q(x) \propto x^{-1-\alpha}$ for a large x with $\alpha = 1.53$ for US stocks (Gopikishnan et al. (2000), 1.4 for NASDAQ shares (Maslow and

Mills, 2000). Good testing data would therefore take into account the fact that US stocks have larger market orders than NASDAQ shares, and the result may have to be scaled.

Chapter 2

Background Theory

2.1 Algorithmic Trading

2.1.1 The competition for speed

Who is involved?

Algorithmic trading has spread to trading, investment, and brokerage firms, as well as retailers. Such popularity compelled algorithmic trading participants to be faster and more competitive, especially since financial markets were modified for electronic execution and communication.

Figure 2.1: Contribution of Algorithmic trading to other global trading volumes, 2015. Chart 1 and 2 – Morton Glantz, Robert Kissell [21]; Chart 3 – Thomson Reuters.[22]

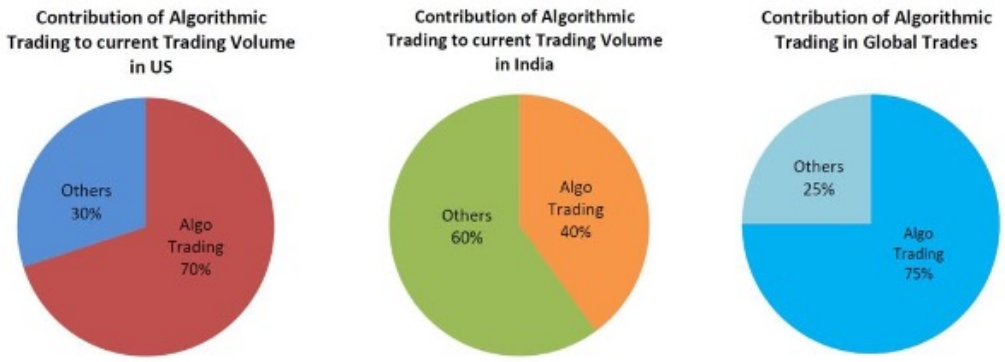
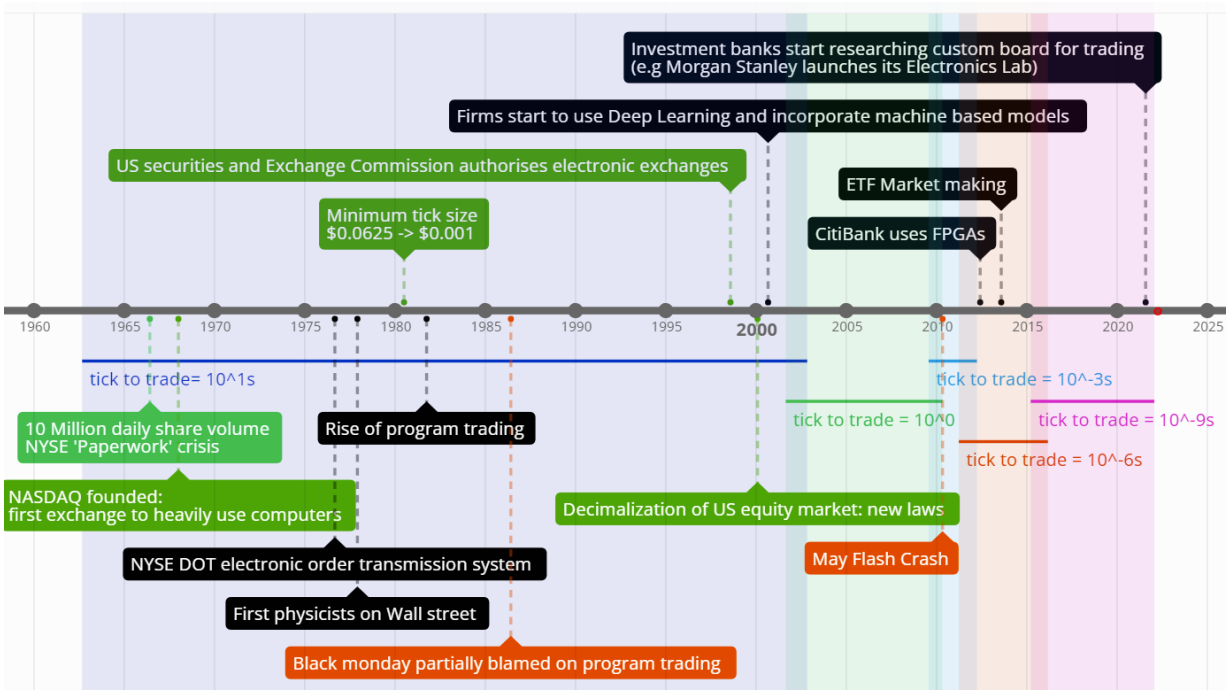


Figure 2.2: Timeline showing events in Algorithmic Trading relevant to this project. Green represents political events, black refers to technical improvements. full chart: https://time.graphics/line/642882

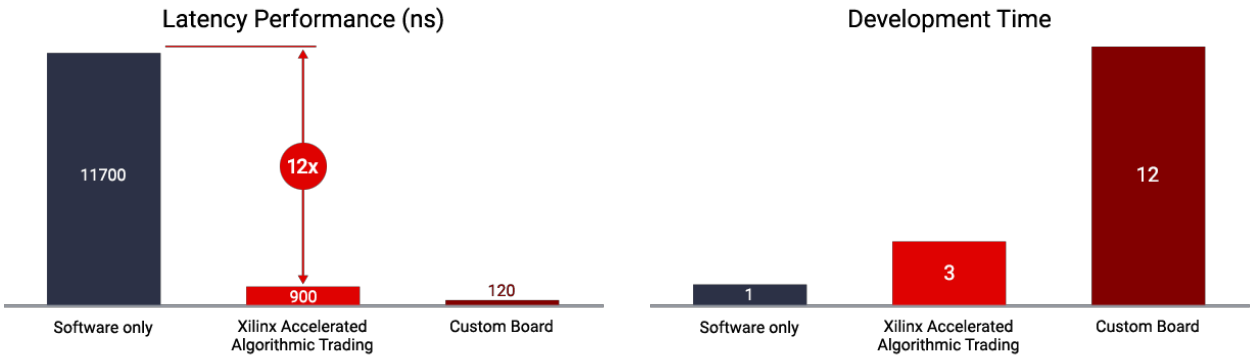


The minimum tick size[1] was drastically reduced from $1/16^{th}$ of a dollar to $1/100^{th}$ in the late 1980s. This altered the market structure, allowing for smaller price differences between bid and offer, facilitating algorithmic trading at a greater scale. This also required new ways to store data since the size of orders increased due to precision arithmetic.

Since then, these investment banks have been working to improve their algorithms to increase trade volume and speed. The Securities and Exchange Commission (SEC) of the United States approved electronic exchanges in 1998, paving the way for computerized HFT [23].

By 2001, the time between each trade was only a few seconds. As shown in Figure 2.2 the tick-to trade decreased to milliseconds in 2010, and nanoseconds two years later. The first improvements were made to the software. Financial firms quickly reached the limits of software through the mean of high-pressure, high-value investments in R&D. Citi Bank then came up with the idea of using a combination of software and electronic hardware (FPGAs) to speed up their model by a factor of 12 (see figure 2.3a). According to New York Stock Exchange (NYSE)

[1]Tick size: precision of each order in dollars, this by how much more the traders are allowed to bid compared the existing price.



(a) Comparing latency performance between software and FPGA algorithmic trading.

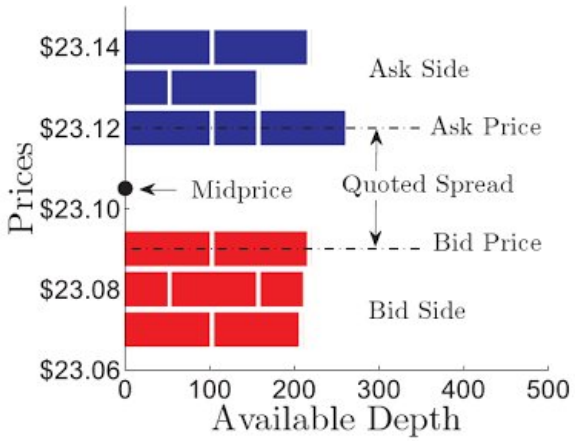
(b) Comparing development time between software and FPGA algorithmic trading.

Figure 2.3: Different data types considered for the trading model. Any change from one to another requires a thorough assessment of the development costs against the plus-value. Figures were taken from Xilinx website. [25]

[24], the volume of trades increased by 164% between 2005 and 2009, as shown in figure 2.1.

2.1.2 Order Book

Figure 2.4: LOB of an example security. The quoted spread and midprice are indicated in the figure. Source: Cartea, A., Sebastian, J. and Penalva, J. [26]



High-frequency trading is studied in the context of sending and cancelling orders through an order book: an electronic list of orders for a given security of financial instrument organised by price level. Each order book records the number of shares and the price at which they are being bid on or offered for: the market depth. The market participants behind these orders are referred to as clients [27]. Because they provide valuable trading information, order books

assist traders and improve market transparency. Figure 2.4 shows a sample order book. The goal is to have the closest bid and ask price. A LOB is a specific case of order books for limit orders, defined as a type of order to buy or sell a security at a specific price or better.

2.1.3 Order types

There are three parts to an order book:

1. **Buy orders**: Transmission Control Protocol (TCP) packets of data with buyer information, the amount they wish to purchase each bid and the corresponding ask price.
2. **Sell orders**: similar to buy orders, with seller information, the amount they wish to sell and the corresponding bid price.
3. **Order history**: this is specific to each market and includes all past transactions. It is widely used to create new buy or sell orders, identify patterns or train machine learning models.

Both buy and sell orders are sent by brokers or investment firms after a thorough risk check and complex computations to ensure that these will result in economic gain. In figure 2.4, buy orders are blue and sell orders are shown in red. This project focuses on the firm's side which send the orders. The different variables involved in the implementation of an order book are:

1. **Size**: number of shares to be bought or sold.
2. **Amount**: price of a single share proposed
3. **Bid/Ask price**: price to sell or buy this share set by the market.

An order remains in the order book until it is fully executed, i.e. until its size is zero from trades. The goal of the model on the left-hand side of figure 2.6 is to respond to questions such as "what is the best bid and offer?" or "how much volume is there between prices A and B?". [28].

The majority of the activity in a LOB is usually made up of add and cancel operations as market makers jockey for the best position [29]. An example is depicted in listing 2.1 for a simple LOB with five limit orders: sell 150 shares of Imperial College Plc. At $10.22, buy 100 shares at $10.13, then buy 180 shares for $10.18.

Listing 2.1: Simplified order book. Price is ordered in ascending order; $Ask_{price} < Bid_{Price}$.

```
ID              Price       Ask Size        Bid Size
Imperial Ltd.   $10.25      130
                $10.22      150                     -- Sell #1
                $10.21      120
                $10.18                      180     -- Buy #1
                $10.13                      100     -- Buy #2
```

Orders on the bid/ask side are the orders to buy/sell. In listing 2.1.3, the best ask is the lowest ask, and the best bid is the highest bid. If the ask was lower than the bid, this would be a crossed market and quickly resolved. The bid will therefore almost always be cheaper than the ask price. A heuristic is that a bid is the revenue of the stock at any given time while the ask is the cost, so the market will only ever offer a profit to itself - not to the liquidity seeker. The spread is the difference between the bid and ask prices, here $3. Traders compete for the lowest asks and highest bids per trade. There are four operations that a LOB must implement:

1. **Add**: places a buy limit order at the end of a list of orders to be executed at a limit price (= sending a `new_order` to check the `accumulated_amount` of past orders doesn't go over a limit). Frequency:$10,000 + /s$

   This affects the accumulated sent orders price in memory.

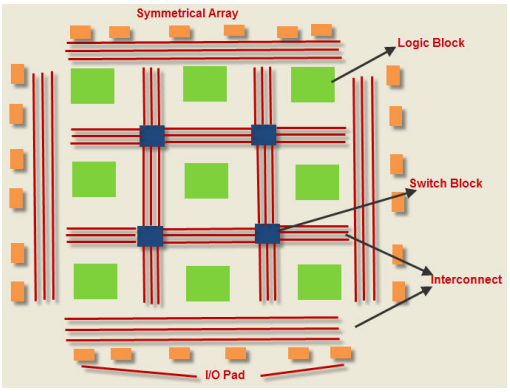
2. **Cancel**: Removes an arbitrary order that was placed during an `add` operation. Frequency: $5,000 + /s$

   This affects the accumulated order price in memory.

3. **Executes**: Removes an order from the inside of the book by passing through the exchange. Frequency:$10,000 + /s$

   This affects the accumulated cancelled orders price.

(a) FPGA board



(b) FPGA Logic block.

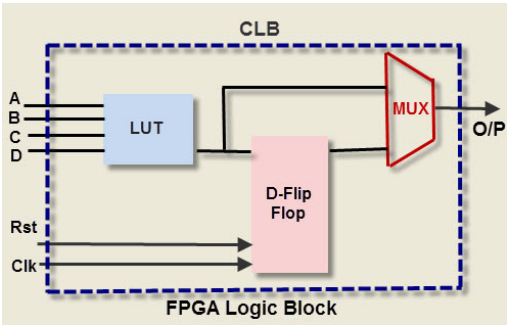


Figure 2.5: Layout of an FPGA. Source: HardwareBee

2.2 Trading Model

2.2.1 Field Programmable Gate Arrays (FPGAs)

FPGAs have gained popularity in algorithmic trading due to their superior speed and ability to re-compute lookup table connections. Banks can now profit from hardware that reacts in nanoseconds rather than having to develop new hardware for each improvement or modification to their trading model. FPGAs can cancel orders, maintain a secure connection and react to the exchange within nanoseconds, whereas software applications react in microseconds (c.f. figure 2.6), resulting in a theoretical[2] increase in speed of factor 12.

This project did not directly use FPGAs. Instead, the focus was set on demonstrating a proof of concept through simulation using a variety of tools. Look-up tables (LUTs) and flip-flops make up the majority of an FPGA [30]. It differs from general-purpose computers and custom hardware in the way that it allows users to program product features and reconfigure hardware for new applications after the product has been installed; it is referred to as field programmable. The two-dimensional array of logic gates in its architecture is referred to as gate arrays.

The three main blocks of FPGAs are shown in figure 2.5a.

1. *Programmable Interconnects*, for routing. Its guarantees that the block is connected, i.e

[2]The study comparing FPGA to software for speedup was based on common programs. It is inferred that a similar improvement should apply to this field, yet has not been reached because it is still in its early stages.

the TCP connections send the orders once they have been checked by the risk check module.

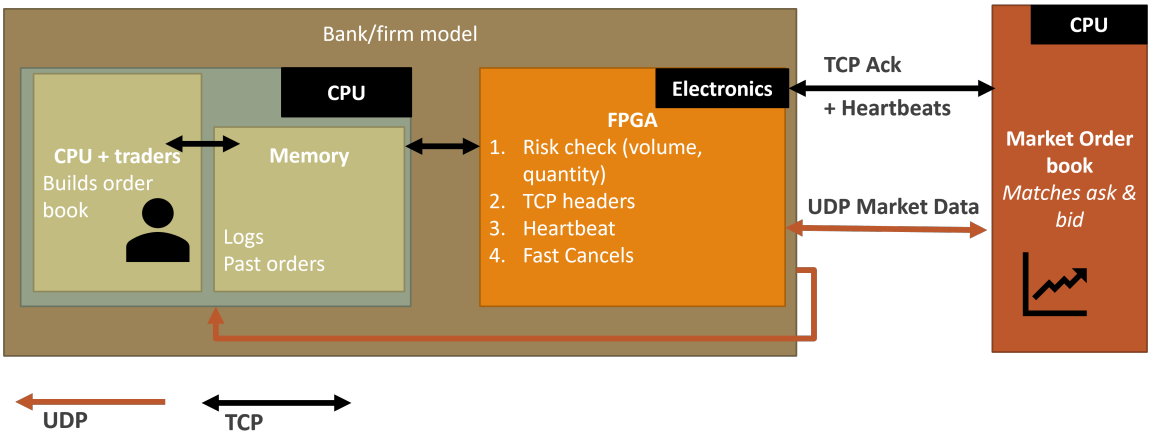
2. *Configurable Logic*, for functions, for example the risk check logic.

3. *Programmable I/O*, for external components. This allows the FPGA to be connected both to the exchange and the internal Central Processing Unit (CPU).

This architecture was kept in mind to reduce the number of flip-flops used, the number of look-up tables used, and the number of connections between logic blocks. The percentage of each logic block used by the simulated code is generally a good performance indicator in FPGA development.

2.2.2 FPGA trading model

Figure 2.6: Simplification of an Electronic trading model. On the left-hand side, the bank or trader combines software and hardware to find the best order to send at a competitive speed.



The FPGA and CPU blocks are both included in the order book electronic trading model in figure 2.6. The hardware and software parts of the order book balance complexity by splitting challenging tasks — managed by computer software — with simple ones from the FPGA. The CPU can store the firm's previous orders for up to months; it adjusts the weights of complex machine learning models by inputting sociopolitical factors to make predictions for the rise or fall of stocks in the future.

In figure 2.6, each trading model is equidistant to the market, which receives buy and sell orders. The TCP connection allows the bank/broker to create their order books and access the market data with no interruption. Hence, the model maintains a TCP connection with the exchange to cancel orders and a User Datagram Protocol (UDP) connection to send orders. As a result, the only arithmetic operation is risk checking.

Data flow is as follows:

1. Brokers, such as Morgan Stanley traders, send orders and cancels orders to make money depending on market fluctuations.
2. The FPGA check these orders, with different rules depending on the orders and/or companies.
3. The FPGA maintains a stable connection with the exchange, such as TCP connection, and sends over the orders, then receives confirmation from the exchange and forwards them directly to the broker.

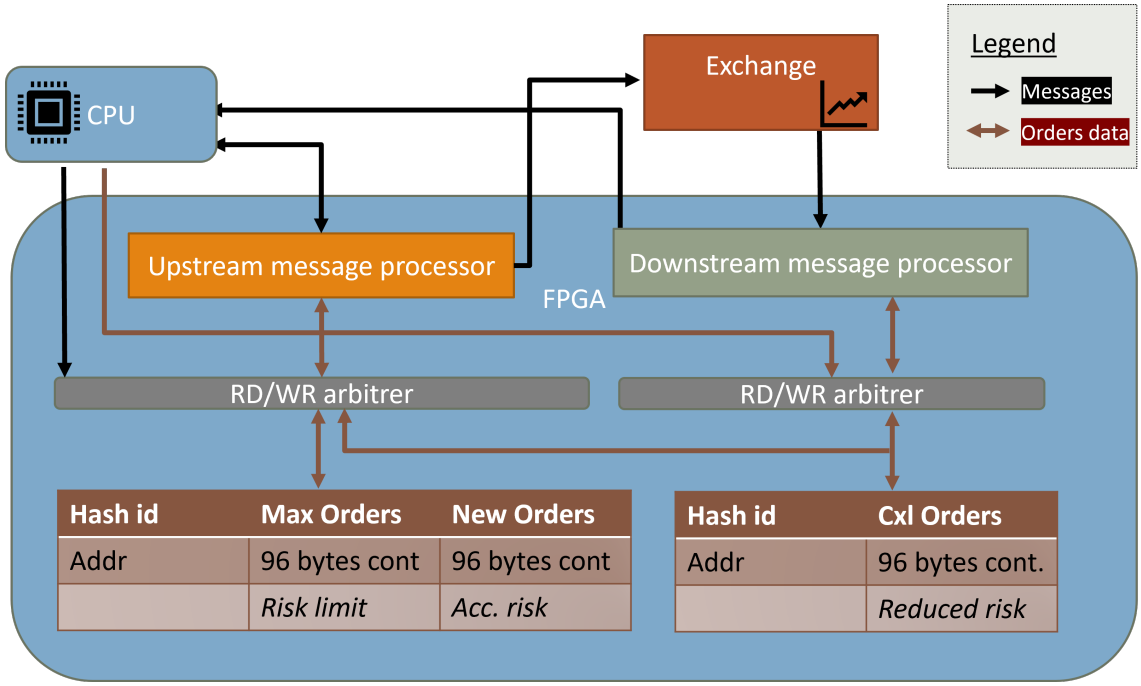
At a high level, the goal was to implement this design, then add caches between the arbiter and processors, as shown in figure 2.7b. The size, complexity, and layout of each cache were compared and tested.

2.2.3 Storing Orders

A block diagram of the FPGA risk check and order processing component is displayed in figure 2.7a. The CPU and exchange send/receive messages to/from the upstream and downstream processors. Upstream processor manages *accumulated risk*, defined in software as `new_orders`. Downstream processor processes *reduced risk*; it forwards and writes in memory the `cancelled_orders`. Each processor can access the memory blocks through an arbiter. These are essential.

1. They ensure that orders are cancelled if and only if they are lesser or equal to sent orders.

(a) Memory model implemented, as it exists within a trading firm.



(b) Caching model implemented.

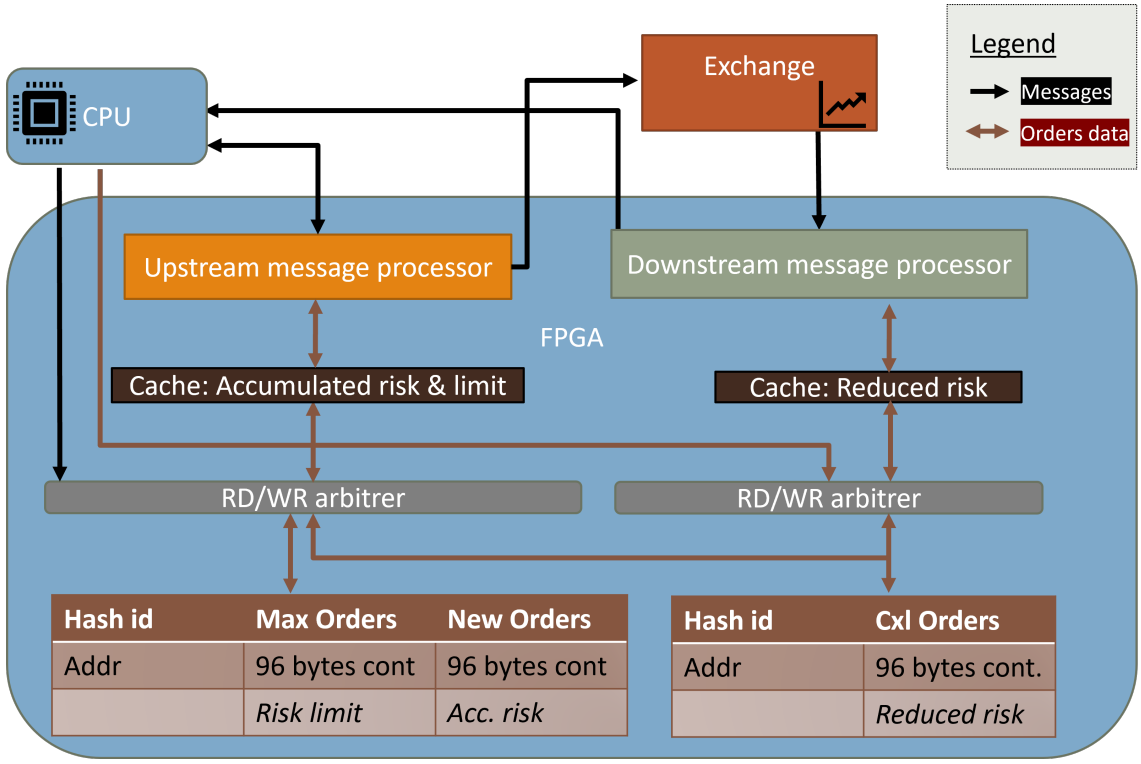


Figure 2.7: Block diagram of the memory and cache models implemented. Some simplifications have been applied.

Figure 2.8: Due to floating-point precision and large data width, the data for each client has to be stored at different addresses. When reading from this client, the data from each address has to be combined to retrieve the initial value.

Client ID	DATA
Imperial	ID, past information, full name, specific headers
addr_1	Accumulated orders (32 most significant bytes)
addr_2	Accumulated orders (32 bytes floating point)
addr_3	Accumulated orders (32 bytes floating point)
addr_4	Accumulated orders (32 least significant bytes)

2. In the case of simultaneous read and write of the same memory lines, for example, changing the risk limit and writing a new *accumulated risk* for the same client; the safest option always has the highest priority. This is discussed further in section 5.

Whilst messages' logic paths are one-way, data exchange is generally two-way in figure 2.7a. In most situations, the processors read data from an entry to check risk before writing a new value that corresponds to the new order/cancelled order.

Bottleneck 1: Order size

In the trading model from figure 2.7a, client identities and orders are stored on the FPGA in 128-byte continuous memory lines as shown in figure 2.8 [31]. This data is large, the majority of the processing time is therefore spent accessing and writing it back.

A single instrument example: NASDAQ instruments

The NASDAQ Total View tool, for example, monitors events in every NASDAQ-listed instruments [17], with data rates up to 20 gigabytes per day and spikes of 3 megabytes per second. Each message is 20 bytes long on average. As a result, during high-volume periods, the trading model's upper limit is $100,000 - 200,000$ messages per second. Theoretically, memory implementation should not matter for accessing orders since all operations except initiating a connection with the exchange have complexity O(1). In reality, the bandwidth and clock rate limit the data travelling in the FPGA bus. On top of this, there are significantly more orders than shown here since the market constitutes all existing instruments; NASDAQ is only one of

them.

As explained in a 2019 meta-study [8], The cost of transporting data between the memory and the processing blocks on the FPGA frequently outweighs its computational advantages. The low bandwidth between the FPGA's internal memory and its sped-up kernel for logic has become the main bottleneck. This is especially true in the context of algorithmic trading where a high internal bandwidth speeds up the kernel workload to make the best use of intrinsic parallelism:

Large data length
For each client, the order size, quantity and several other metrics are stored in a single continuous memory line, as displayed in figure 2.8. This makes it more difficult to retrieve the size or quantity alone as the correct memory line has to be fetched and computed.

Large data width
On top of the order size, quantity and other relevant information, the client id has to be stored in memory alongside the accumulated orders and maximum allowed to trade; both of which require 128 bytes due to floating-point decimal precision. This is larger than the Xilinx standard 32kbits BRAMs [25] Such detail is crucial since fluctuations of 10^{-3} are detected and acted on by the market. It is impossible to retrieve the entire value in one clock cycle.

2.2.4 FPGA operations on the LOB

Since all operations from the LOB are not implemented on the Field Programmable Gate Array – yet– its processors perform limited functions:

1. *TCP heartbeats*: Initiate a Transmission Control Protocol handshake with the exchange, and send regular heartbeats to acknowledge a strong connection to the exchange.

2. *Risk Check*: Simple arithmetic to ensure incoming orders are allowed before sending them as User Datagram Protocol (UDP) market data.

3. *Fast Cancels*: Quickly sends `cancelled_orders` to the market as User Datagram Protocol (UDP) packets.

Risk checking arithmetic is a simple comparison between the values in each memory entry:

$$\begin{aligned} Max\ allowed\ to\ trade \leq\ & Accumulated\ orders - Accumulated\ reduced\ risk \\ & + Incoming\ Order\ Value \times Incoming\ OrderQuantity \end{aligned} \tag{2.1}$$

Bottleneck 2: Hardware/Software balance

The split of tasks between the CPU and FPGA board in figure 2.6 varies depending on brokers and firms. Instead of searching for minor software improvements, how about writing an entire LOB on custom Hardware? The advantage of using hardware does not solely rely on the hardware itself, but rather on the hardware implementation design. The market is competitive; banks like *Barclays, J.P. Morgan, Morgan Stanley* and *Citibank* all have their implementation of an FPGA-based order book. However, the cost of moving towards a hardware-based model outweighs the benefices.

2.2.5 A good metric: tick-to-trade

Tick-to-trade is the time interval between receiving a market 'tick' (a price movement in the market) presenting the opportunity to the algorithm, and processing the buy or sell order [32]. Firms that respond quickly have systems that can generate trade orders faster than those of their rivals. And the slower the technology, the slower it will be to place orders on a market that has moved on.

By speeding up and improving the LOB simulated, a good metric to observe performance change is tick-to-trade. The testbench measured the difference in tick-to-trade latency to rate the performance of different models.

2.3 Caches

2.3.1 Write-back and write-through

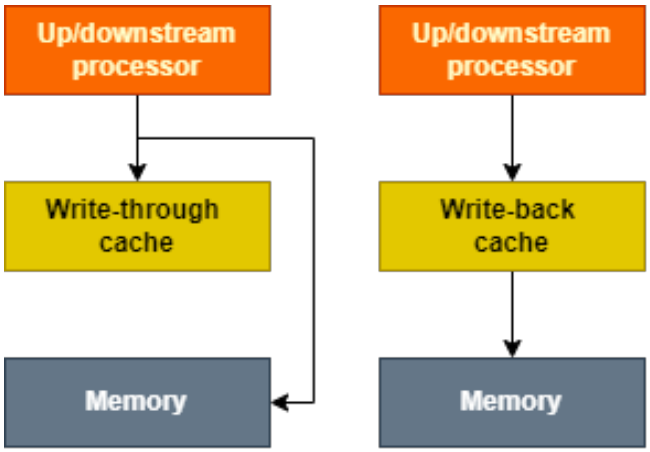
Both write-back and write-through protocols were implemented in this project to compare the performance of the different protocols outlined in table 2.1.

	Write-back	Write-through
Data is updated	Written to cache	Written to cache and memory
Operation	☹ Complex: data is lost if not done properly	☺ Easy
Cache-memory coherency	☹ Main memory and cache may contain different data	☺ Main memory always contains same data as cache
Data Write speed	☺ Fast	☹ Slow

Table 2.1: Write-back and write-through cache features.

When a cache miss occurs in a write-through cache, the data item can easily be found since the data is always in memory This provides additional safety (no data loss), simplifies the design – there are only two processors, it is easy to manage the memory and there are no simultaneous read or write – and increases speed – it is easier, hence faster to compute the correct address from memory. With "adequate memory bandwidth to support the traffic from the processors write-through simplifies the implementation of cache coherence" [33].

Figure 2.9: Simplified flowchart of the data path for write-through and write-back caches on a cache miss.



Write-back design is more complex. The latest value of data items can be in the cache without

any copy in the memory. Using the same snooping scheme for writes and cache misses as described for the write-through cache, the correct version can be found.

1. Each cache processor snoops all addresses placed on the bus
2. If a processor holds a dirty copy of the requested cache block, it will return this dirty copy and abort the memory request - the memory likely holds a dirty copy of the data item.

2.3.2 Write allocate methods

For the two types of caches, two methods were available for handling write-misses: write allocate and no-write allocate.

1. Write allocate, "fetch on write" has the data at the write-miss location loaded in the cache, this is a similar approach to a read-miss, and it is followed by a write-hit operation.
2. No-write allocate, "write around" does not have the data at the write-miss location loaded in cache: it is directly written to the memory.

2.3.3 Snooping protocol

In symmetric multiprocessing (SMP) systems where several processors access a single cache, the snooping protocol ensures memory cache coherency. Each processor access the cache through a bus; watches the bus, or snoops it, to see if it has a copy of the requested data block. Other processor cache copies must be invalidated or updated before a processor can write data. There are several snooping protocols to manage cache access.

2.3.4 MESI protocol

The MESI protocol is one of the most widely used invalidate-based cache coherence protocols that support write-back caches. This protocol is mostly used in multi-core architecture where

a design choice must be made between write-back and write-through caches[34]. Write-back caches can save a significant proportion of the total bandwidth that would otherwise be wasted if a write-through cache was used. In write-back caches, the dirty state indicates that the data in the cache differs from that in the main memory. If the block is in another cache, the MESI protocol requires a cache to transfer on a miss. In comparison to the MSI protocol, this protocol reduces the number of main memory transactions. This is a significant step forward in terms of performance [35].

This protocol has four states.

1. Modified: the block has been updated in the cache and not written back to memory.
2. Exclusive: The cache line is present only in the current cache, but is clean - it matches the main memory. It may be changed to the Shared state at any time, in response to a read request. Alternatively, it may be changed to the Modified state when writing to it.
3. Shared: this block is read/written by the two processors
4. Invalid: the block holds a dirty value of the data

For any given pair, the permitted states of a cache line are depicted in table 2.2.

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

Table 2.2: MESI protocol transitions.

2.4 Finite-state machine (FSM)

2.4.1 Blocking and non-blocking assignments

SystemVerilog allows *combinational* or *sequential* logic to be generated depending on the use of *blocking* or *non-blocking* assignments, respectively. Whilst software languages execute each

line one after the other, a hardware language like SystemVerilog lets the developer choose to execute logic concurrently or sequentially.

Listing 2.2: Non-blocking assignment example in SystemVerilog.

```
/* Non-blocking assignent example */
always @(posedge clk)
    a <= 1'b1; // clock cycle #1
    b <= a; // clock cycle #2
    c <= b; // clock cycle #3
end
```

Listing 2.3: Blocking assignment example in SystemVerilog.

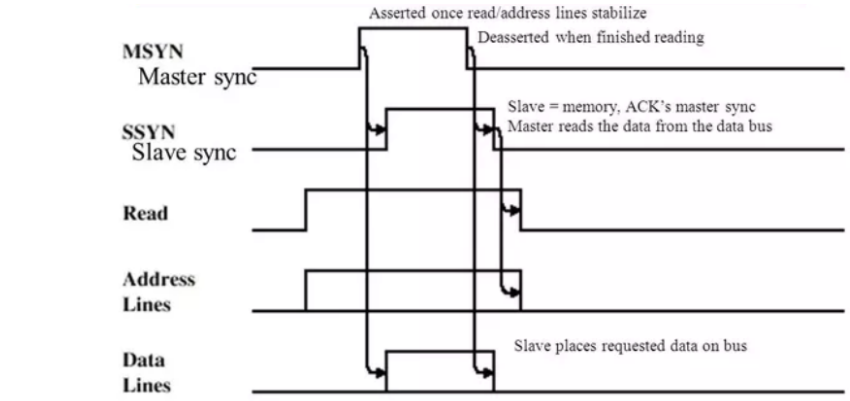
```
/* Blocking assignent example */
always @(posedge clk)
    a = 1'b1; // clock cycle #1
    b = a; // clock cycle #1
    c = b; // clock cycle #1, c = 1
end
```

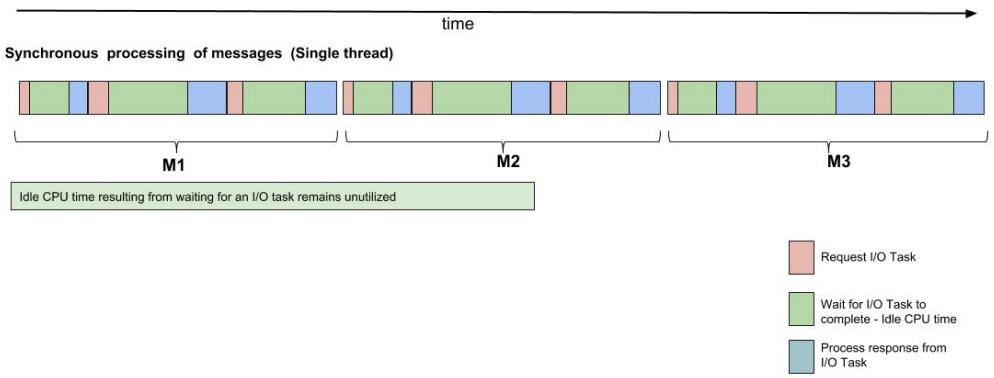
For sequential logic, such as FSMs, clocked blocks with non-blocking assignments ensure that each operation is performed on the clock rising edge, in the correct orders. This is the case in listing 2.2 When designing a more general model such as the LOB, combinational logic sped up the process; blocking assignments shown in listing 2.3 along with `always` blocks triggered on input changes performed better [36].

2.4.2 Synchronous and asynchronous models

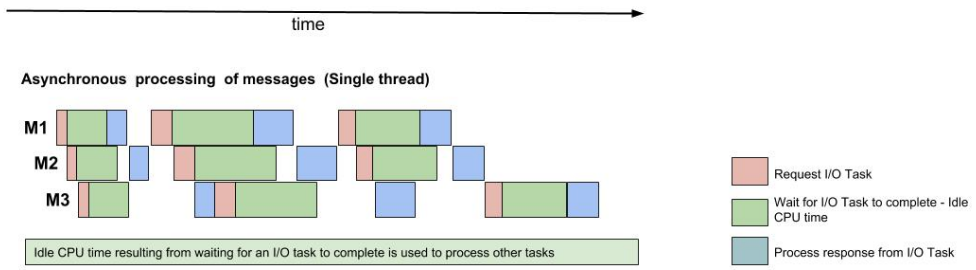
Compared to synchronous memory, every access request of the asynchronous memory has its own read and write latency, which can happen at any time rather than only rising or falling clock edges.

Figure 2.11: Time diagram of an asynchronous master/slave design. In this context, the processor drives the memory read/write. Data can be returned between clock edges. source: github.io





(a) Synchronous processing overview



(b) Asynchronous processing overview

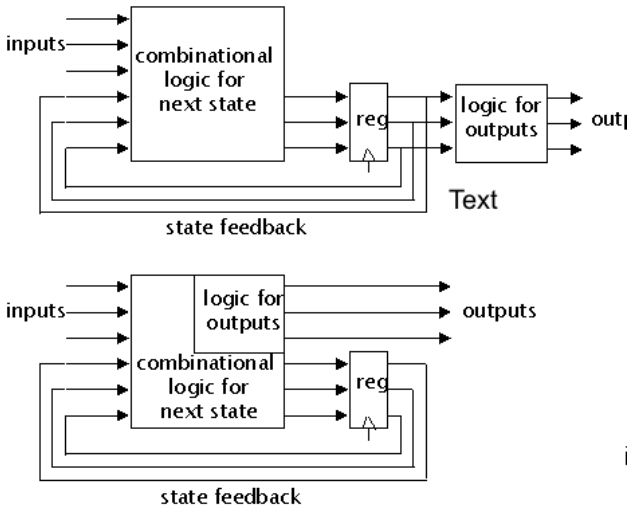
Figure 2.10: Asynchronous memory and logic designs allows to pipeline tasks and have a better throughput. source:github.io.

Asynchronous memory is generally more difficult to implement than synchronous when considering outputs values stability. Asynchronous memory controller processors share data and address pins with memory controller processors. Asynchronous chip selects, bytes, and read-

/write enable are all generated. There is a lot of freedom in this design; it is possible to generate a separate asynchronous memory control processor for different regions of the memory, should different read/write latency values be needed [37].

2.4.3 Moore and Mealy designs

Figure 2.12: Moore vs Mealy design.



In a Moore FSM, all outputs of the system solely depend on the states whereas a Mealy design's outputs depend on states and external inputs as depicted in figure 2.12.

Whilst Moore machines are usually popular because their output signals are synchronised with the clock; they were not best suited for this application. Ideally, the states would change as quickly as possible regardless of the clock. For example, the upstream processor must send the order immediately after the memory has returned the accumulated and maximum order values. Moore machine output signals cannot change until the rising edge of the next clock cycle to avoid setup timing violation.

On the other hand, a Mealy machine input signal may trigger a change in the middle of a clock cycle. If such change occurs after the time threshold for the next rising edge, the registers that hold the FSM next state could receive an incorrect output. Such design is more difficult to maintain, work with, and debug.

Topology	Pros	Cons
Moore	Simplicity Stability Predictability: safe to use	Lowest speed Pre-processing required May need extra signals for control
Mealy	Highest speed Several states can be processed in one clock cycle	Highest number of memory positions (states) Potential failures hard to debug

Table 2.3: Pros and Cons of using Moore vs Mealy design in Algorithmic trading.

2.4.4 Race conditions

A race condition occurs when a single system attempts to perform more than two operations simultaneously that must occur sequentially. This can happen when using blocking assignments in SystemVerilog. Especially when these assignments are inside `always` blocks that are triggered by input changes rather than the clock. The result of a race condition is dangerous: the wrong data can be read, written or the whole system may crash.

2.5 Language and Tools

2.5.1 SystemVerilog

Model implementation, development and testing were implemented in **SystemVerilog**, a hardware description language. It is generally used for high-level behavioural modelling, Register Transfer Level (RTL) behavioural modelling and well as design verification and development of testbench [38].

Listing 2.4: example of SystemVerilog assertions

```
1 //example of an immediate assertion  if (A == B) ...  // Simply checks if A equals B
2   assert (A == B); // Asserts that A equals B; if not, an error is generated
3 // example of a concurrent assertion
4   assert property (@(posedge Clock) Req |-> ##[1:2] Ack);
```

SystemVerilog modules were used to map configurable logic blocks from designed blocks such as figure 2.7a. SystemVerilog is a superset of another HDL: Verilog. The latter was dismissed to

use more data types and build a stable testbench using Object-Oriented Programming (OOP).

SystemVerilog Assertions (SVA)

SystemVerilog Assertions (SVA) are used to validate the behaviour of a design. They provide functional coverage information as they are checked during simulation. There are two types of assertions:

1. Immediate: only depends on the variables, it does not depend on the clock. This is the case for the first SVA in listing 2.4
2. Concurrent: uses sequences and properties that describe the design's temporal behaviour. This is clock dependent and may change through time, as shown in the second SVA of listing 2.4.

SVAs are particularly useful when randomising tests.

2.5.2 Icarus Verilog

FPGA architectural design flow comprises design entry, logic synthesis, design implementation, device programming, and design verification, as shown in figure 2.13.

Basic simulation and synthesis were performed using Icarus Verilog. This tool operates as a compiler, from a source code written in Verilog into some target format. For batch simulation, the compiler can generate an intermediate form: vvp assembly. This intermediate form is executed by the "vvp" command. For synthesis, the compiler generates netlists in the desired format. This was useful for checking for code behaviour under predefined circumstances [39].

2.5.3 Questasim

Sophisticated simulation and verification were achieved using Questasim software [40]. Different modules could be compiled into a single working library. The compilation could be done through

the command line or using a simulation environment that provides a thorough report on variable usage, clock timings, and system performance.

This was particularly useful when comparing two trading models. Timing diagrams and clock cycles could easily be compared to obtain the average tick-to-trade.

Altera provides the entry-level Model-Sim/Questasim Altera software, along with compiled Altera simulation libraries to simplify the simulation of designs. Examples of bash scripts used are shown in listings 2.5 and 2.6.

Listing 2.5: bash sript for Questasim Verilog compilation and simulation.

```
vlog -work work -O0 +fcover +acc -f code.sv
vsim -cvgperinstance -c <ARGUMENTS> work.tb_top work.glbl -do
"coverage save-onexit <Name_of_File>.ucdb; run -all;exit"
```

A complete html or text report can be obtained as well using listing 2.5.3

Listing 2.6: bash sript for Questasim Verilog reporting.

```
vsim -cvgperinstance -viewcov merged.ucdb -do "coverage report -file
final_report.txt -byfile -detail -noannotate -option -cvg"
```

2.6 Verification

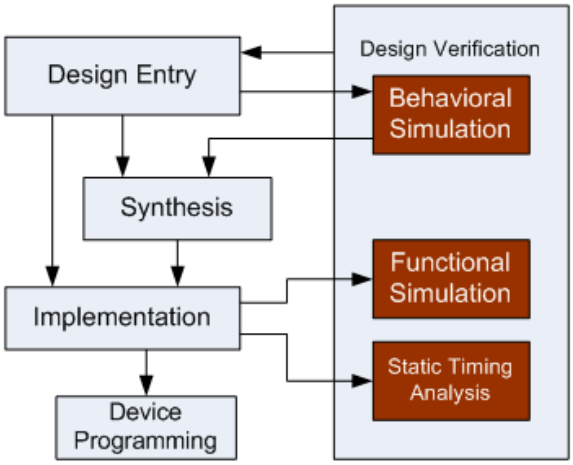
2.6.1 Verification testing

In trading, the cost of testing tends to be greater than 70% of the total product [41]. The focus is set on modelling systematic risk and systematic risk: the risk for one company-level risk to trigger a collapse and the risk inherent to the entire market, respectively. A software or hardware malfunction may cascade failure and result in a severe economic downturn.

For this reason, good testing is vital.

When writing traditional unit tests in SystemVerilog for the FPGA model, the vast majority of the input space is untested. There is no sound basis for extrapolating from tested to untested

Figure 2.13: Verification and Synthesis cycle in FPGAs. Figure taken from Xilinx website. [25]



cases and more tests must be written. Directed verification testing tends to be performed in late development stages to check model behaviour, it can be expensive to do changes to the model at this stage. Indeed, the later the software development stage, the more challenging fixing a model's errors is [42]. "Verification testing is a black-box testing strategy at either the system or subsystem level, and is performed to ensure that the system under test meets its requirements" [43]. In general, testing heavily relies on manual efforts in thinking of test cases and creating them; typically only a few tests are created to save time. For example, in software companies, test sets are generated from banks of tests and slightly modified to fit the current model. One downfall of such a technique is that tests must be re-written or updated every time the model is modified.

For the case of FPGA models – where the device's behaviour is meant to be changed regularly – traditional testing appears costly and pointless; writing hundreds of tests at every cycle of improvement of the model is a colossal task.

2.6.2 Coverage

In hardware verification, coverage is split into code coverage, functional coverage and statistical coverage. These three pillars define good testing in the context of this report.

Code coverage provides information about specific lines of the register transfer level (RTL)

code, such as lines hit, branches hit, expression hit or signal toggles hit. This is usually a product of other tests and ensures that different parts of the code are being tested. A full code coverage implies that all branches and lines have been run during testing.

Functional coverage collects information about user-defined scenarios or requirements. Directed tests are a good example of functional coverage – they show the success or failure of a constrained directed stimulus. More general randomised tests can also check for functional coverage.

Statistical coverage collects statistics across a group of simulations. It allows evaluating and assess the model, and enables fine-tuning of constraints to improve the stimulus. For example, functional coverage may be successful and code coverage may reach 100%. In this example, it is still possible that a single reset case was tested 99% of the time, with the other behaviour for the remaining 1%.

Coverage is defined based on points. A 100% coverage does not mean that all the code has been tested if the testbench does not declare all modules. By generalising the input space and improving the model step by step, a greater coverage space can be reached [44].

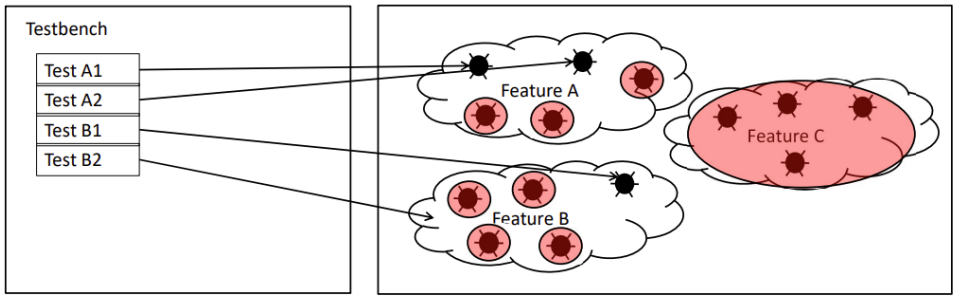
2.6.3 Directed versus Random testing

Directed testing

Directed unit tests ensure that each method of the classes and components used in the model is working.

Unit tests are cheaper and faster to implement and run. They constitute the most basic way to increase code coverage[45]. Directed tests are simple tests in which a specific situation for a known feature is recreated and the expectations are defined accordingly. In figure 2.14, several tests were created for a single feature; each one of them may have uncovered a different bug or misbehaviour in the model. As a result, direct testing usually has a heavy implementation cost and does not guarantee full code coverage. Whilst it is useful to test specific features, a more general approach was eventually preferred.

Figure 2.14: Visualising Directed Testing. Source: Imperial College London EE department [46]



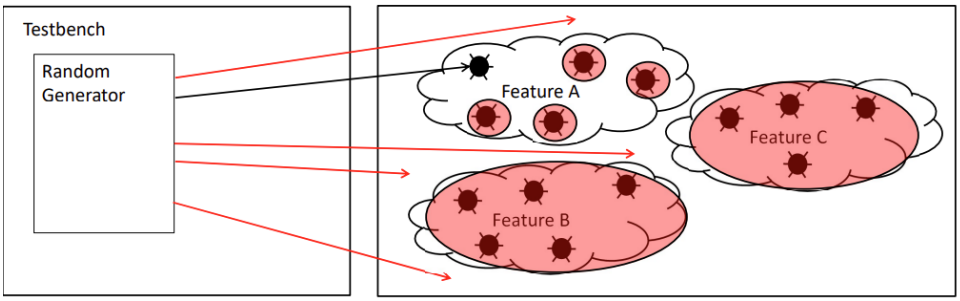
Random testing

Random testing is popular: it is simple to implement and allows for bias-free test selection. It is used to test a wide range of pieces of code from Java programs [44] to Haskell [47] to Graphical User Interfaces.

The effectiveness of random testing is questioned in the research world as it is time-consuming and less precise than systematic testing, yet can uncover some behaviours which would not have been tested otherwise [48]. To generate tests, a random input is applied to the model under test, and it is modified systematically to ensure that it can cover different paths [49] [50].

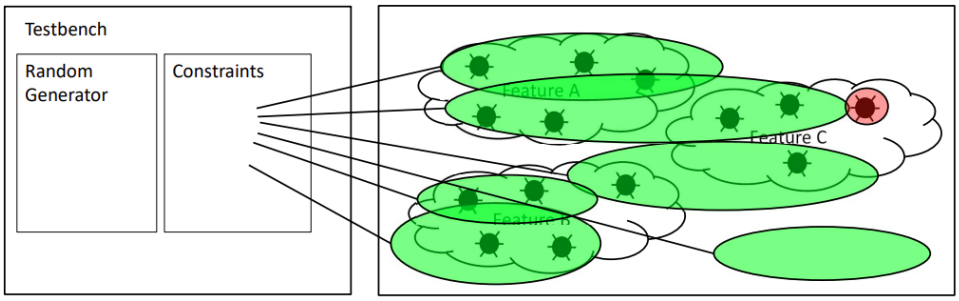
The main problem with random testing in the context of verification for HFT is that it only supports unit testing rather than black-box system testing. Each component would have to be individually tested, which makes scaling the tests difficult. Since a common cause of faults in FPGA development is communication among modules or scalability under large loads, such a technique was deemed better than directed testing but not ideal.

Figure 2.15: Visualising Random Testing. Source: Imperial College London EE department [46]



Constrained Random Testing

Figure 2.16: Visualising Constrained Random Testing. Source: Imperial College London EE department [46]



Constrained random testing generates a wide range of random scenarios within set constraints. Test parameters are restricted in order to keep the values within a specification range. This is a useful method due to the subtle nature of hardware faults and the variety of stimuli required to cover all scenarios in hardware verification. For example, constraints on the left-hand side of figure 2.16 cover a greater percentage of the testing space than directed testing, on the right-hand side of the figure. Constraints from this method still leave room for bugs and errors, shown in red, yet it was judged better than directed testing.

As a result, constrained random testing was considered to be an efficient methodology for Verification in HFT. Random variation of input values allowed to test a large section of the input space. The testbench was defined from the input variables. By randomising a greater percentage of the input space at each test, confidence was gained in the model's stability.

Chapter 3

Requirements

3.1 Executive Summary

The main objective was to produce a proof of concept of the efficiency of using caches over a memory system on FPGAs for high-performance engineering (HPE):

1. **Proof of concept of cache implementation on FPGAs**

   A simplified LOB must be implemented, simulated and compiled in SystemVerilog. Time and resource analysis tools allow the author of this project to study how the simulated cache is accessed compared to the on-chip memory.

2. **Coverage Analysis**

   Both models – memory and cache – must have a defined behaviour under critical circumstances. Coverage analysis is performed to ensure that the LOB will never behave erratically.

3. **Final Year Project report**

   The results of the experiments and tests must be recorded to perform a critical evaluation.

3.2 Specific requirements for this project

LOBs are complex and large systems that cannot be studied in a year. This project was proposed by the Morgan Stanley high-performance computing (HPC) team, as a proof of concept to improve an existing model. The evaluation of the success of caching over using DPBRAMs will be used to motivate a new investment got the existing FPGA model. All assumptions and simplifications will be made keeping in mind the existing structure. One success metric will be to ensure that the implemented simplified model is as similar as possible to the one used by this company.

3.3 Projects objectives overview

Table 3.1 displays the list of requirements for the success of the project.

Software Simulation:

1. Baseline model
2. LOB in SystemVerilog
3. FPGA BRAM memory modules must be faithful to reality.
4. Cache-like behaviour implementation in SystemVerilog
5. Testench and constrained random verification for coverage.
6. Behavioural and functional verification

Analytical Work:

1. Number of orders sent/received model comparison under critical, common and ideal trading circumstances (repeated orders for few clients)
2. Performance comparison: code coverage, functional coverage.

Deliverable	**Chapter**	**Description**	*Done?*
Baseline model.	2&4	SystemVerilog implementation and modelisation of the existing LOB trading on the FPGA board. The model processes incoming orders from the *traders* in the format $client_{id}, amount(\$), quantity$, as well as cancelled order in the same format. Each incoming order must be checked and processed .	✔
Testbench	6	SystemVerilog testbench for the model with a driver and generator to allow constrained random testing of each variables. This improved the certainty of the model efficiency and ability to perform its tasks.	✔
Cache model.	2&4	Variation of the baseline model with a simulated cache to store a minimum of one cache line. Instead of the usual 3+4 clock cycles explained in section 2, the expected speed is a single cycle cache read and write.	✔
Performance and time comparison.	7	The focus of this project is *time* speed up over resource utilisation. Models are compared with this metric, yet performance from resource utilisation may also be interesting to look at.	✔
2-entries or more cache models.	5	Variation of the baseline model with different cache structures, to compare and contrast cache models depending on the market data. The performance varies depending on how often the same orders are repeatedly sent, as well as the overall variance and volume of orders (greater volume and variance require a bigger cache).	✔
Cache types	4&5	Comparing cache policies and sizes.	✔
Formal verification	6&7	Thorough behaviour check with Formal verification is an added value to ensure that all input combinations are well behaved.	✘

Table 3.1: Deliverables and their importance for the completion of the project.

Project completion	*Project satisfaction*	*Project successful*
Necessary to implement	Desirable to implement	Optional to implement

Table 3.2: Legend for table 3.1

3.4 Deliverable: Software Simulation

3.4.1 Realistic Baseline Model

The first requirement shown in table 3.1, was to implement a satisfactory risk check model. Because writing a full LOB in SystemVerilog would have been too time-consuming, this project focused on a subset of the system: the FPGA risk checking and order cancellation modules. Morgan Stanley uses the following modules to implement them:

1. An upstream processor to check incoming new orders. It connects to the exchange and internal CPU, performs risk checks and forwards or drops the orders accordingly.
2. A downstream processor to cancel orders. It creates UDP packets and forwards them to the market Exchange.
3. All past orders and maximum to trade for each client are stored in a continuous BRAM along with other information. Cancelled orders are stored in another memory block to facilitate simultaneous read/write

The project's success was judged on the model's simplifications, assumptions, and adherence to Morgan Stanley's trading FPGA model.

3.4.2 Trading Simulation

Comprehensive testing

Before the model could be used as a baseline to build caches on, it has to undergo extensive testing. Under a small set of constraints, unit-testing ensured that each module behaved as expected.

Constrained-Random Testing was used to simulate critical situations. The rate at which orders were sent or cancelled had to correspond to real-world data. In addition, test data need to be comparable to the traded orders.

Because the real model has access to trillions of dollars, a minor error or misbehaviour is not tolerated by banks. Proving that this model is error-free and that all possible input combinations result in a defined behaviour was a key requirement.

Faithful data

One of the most significant limitations when testing LOBs is confidentiality. Access to trading data is severely restricted. It gives one the advantage of knowing previous market fluctuations and the orders that correspond to them. A requirement for a satisfying trading simulation is to create a realistic subset of orders and cancellations for different clients that are sent at a given time interval.

Storing orders

For the model to be transferable to reality, the implemented cache needed to use a well-justified cache protocol in SystemVerilog. While the MESI protocol has been mentioned by the HPE team of Morgan Stanley, the design choice was up to the point where it must make a difference in comparison to memory, because on-chip memory already eliminates the typical hardware/-software data fetching delay.

Cache Design

The cache models were compared to the baseline model. For this comparison to be relevant, the design must be completely similar to the memory model.

3.5 Ethical, Legal and Safety Considerations

Ethical Considerations

The use of a highly efficient algorithmic trading model may raise ethical concerns. The model can compensate for human biases that can harm financial markets and traders by handling a large volume of trade.

These appear to raise some concerns. Because public APIs provide access to brokers, markets, and the exchange, the ability to invest in better technology may be rewarded by better models, resulting in greater profit. The benefit derived from this investment, however, is quantitative rather than qualitative. Trading necessitates complex decision-making that is extremely challenging to implement in hardware today.

Ethical	**Unethical**
Liquidity: its trades make up over 70% of the total trades in a day.	Unfair advantage: machine learning models or computers can influence the market. This is a rising issue as there were several cases when bitcoins and NFT were heavily influenced by banks. HFT models heavily invest in a single platform to lower the price of a coin and stop anyone else from trading it - monopolising this market.
Profitability: the return of trades outweighs the cost of implementation [51]	Small investors: Research in HFT is particularly costly. Instead of solely relying on investment skills, trading now requires resources to gain enough power and supplies to create an efficient and fast model that can maintain a connection with the exchange.
Reduced costs: the tick size was reduced as a result of HFT's popularity. Smaller price differences allow traders to split their trades and diversify the risk of a single trade.	Software error risks. An error in the program or attack on the model can result in a cascading effect, as it was the case during the may flash crash [1].
Market efficiency: software can find any discrepancy and make the best use of them to generate profit - removing this discrepancy from the rest of the market. All remaining trades have a small difference between the bid and ask price.	

Table 3.3: Ethical considerations in HFT. Summarised from Seven Pillars website [52].

“ Ostensibly HFT users do not intend to deceive. Instead, HFT aims to give its users a

competitive advantage. Yet, HFT strategies affect more parties than just HFT traders – small investors, large trading firms, analysts, brokers, and other market participants may also be affected. (...). The mere existence of the opportunity to misuse HFT does not argue for the activity's immorality. HFT also provides benefits by way of liquidity, jobs, and market efficiency. The opportunity for misuse and its damaging consequences may argue, however, for the intense scrutiny and policing of HFT" [52].

This project also follows the recommendation to reduce the potential abuse of HFT [53]. The Markets in Financial Instruments Regulation imposes:

1. Acquiring authorization to continue to use HFT techniques;
2. Storing time-sequenced records of algorithmic trading systems and trading algorithms for the past five years, and opening these up to government monitors;
3. Implementing risk control mechanisms to ensure that trading systems are resilient and have enough capacity to prevent sending erroneous orders or contributing to a disorderly market.

The added value of this project is to increase the number of orders rather than their quality. Data is stored for a maximum of 4 years and the project's main aim is to build resilient risk control in hardware. The opportunity for misuse is limited – no trader has access to the model which is intended to supplement, rather than replace, the existing trading engine. At this level of complexity, ethical risks are very limited.

Legal Considerations

Morgan Stanley's algorithmic trading team devised and proposed this project. Investment banks' politics and profits are built on highly protected research models, smart investments, and firm-specific decisions based on private data. As a result, the model had to be simplified to fit the scope of a final-year project. It was not possible to copy or access the existing LOB SystemVerilog codes because they are protected by law and are complex for the scope of this

project. Any snippet of code, document, or data from the company is considered confidential, and thus may be subject to legal intervention. Access to any data is restricted under the Global Data Protection Policy. All assumptions and models were therefore kept general to any company and standard algorithmic-trading structures. The testbench data used public market data and was standardised for this project.

Safety Requirements

This project only implements the Proof of Concept, all work was therefore completed on software rather than hardware.

The IEEE Standards for Software Safety were followed throughout the whole project development. The following elements have been considered for the submitted code:

1. Software Life Cycle
2. Documentation
3. Software Safety Program Records
4. Software Verification and Validation Activities
5. Previously Developed or Purchased Software
6. Software Safety Requirements Analysis
7. Software Safety Design, Code, Test and Change Analysis
8. Operation Support, Monitoring
9. Maintenance
10. Plan Approval
11. Development and Installation

Based on IEEE Standard 1228–1994; used with permission [54].

Chapter 4

Analysis and Design

4.1 General design choices

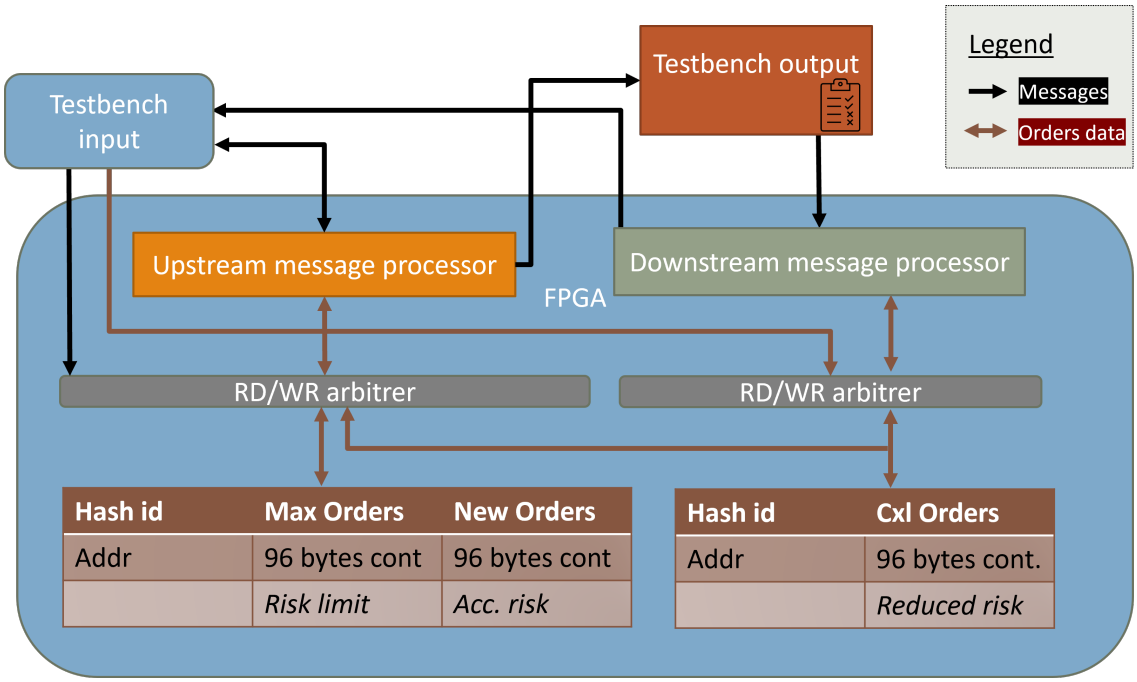
Figures 4.1 and 4.2 show the designed models for memory and cache, respectively based in figures 2.7a and 2.7b. The mapping between real and test designs was straightforward. In the latter, data was driven by a testbench rather than the exchange and CPU. As a result, instead of going through the downstream processor, the testbench could read the memory output signals directly. Furthermore, the 128-byte data from the reference model was changed to 8 bytes for the model. Each read and write request to the memory was subject to a constant delay in the model instead of the hardware delay in reality.

4.1.1 Simplifications and Assumptions

Simplifications to the baseline memory model were made for the design to be easier to implement:

1. Software cannot simulate the hardware delays caused by the bus size. Testing and comparing delays were critical to the project, a read request incurs a penalty of 3 clock cycles,

Figure 4.1: Memory model after applying design choices. See figure 5.1 for the model before simplifications.



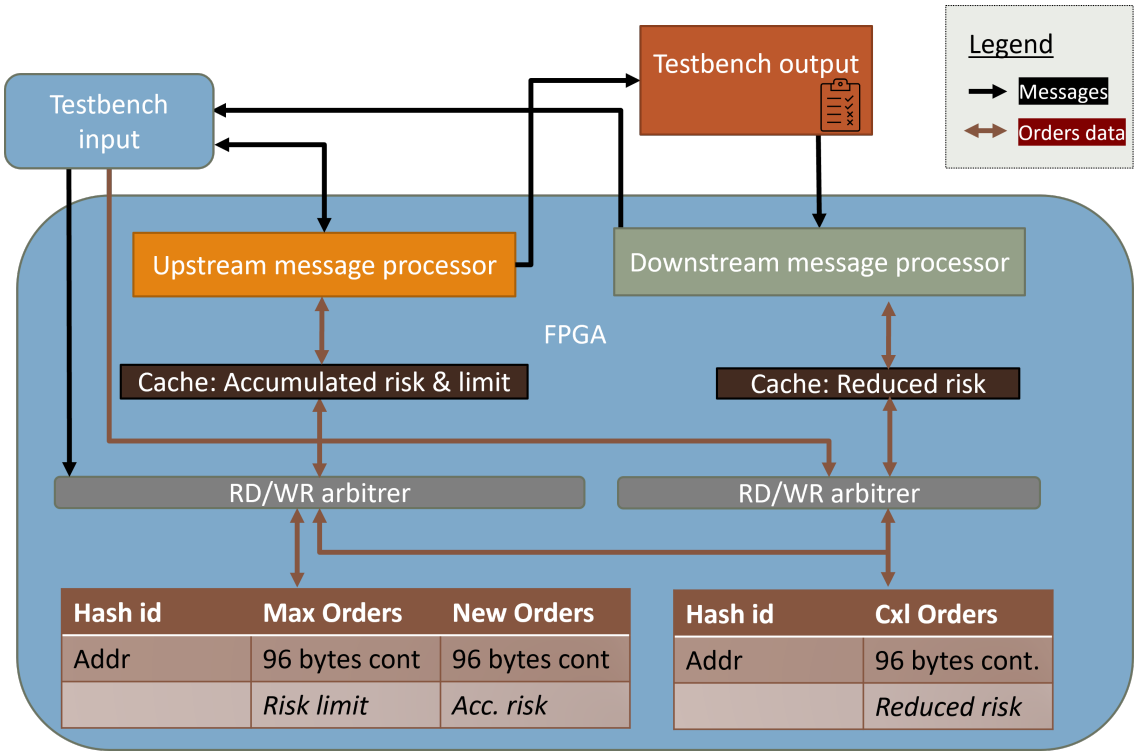
and a write request incurs a penalty of 4 cycles for a total of $3+4=7$ clock cycles. These values are an average of the data collected by the HPC team at Morgan Stanley [55].

2. In reality, the client address is mapped to its corresponding memory index using an extraneous hash table. Because this mapping does not influence the difference between caching and not caching, the cost of developing a hash table was deemed too time-consuming. Instead, the memory address was considered to be the `client_id`. This reduced the chance of mapping errors while debugging.

3. On model 2.7a, cancelled order packets are all delivered from the exchange; new orders arrive from the CPU. To mimic this design, the testbench accepts new and cancelled orders, then generates order confirmations that correspond to TCP acknowledgments from the exchange to the CPU or from the CPU to the exchange. This standardises the model: inputs on the left-hand side drive outputs on the right-hand side of figure 4.1.

4. TCP and UDP protocols were not implemented; the testbench could run in the same directory as the main model. Implementing a competitive TCP/UDP protocol in SystemVerilog that was similar to the high-end, greatly improved one used in banks would

Figure 4.2: Caching model after applying design choices. See figure 2.7b for the cache model before simplifications.



have been a major undertaking. This part of the model was therefore put aside for the scope of this project; no delays were added.

Table 4.1 summarises the assumptions and design choices made based on the possible sources of delays and whether they were considered for this design.

The model was also designed to mimic realistic trading conditions. The points taken into consideration were as follows:

1. Any combination of inputs can occur. The model's behaviour should always be well-defined. Few input variables and a default value for each case must be declared.

2. The clock rate and throughput of the simulation model do not matter; the model is asynchronous in reality. Since the block diagram shown in figure 2.7a is a sub-part of a greater system, the clock rate for simulation depends on the model's efficiency.

3. Any speed-up is welcome, as long as the added functionality is well-tested and completely safe. Development cycles are particularly long in algorithmic trading due to thorough

testing. Models are tested in software, simulated on a 'dev' environment, and checked several times – manually and automatically – before they can be considered as possible modifications.

Table 4.1: Causes of delays within the scope of the trading model and simplifications.

Delay type	Delay Source	Considered?	Justification
Hardware	Memory Read	3 cycles	Reading the order quantity and value from the block RAMs causes hardware delays due to the large data size and floating-point precision. The floating-point corresponding to the cumulative number of orders or the maximum amount to trade was fetched and combined in a single register.
Hardware	Memory Write	4 cycles	The memory addresses are not aligned, the addresses at which each of the indexes of the destination value have to be written require extra computation hence delay. Furthermore, the registers cannot write simultaneously to all 4 addresses physically.
Connection	TCP handshake	✖	Trades begin at the start of the day. A two-way handshake is used to establish the TCP connection. For each client, the `maximum_to_trade` is stored in memory. Traders must wait for the connection and memory write operation, which causes a delay. Because memory is still required, caching does not affect the delay.
Connection	Sending orders	Constant	New orders are sent via TCP connection. The exchange always sends back an acknowledgement message to confirm the order was written. This handshake adds complexity to the upstream FPGA processor. The model must wait for the memory or cache write to occur, just like it waits for the exchange in reality.

Table 4.1: delays in the model – continued on next page

Table 4.1: Continued from the previous page

Delay type	Delay Source	Considered?	Justification
Connection	TCP Headers	✗	In reality, the data must be formatted, TCP headers are added to the orders before sending them to the exchange. This was not part of the design. The potential delay caused by adding headers is lower than one clock cycle, hence too small to be simulated in software. Whilst the cache could store the TCP headers in a separate field, this would diminish its efficiency. A greater cache size may cause the same delay as the memory when fetching data.
Data	Incoming orders	✗	The CPU issues new orders. Traders or computers make decisions, validated by the FPGA. Moving data between the platforms [8] causes a small delay. Because this project focuses solely on the distinction between memory and cache on the FPGA, this delay was ignored; it has no impact.
Structural	Sub-system	✗	The design in figure 4.1 is only part of the FPGA design used by the algorithmic trading team in Morgan Stanley. This involves more connections with other modules and greater complexity. Any delays caused by repercussions on other modules were disregarded as this could not be tested without the real model.
Connection	Two-way Ack	✗	Cancel orders come from the CPU, to the exchange and back to the CPU. This back and forth data exchange was not simulated in the model. The pipeline of activity may be influenced when the downstream processor waits for the exchange. Since only one testbench was needed to assess memory usage and speed, the two-way exchange was replaced by a confirmation that the order was written in memory.

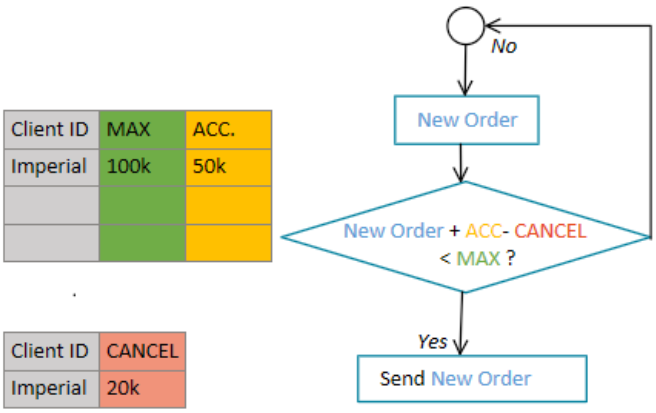
Table 4.1: End of delays causes and justification table from the previous page.

Definition of improvement

As well as predefined assumptions, some design considerations were made in order to improve the model. Improvement was defined as changing the baseline model towards a version which was more similar to reality than the one described in chapter 4. Alternatively, a speed-up on this model that enables a faster or easier use of caches is also an improvement.

4.1.2 Risk Checking

Figure 4.3: Risk check flowchart.

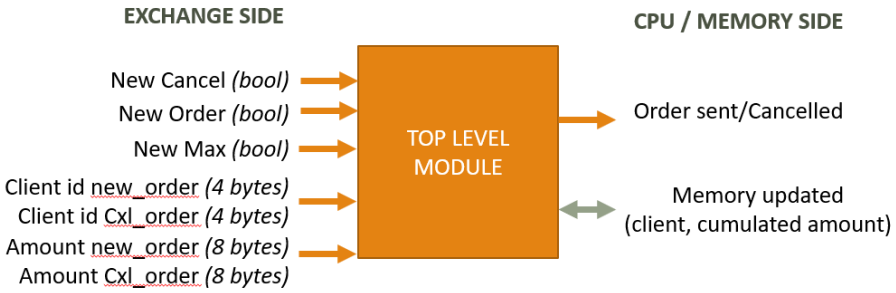


Basic risk checks were performed by comparing the sent orders, cancelled orders, and the maximum amount to trade allowed per client following the equation 4.1.

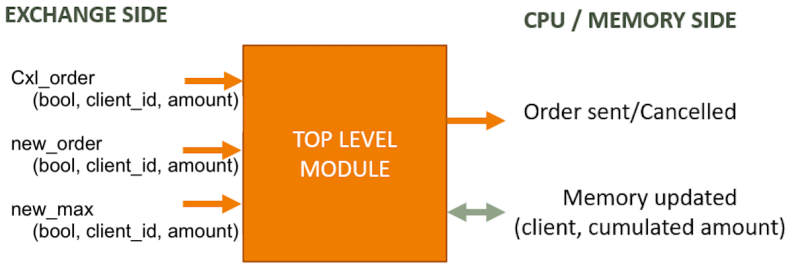
$$Max_{order} > Accumulated_{order} + New_{order} - Cancelled_{order}? \, Send_{order} = 1 : 0 \tag{4.1}$$

In reality, the `maximum_to_trade` is set each day by traders or computers for all firms' clients. This ensures that the order book is balanced; the algorithm will not take too much risk by investing most of its value in a single share that is performing well this day. This value is always updated after a request from the traders. In the model, the new maximum was only written if its value was greater than the current amount traded so far: `accumulated_orders - cancelled_orders`. This guaranteed that risk checking was never affected by an update maximum operation.

4.2 Data type design



(a) Integer I/O model



(b) Packed structure I/O model

Figure 4.4: Different data types considered for the trading model, block diagrams

Control bits were used in conjunction with data bits to simplify the design. For example, the upstream processor was driven by `exchange_go` and `CPU_go` signals in the model. Data bits of the exchange amount, `Client_ID` and corresponding CPU data signals were continuously sent and received.

A more complex implementation employed SystemVerilog data types to merge data and control bits in packed structures, shown in listing 4.1. Instead of multiple inputs, the model was designed to be driven by packed structures from figure 4.4b . The inputs were changed to `exchange_data` and `CPU_data`, allowing for a more realistic block diagram. The control and data bits for the `client_ID`, as well as the amount to send, were stored in each packed structure as shown in the listing below. To be implemented, this model required an extraneous step of unpacking each structure and sending packed data from the memory instead of raw data.

Listing 4.1: Example of packed structure in SystemVerilog.

Criteria	Control & data bits	Packed structure
= True to reality	InputOutput block diagram is different from reality due to extra control bits. Inter-module behaviour is slightly more similar; the data is easily accessible and memory blocks output simple integers.	InputOutput block diagram is more similar to reality as there is the same number of inputs and outputs on each side. Data is easily accessible and logically grouped.
Speed	May be faster: less operations must be performed. However, the extraneous bits will require more connections which could cancel off the time saving described.	Potentially the same as for simpler data types.
Logic	Logic is easy to design and analyse: each variable has a distinct signal that can be tracked in Questasim.	Handling structures is more delicate than simple data types in SystemVerilog; the logic may be slightly more complicated.
Storing data	Building cache and memory protocols for read and write is messy. Memory modules need extra signals to request readwrite permissions; the caching model requires even more control bits.	Implementing the memory and cache is easier; data bits are self-contained.

Table 4.2: Comparison of using data and control bits versus packed SystemVerilog structure for design coherence.

```
struct { // Structures -> a collection of variables of different data types
    byte    val1;
    int     val2;
    string  val3;
} struct_name;
```

Except for memory and cache design on the last row, both designs appeared similar when compared using table 4.2. Data types as packed structures allowed for a well-contained, logical code that is easier to read, as shown in the code from chapter 5. As a result, structure data types were chosen.

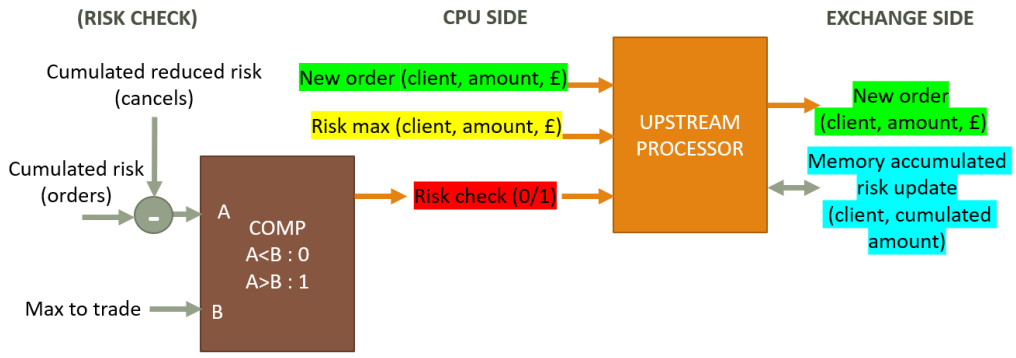
It is crucial for the design that the downstream processor always holds the most recent value of the accumulated cancels and writes them to a cache or memory. In the case where the upstream processor reads this line and the value is too small, an order may be deemed safe when not. The use of packed structures guaranteed that no step was skipped; the processor could not process a new order before receiving the confirmation of a write operation to the downstream

memory.

4.3 Logic design

4.3.1 Logic blocks

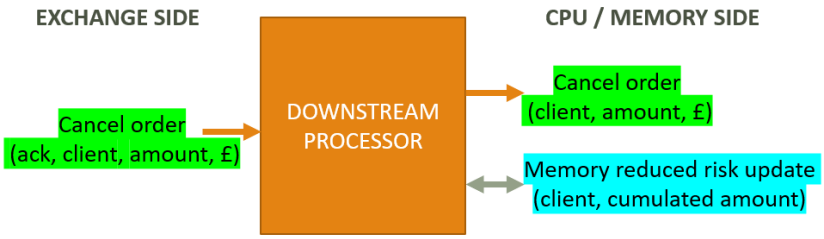
Figure 4.5: Upstream processor block diagram.



Any arithmetic or logic check related to sending orders is managed by the upstream processor in figure 4.5. The first module – in brown – computes risk checking. If the order is safe to send (the summation of the `new_order` and `accumulated_amount` is under the threshold `max_to_trade`), the main module writes in memory the new accumulated trade value and waits for a memory write confirmation signal. Each client's `maximum_to_trade` was handled by the upstream processor in the design. It flushed the current value in memory for the `client_ID` and overwrote it with the new maximum when a maximum update request was received.

Deciding to update the maximum before or after sending an order could have influenced the results of risk checking. The logic between the two parts of this processor needed to be well-designed. The `update_max` input was always given priority in the state machine diagram from 4.8.

Figure 4.6: Downstream processor block diagram.



On the other hand, the downstream processor was designed to handle cancel operations only. Upon reception of a cancel order packed structure, it appends the amount in memory for the requested `Client_ID`, and waits for confirmation.

The priority scheme between the three operations described above – `new_order`, `update_max`, `cancel_order` – was set in the top-level module to process all possible input combinations. In the unfortunate case of receiving a `cancel_order`, `new_order` and `update_max`for a singular client, safety was defined as:

1. Performing risk checking by reading the previous values of `cancelled_orders`, `accumulated_orders` and `maximum_to_trade`. Comparing these values with the incoming `new_order`.

2. Updating the `maximum_to_trade` with the input `update_max` to hold the latest value. Simultaneously, the order was cancelled since these access different memory blocks and do not overlap.

3. Once risk checking was done – before or after step 2. – the `new_order` was sent or kept if it failed risk check.

4.3.2 Finite-state machines

For better structural integrity and to keep variables distinct from the design's behaviour description, both of the processors were implemented as FSM. These modules are well-supported

in SystemVerilog. Since FPGAs are made up of flip-flops and registers, any sequential circuits can be easily built.

The two processors' state machines are uncorrelated as shown in figures 4.7 and 4.8. Both FSMs are triggered on a change of variable or rising clock edge, which is convenient but may restrict speed-up and limit malleability.

Figure 4.7: Downstream processor state machine.

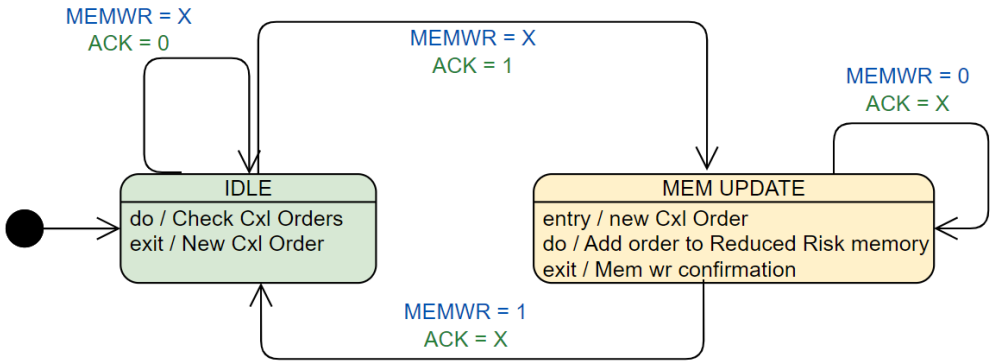
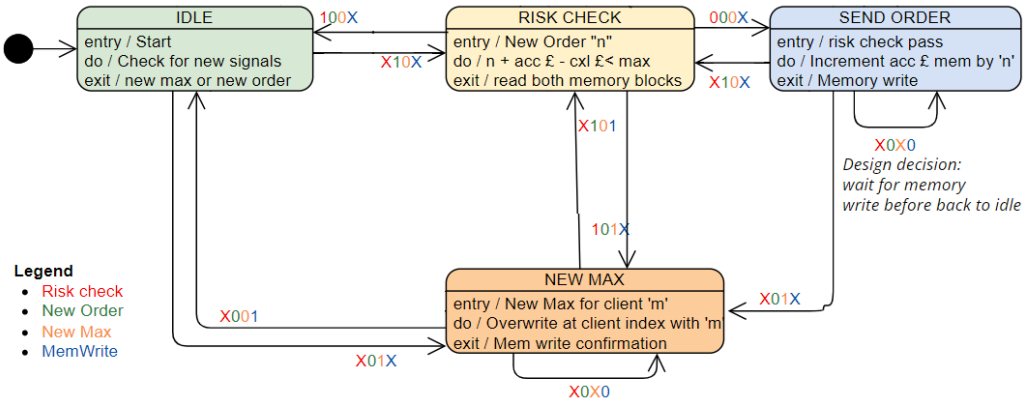


Figure 4.8: Upstream processor state machine.



The design decision between a Moore and Mealy state machine held onto the chosen value for the clock cycle. A high-frequency model benefited from Moore's stability without suffering from possible delays; a low-frequency clocked model had a high computation rate per clock cycle as it went through different states at once, as shown in table 2.3.

A good compromise involved a mix between Moore and Mealy. Some states such as risk checking

and sending orders were allowed to change during a clock cycle; reset signals and returning to idle were both performed at the clock rising edge.

4.3.3 Removing states

Initially, the upstream state machine in figure 4.9a was designed with combined states in order to process several orders at the same time.

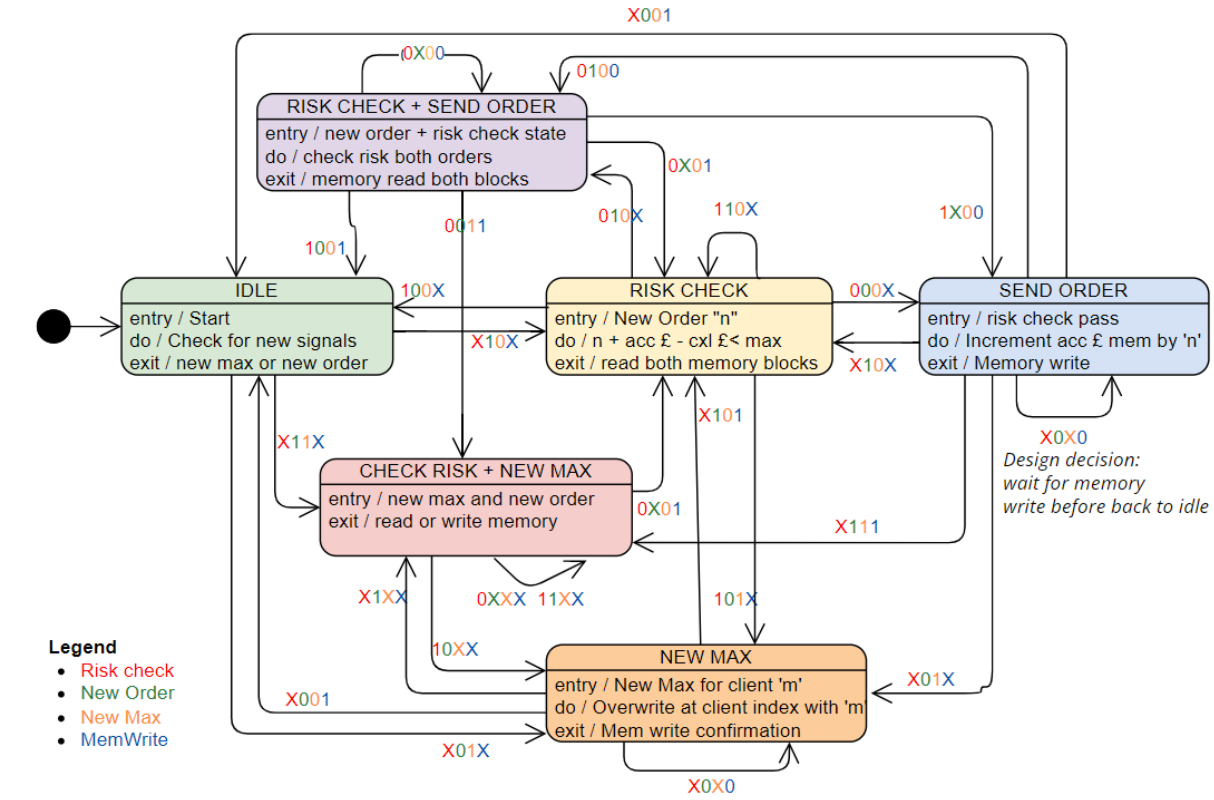
Depending on the design choice, orders may have arrived at a greater frequency than the clock frequency. In this situation, the states `RISK_CHECK_SEND_ORDER` and `RISK_CHECK_NEW_MAX` may have allowed processing two orders simultaneously. State-transition tables can be found in Appendix 10.8. For security purposes, the model can only be in one state at a time; the combined states could allow moving onto the next state for the current order – `SEND_ORDER` – and directly move to risk checking without waiting for confirmation that the order was written in memory. The timing diagrams 4.10b and 4.11a display this case for the models for the extreme case of a difference in clock frequency of factor 2.

Similarly, combining `CHECK_RISK` and `NEW_MAX` would let the processor computes the risk limit for an order at the same time as writing the maximum for a different client. This way, less information would have been lost and there was no need to prioritise between orders.

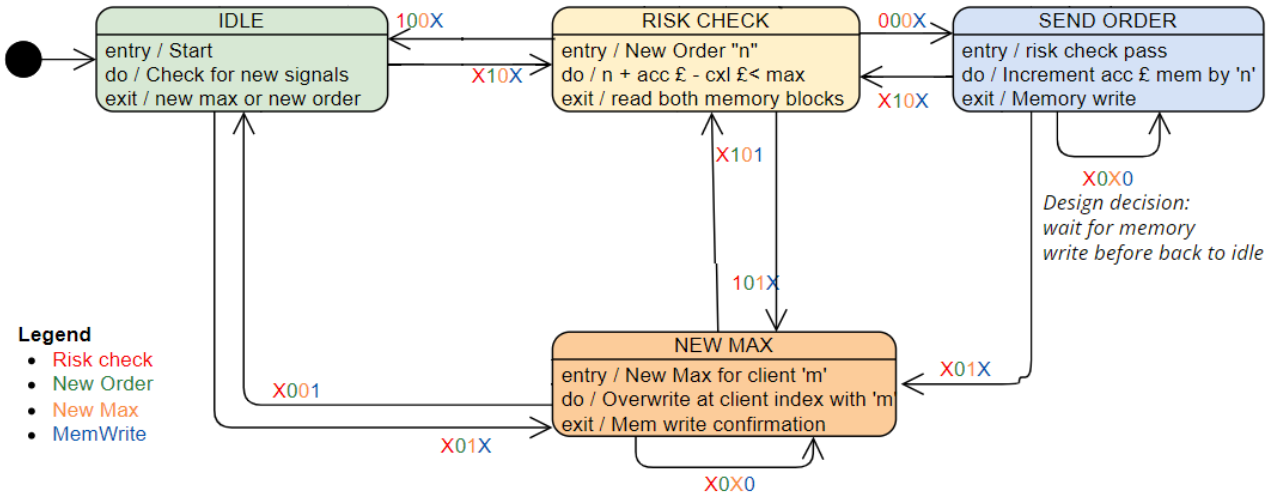
On the flip side, this design caused the processor to take longer to return to `IDLE`. On timing diagram 4.10a, the processor does not prioritise correctly the maximum to update before sending the orders. Figure 4.11b also suggests that the simpler model was more efficient when the clock rate was increased.

After implementing both models, the following points were considered regarding the design of the upstream processor:

1. The purpose of having simultaneous states: the real model does not have combined states.
2. Where is the limit of using combined states? If the processor were to perform two operations simultaneously, it would be possible to combine all states in one and always perform

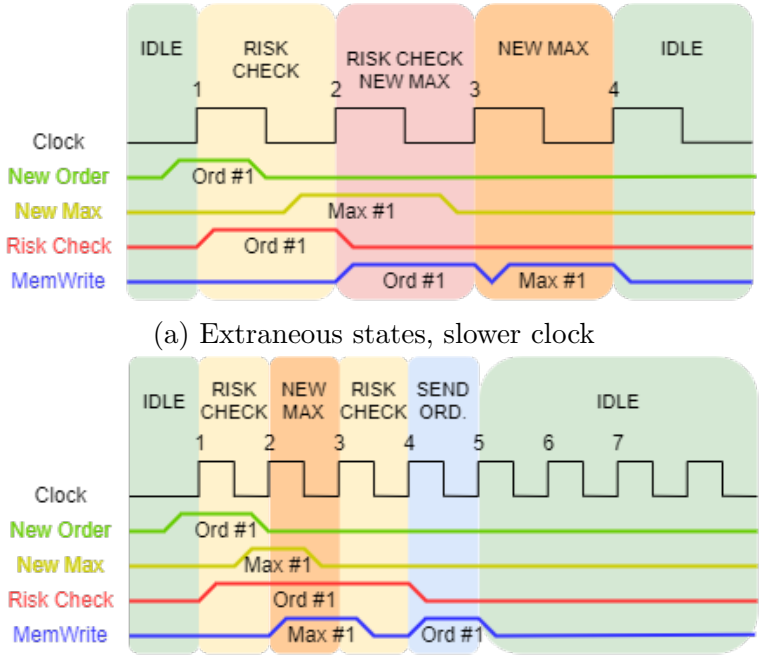


(a) Extraneous states for speedup



(b) Simplified state machine

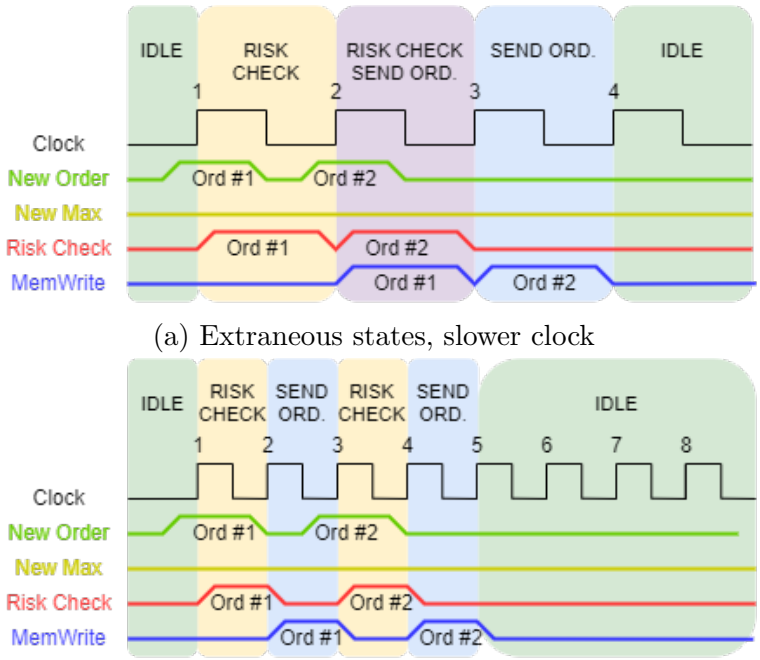
Figure 4.9: Upstream state machine state diagram comparison between two models.



(a) Extraneous states, slower clock

(b) Simplified state machine, faster clock

Figure 4.10: Upstream state machine state diagram comparison between two models for the case of **two consecutive new orders**.



(a) Extraneous states, slower clock

(b) Simplified state machine, faster clock

Figure 4.11: Upstream state machine state diagram comparison between two models for the case of a **consecutive maximum update and new order**.

all operations. Whilst this may speed up slightly the model, it would make it less realistic and limit testing accuracy.

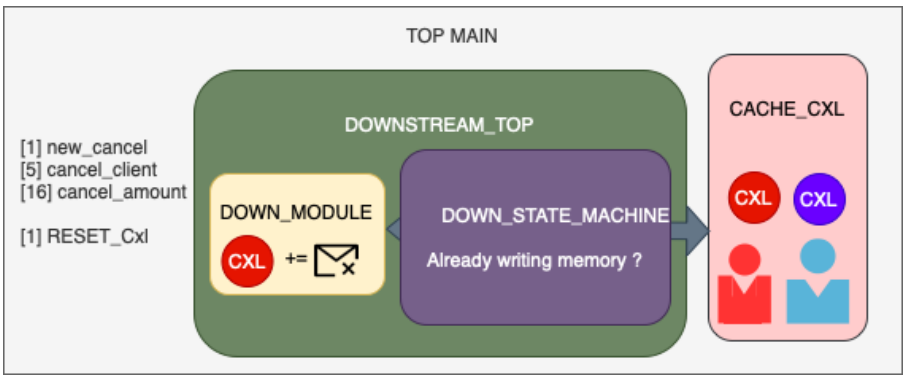
3. The choice of the clock rate was also questioned. The upstream processor has the most operations per cycle. It is the module that sets the bounds of the clock frequency. Simplifying the model allows for a higher frequency, hence the orders would still be processed at the same speed with a simpler model.

Due to these considerations and the results of basic time analysis on the two models, the intermediate states were removed. They offered little to no speedup, as shown in figure 4.9b.

4.3.4 Removing FSM

After careful consideration, the downstream FSM from design 4.6 was removed. In this design, the processor is driven by one input that contains two data variables and one control bit. Since this block merely just appends the new amount to the correct line of the reduced memory risk, a finite state machine was not the best-suited tool to implement it.

The FSM offers stability; the processor either can be in a `memory read`, `memory write` or `idle` stage. New states can easily be added by updating the lookup tables, adding a new transition method and a case statement in the SystemVerilog code. Instead of the FSM, a concise module with sequential read and a write, enabled by the read data or confirmation signal was more efficient. This economised hardware usage; the FSM uses more flip-flops and registers to hold values than a simpler design. Removing the FSM on the downstream processor saved static memory, which is in line with the algorithmic trading goal of optimising model efficiency – this static memory can be used by other modules.



(a) FSM downstream

Figure 4.12: Downstream processor block diagram. Visualising the tasks performed by this processor. The FSM is indicated as a purple box.

4.4 Memory design

Accumulated memory

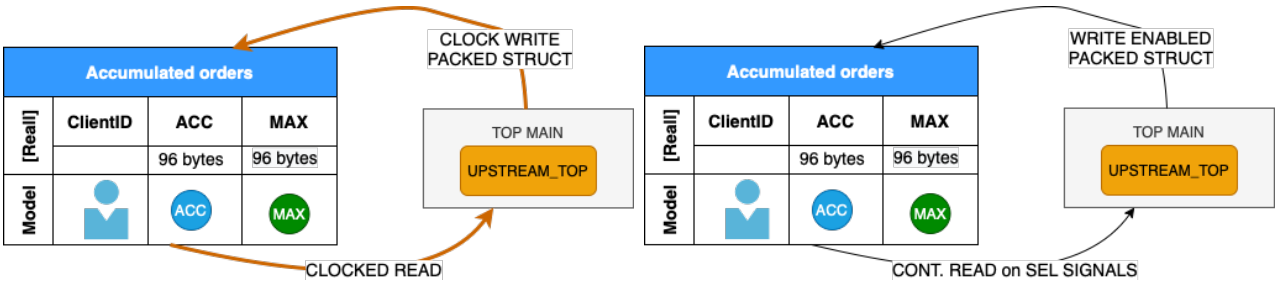
Both the `accumulated_orders` orders and `cancelled_orders` write operation consisted in adding the new value to the amount already in memory. This could have been done by the processors which would have to read the memory, then add and write back.

A design choice was made between implementing this addition within the processor or memory module. For simplicity and speed, the memory write was defined as a memory add for the orders, and kept the same for the `update_max`.

Synchronous versus Asynchronous

Although the memory controller can be connected to an FPGA, the design of such a device is still simplified compared to the real trading model. The BRAM supports asynchronous read and write, but the trading model does not specify if a separate memory controller exists. In the implemented model, some of the controller's tasks were done by the upstream and downstream processors, who requested and ensure that all memory operations were well-defined and non-concurrent.

On the one hand, synchronous memory facilitates risk assessment by preventing memory access



(a) Clocked Memory (b) Asynchronous Memory

Figure 4.13: Different memory modules for the trading model

outside of clock rising and falling edges. Cache speedup is limited by the clock latency (the minimal throughput is based on one clock cycle to read + operations), it could therefore limit testing abilities.

Synchronous memory was safer to implement than asynchronous; the considerations here were based on chapter 2. Inputs and outputs of the real model are well-defined at all times, the situation was well-suited for algorithmic trading where an error is not permitted. Most memory-CPU models use a synchronised memory. However, a slow clock would have resulted in longer memory access latency [56].

Speed up with an asynchronous model

The trading model used by banks is mostly asynchronous. This allows for great speed; operations can be performed simultaneously. In figure 2.10b, tasks can be performed during the same clock cycle. In like manner to a clocked CPU, the latency of a single task may not have been affected, but the overall throughput was improved. This is crucial for traders who want to send a high quantity of orders at the same time.

4.5 Cache design

Based on chapter 2, write allocate offers better performance when data was written to the same cache repeatedly. The cache design therefore implemented this method for handling

write-misses. An example flowchart of write-back using write allocate is displayed in Appendix 10.4.

A secure trading model could only be implemented with a good cache coherency protocol to ensure it always returned the correct value. The cache processor implemented an invalidate protocol by acquiring bus access and broadcasting the address to be invalidated. The two processors snooped on the bus and invalidated any address matching the broadcasted one.

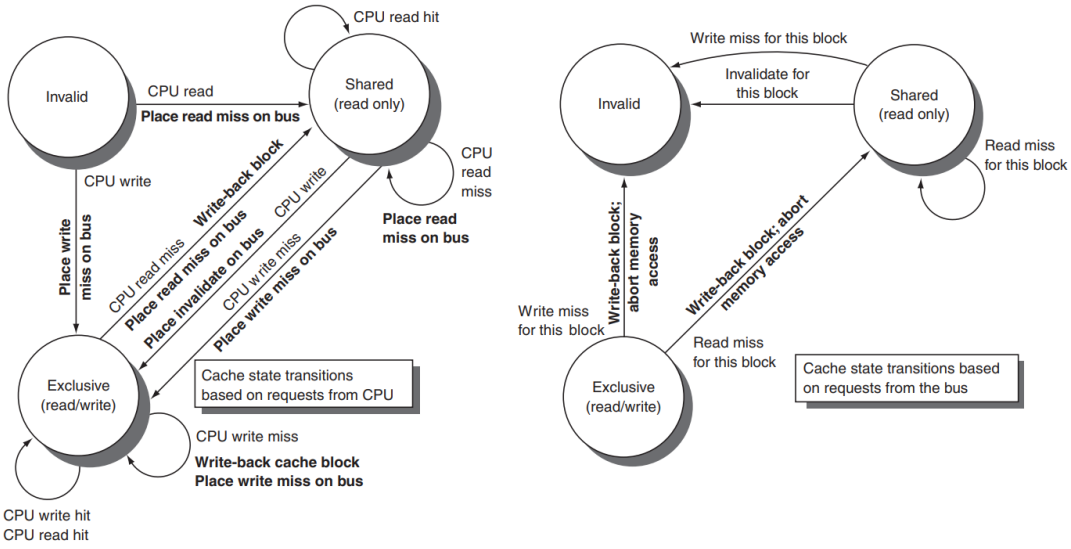
There was no need for a snooping protocol to handle simultaneous write. Each processor can only write to one cache [57]. In this design, cache tags handle the snooping protocol and the valid bit triggers the invalidation of each cache line. Read misses are handled by the snooping protocol. Write misses are similar; there is no need to send the write operation if another processor holds the correct value of the cache line. This avoids the delay caused by a read and write operation to memory.

An extra bit was added to each cache block to track if it is shared. For example, a new order and new cancel could try to read from the same cache block from different processors. Once the block is invalid, the state of the owner's cache block changes from shared to exclusive and no processor can access it until it changes back. There was no need for the Exclusive state here since there were only two processors who cannot write simultaneously to the same address.

Figure 4.14 displays the logic for each state and transition. Any access that requires permission from the processor is shown in parentheses under the name of the state. Bus actions are generated from the state transitions shown on the arrows. The processors' actions trigger state transitions shown on the left-hand side diagram whilst the bus operations are shown on the right-hand side diagram.

Cache size could easily be managed within the cache module by changing the `TAG_MAX` variable. This facilitated the testing process as seen in chapter 6. The cache layout was unchanged for the scope of this project. The goal being to compare caching to a baseline model, block layout was not tested here and can be studied as future work.

Figure 4.14: FSM for the cache coherency with state transition. The variable names are the same as in the code. Source: Computer Organisation and Design, D. Patterson and Henessy [33]



4.6 Timing design

Race conditions

Due to the clock cycle penalties from the memory module, some upcoming orders sent by the testbench (at every clock rising edge) had to be dropped. There were race conditions between each of the processors writing the current order to the upstream/downstream memory block and the upcoming orders holding different amounts and client indexes. Proper timing and sequential logic prevented any confusion between the current order and the following ones when the order was processed.

Furthermore, simultaneous read and write risks were studied to guarantee no concurrency between the processors.

	Maximum to trade	Accumulated Orders	Cancelled Orders
Maximum to trade	✓	✎	✎
Accumulated Orders	✎	✓	✎
Cancelled orders	✎	✎	✓

Table 4.3: Simultaneous write risks, ✎=simultaneous write.
Red indicates that the block has to be locked, green shows that the data can be written safely

	Maximum to trade	Accumulated Orders	Cancelled Orders
Maximum to trade	✓	👓	✎👓
Accumulated Orders	👓	✓	✎ 👓
Cancelled orders	✎👓	✎👓	✓

Table 4.4: Simultaneous read and read write risks, 👓=simultaneous read
Red indicates that the block has to be locked, orange shows that both items will be read/written at the same time; they share the same address in memory

Simultaneous write requests

The double-entry table 4.3 shows that there is only one problematic case of simultaneous write operations. Since only the downstream processor can write to `cancelled_orders`, it can always be overwritten regardless of the current state each of the processors is in. Writing a cancelled order is always prioritised over reading operations.

Simultaneous write of `accumulated_orders` and `maximum_to_trade` was more complex to handle.

Accumulated orders and `maximum_to_trade` shared the same index in this design. From a memory perspective, the whole block was written in memory if any of one is changed. On a cache level, the two values could be written separately. However, this was considered unsafe since it introduced potential errors when performing risk checking. Due to the high rate of upcoming orders, the sequential orders may have been mistaken as simultaneous. The final design decision was to always prioritise the new maximum.

Simultaneous read requests

Whilst less likely to cause major errors, simultaneous reads were more likely to generate minor problems on table 4.4. A simultaneous read to the same index line from the upstream block could have been problematic if only part of the block was dirty. For example, if the

`maximum_to_trade` was changed but not the `accumulated_orders`, the entire line may have been discarded when the new value is written. This was considered when implementing the snooping protocol for the cache.

Simultaneous read and write A greater concern arose when the downstream processor tried to write the `cancelled_orders` at the same time as the upstream processor was checking risk and reading this value. In this situation, the snooping protocol must give priority to the write operation before performing a read action.

The following rules could be deducted from the concerns above:

1. Writing always has a greater priority than reading when the two actions conflict within the same cache line.
2. Writing `maximum_to_trade` is always prioritised to writing `accumulated_orders`. An extraneous check must be performed to ensure that the maximum is greater than the current amount traded so far.
3. There may be a possibility to set a `dirty` flag on only part of the data from the upstream processor in order to preserve the data that has not been changed.

A cache controller with a snooping protocol such as MESI was able to handle such distinctions.

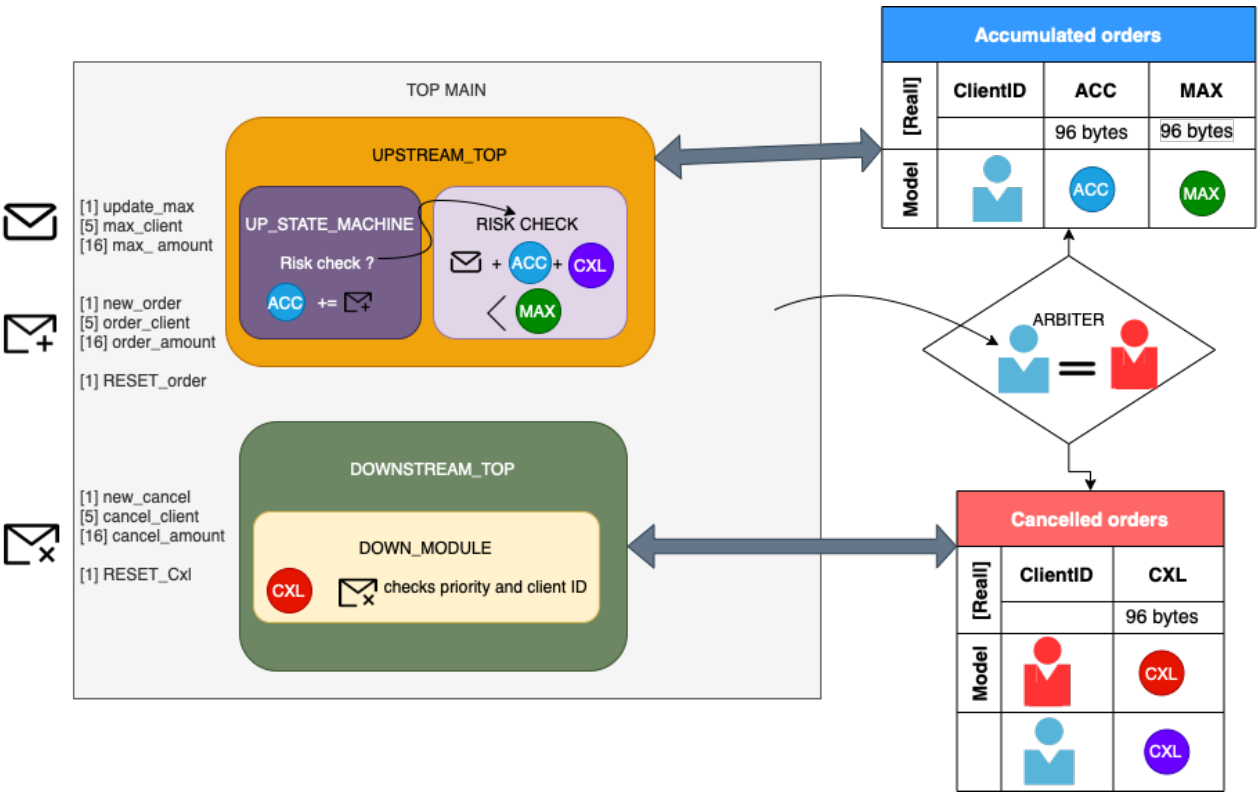
Chapter 5

Implementation

5.1 General implementation

The baseline model was implemented following diagram 5.1. Each box in this figure corresponds to a SystemVerilog module.

Figure 5.1: Baseline Model block diagram.



The top-level module links the `upstream_processor, downstream_processor` and `downstream_RAM` or `cache` modules. Since the `upstream_memory` is only accessed and written by the `upstream_processor`, it can be declared inside the processor module. The `always` block, line 23 of listing 5.1 is triggered on a new downstream order or on the clock cycle only for the same reason.

Listing 5.1: Code snippet: logic of the top-level module.

```
1  //including all dependencies (not shown)
2  `define SAFETOTRADE  ($signed({1'b0, max_to_trade[15:0]})>
3                        $signed(accumulated_orders+ (~cancelled_orders[15:0]+1)))
4
5  module top( clk, HRESETn, cpu_client_id, cpu_amount, cpu_go, cpu_new_max,
6              exchange_client_id, exchange_amount, exchange_go, /*to display testing*/
7              cancelled_orders, accumulated_orders , max_to_trade);
8
9    // declaring input and outputs (not shown)
10   reg[31:0] cancelled_orders_reg;
11   cpu_req_type downdatareq; //CPU request input (CPU->cache)
12   mem_data_type mem_data; //memory response (memory->cache)
13   //outputs,
14   mem_req_type mem_req; //memory request (cache->memory)
15   cpu_result_type downexchrest; //cache result (cache->CPU)
16   assign cancelled_orders  = cancelled_orders_reg;
17
18 /*declare dm_cache_fsm_downstream (not shown)*/
19 /*declare downstream processor    (not shown)*/
20 /*declare upstream processor      (not shown)*/
21
22     always @(clk, downexchrest.ready) begin
23         downdatareq.valid = 1'b1;
24         downdatareq.data =exchange_amount;
25         downdatareq.rdindex[13:4]  = cpu_client_id;
26         downdatareq.rdindex[31:14]  = '0;
27         downdatareq.rdindex[3:0]  = '0;
28         cancelled_orders_reg = downexchrest.data;
29       end
30 endmodule
```

Signed arithmetic in SystemVerilog

By default, arithmetic is unsigned in SystemVerilog. For example, the comparison $-10 > 19$ returns `True` since $a = -10_{10} \rightarrow 10110_{CPL2} \rightarrow 22_{10} > 19_{10}$.

Listing 5.2: Unsigned comparison in SystemVerilog for risk checking.

```
1    mem_dataup_wr = correct_amount;
2    result = (accumulated_orders_reg+ (~cancelled_orders_reg+1) + amount );
3    pass_checks = max_to_trade_reg[15:0]>result;
4    mem_requp.we = pass_checks;
```

Since traders may cancel orders from previous days that do not appear on the order book, the result of risk checking may be negative. This is the case in listing 5.2 where the comparison between `result` and `max_to_trade` is unsigned. In this situation, the order should always be processed. The difference between the maximum to trade and the amount traded/cancelled so far is still positive.

As a result, each element of the comparison was sign-extended in listing 5.3 to obtain correct results.

Listing 5.3: Signed comparison for risk checking as implemented in the upstream processor.

```
1    mem_dataup_wr = correct_amount;
2    result = (accumulated_orders_reg+ (~cancelled_orders_reg+1) + amount );
3     //extend for neg values
4    pass_checks = $signed({1'b0, max_to_trade_reg[15:0]})>$signed(result);
5    mem_requp.we = pass_checks;
```

5.2 Data type implementation

Data structures were defined to be shared across all modules to provide coherency and logic. Each request, result, and tag type used between modules is defined by the package `cache_def`.

The `valid` bit from listing 5.4 simply compares the requested address to the memory address to ensure that a match exists. The memory and cache data type's `ready` bit will be useful in

a hardware implementation where the register holds only a partial value. When the hardware bus returns part of the floating-point decimal, the processor could have mistaken this for the entire data read from memory - `ready` is not asserted until the data is completely written/read. It also ensures that the cache holds an uncorrupted version of the item to retrieve.

Furthermore, the cache uses a `rd_index` and `wr_index` in order to mitigate the risk of conflicting read and write operations described in chapter 4 in the downstream memory block.

Listing 5.4: SystemVerilog fragment of the cache def module to define cache and memory structures.

```
typedef struct { //data structure for cache memory request
     bit [9:0]index; //10-bit index
     bit we; //write enable
 }cache_req_type;
 typedef bit [127:0]cache_data_type;//128-bit cache line data
 typedef struct { // memory request (cache controller->memory)
     bit [31:0]addr; //request byte addr
     bit [127:0]data; //128-bit request data (used when write)
     bit rw; //request type : 0 = read, 1 = write
     bit valid; //request is valid
 }mem_req_type;
 typedef struct {  // memory controller response (memory -> cache controller)
     cache_data_type data; //128-bit read back data
     bit ready; //data is ready
 }mem_data_type;
endpackage
```

The most significant bit and least significant bits are dynamically set so that different cache sizes can easily be tested without disrupting the module and influencing only the length of cache tag bits. In listing 5.5 below, the block size is $31 - 14 = 16$ bits.

Listing 5.5: SystemVerilog fragment of the cache def module to defined cache tag type.

```
package cache_def; // data structures for cache tag & data
 parameter int TAGMSB = 31; //tag msb
 parameter int TAGLSB = 14; //tag lsb
 typedef struct packed {
 bit valid; //valid bit
 bit dirty; //dirty bit
 bit [TAGMSB:TAGLSB]tag; //tag bits
 }cache_tag_type;
```

5.3 Logic implementation

5.3.1 Downstream processor

The downstream processor was defined as a sequential circuit triggered when any of `new_cancel`, `memwr` or `RESET_CXL` is asserted. At every clock cycle, it can perform one of two operations: add `cancel_amount` to the existing cancel value, read from memory or write it back to memory.

From a SystemVerilog perspective, `downstream_processor`'s logic is driven by the top-level module. The module `downstream_top` unpacks the variables and keeps track of the previous client and amount. The `downreq.we` input of the downstream state machine triggers a new write in the memory. It is computed as shown in listing 5.6. If there is a change in the `client_id` or `amount`, a new order has been received. Whilst the memory controller is sequential, the top-level modules are combinational and mostly combine the input and outputs of the existing modules.

Listing 5.6: Code snippet of the downstream module .

```
1  `include "code/shared/ramdownstream.sv"
2  `include "code/shared/cache_def.sv"
3   import cache_def::*;
4
5  module downstream_top( clk, client_id, amount, downdatareq);
6   // input and output definition (not shown here)
7    reg old_client;
8    reg old_amount;
9  always_comb
10   begin
11     downdatareq.wrindex[13:4] = client_id;
12     downdatareq.rw = ((client_id!=old_client) || (old_amount!=amount));
13     downdatareq.valid = 1'b1;
14     downdatareq.data =amount;
15     old_amount = amount;    old_client = client_id;
16     // SVA to check write enable logic (not shown here)
17 end
18 endmodule
```

Continuous cancels for one client

During testing, the client's memory line was read continuously; a new request was made every time the client changed. When orders were cancelled one after another for a single client, the processor had to

1. Read the accumulated cancelled orders (cancel #1)
2. Append the new cancel
3. Write this value in memory
4. Read the accumulated cancelled orders (cancel #2)
5. Append the new cancel
6. Write this value in memory

Steps 3 and 4 could have been avoided by using an arbiter or a cache where the value is passed directly within the module.

5.3.2 Upstream processor

Since the upstream processor performs more operations than the downstream processor, it required more logic handling.

Firstly, the FSM was implemented according to state machine diagram 4.8. The simple implementation defines the processor's behaviour and how it prioritises incoming orders. This ensures that each state was completed before returning to idle. Without the proper FSM implementation, the upstream processor could have sent a new order before the previous one was written in memory. Not only is this unsafe and unrealistic regarding the trading model, but the number of unprocessed trades also increases exponentially as time goes by, and the model would end up not processing any trades. For these reasons, it was essential for the upstream processor to wait for the read confirmation in state `MEMWR` before returning to its `IDLE` state.

Listing 5.7: Upstream state machine top-level module.

```
module upstream_processor_top(clk, client_id, amount, new_order, new_max, accumulated_orders,
max_to_trade, thenewmax);
 // input and output definition            (not shown)
 //declares RAM UPSTREAM or CACHE UPSTREAM (not shown)
 // declares upstream processor FSM        (not shown)

always @(client_id, amount)
  begin  : main_process
    accumulated_orders_reg = cpu_res.data[15:0];
    cancelled_orders_reg =  cancelled_orders;
  fork begin: read_fork // for hardware delay simulation
    cpu_req.rdindex[31:14] = '0;
    cpu_req.rdindex[13:4] = client_id[9:0];
    cpu_req.rdindex[3:0] ='0;
    cpu_req.valid = 1'b1;
    // wait until cpu res rdy
  end : read_fork
  join
  wait fork;
    max_to_trade_reg = cpu_res.data >> 16;
    if ((new_max) && (amount>(cpu_res.data[15:0])) ) // update max, shift amount 16 bits to left
      correct_amount = amount << 16;
    else
      correct_amount = amount[15:0];
    cpu_req.data = correct_amount;
    result = (accumulated_orders_reg+ (~cancelled_orders_reg+1) + amount );
    pass_checks = $signed({1'b0, max_to_trade_reg[15:0]})>$signed(result); //extend for neg values
    cpu_req.rw = pass_checks;
    wait (cpu_res.ready == 1);
  end  : main_process
  //SVAs for risk check (not shown)
endmodule
```

In the case of continuous orders for the same client, the accumulated value was first written in memory/cache, read back and the new order value was appended to memory/cache. This ensured that the memory/cache line held the latest accumulated value at all times.

Comparatively to the downstream processor, a new order was defined as a change in amount or in client input variables. As a speed-up, the risk check was directly implemented from the variables that were continuously read. Since the last `@always` block was triggered by the

sensitivity list `client_id, amount`, the variable `pass_checks` always held the correct value [58].

5.3.3 Challenges

1. In the top-level module, ensuring that the modules held the latest values as well as avoiding extraneous clock cycles was a challenge. For this reason, data was passed directly between the input and output of each processor to the memory module. Defined behaviour was guaranteed by using select signals that trigger a read or write.

2. Detecting new orders or changes in order was a difficulty. Whilst TCP or UDP packets are received in reality, a change in one of the inputs may not constitute a satisfying new order trigger. For this reason, a write enable variable was defined upon signal change of value. A new order was here defined as a change in the `amount` or `client_id`.

3. Difficulties were also faced regarding the state machine implementation. Fewer states led to a simpler model with a slower clock cycle; more states supported processing different orders simultaneously, but the added complexity was not worth any noticeable speed-up. Since the testbench continuously varied the inputs, the processor's clock cycle was the bottleneck for order throughput. After several model iterations, the current one performed the best and was chosen empirically.

4. Correct sign-extend and signed comparisons in SystemVerilog were both initially challenging to debug.

5.4 Memory implementation

Synchronous memory blocks were implemented. Read and write signals are more steady and it helped prevent both processors from simultaneously trying to access the same memory line in the same clock cycle. A memory controller was added to obtain a model closer to reality and add an interface between the memory and the cache.

The data structure simplified memory implementation. In this model, both the upstream and downstream memory have a data request command, read/write ports and main memory – `data_mem`. The memory was initialised at the beginning from a `.mem` files, shown on line 14 of listing 5.8. This allowed writing the maximum to trade from simulated previous days of trade in memory as well as to save the next days' worth of data. If a request to write was detected, the new value was added to the existing amount in memory.

For order handling, the memory write action was defined as an addition to the data in memory as explained in section 4. This logic is handled by the processor in this design. As a result, a request to read/write only the accumulated orders or maximum to trade reads/writes the entire memory line.

Listing 5.8: Example of a memory definition in SystemVerilog.

```
1  module mem_controller(input bit clk, //write clock
2    output mem_data_type mem_data, //memory response (memory->cache)
3    input mem_req_type mem_req) ;//memory request (cache->memory))
4    // register initialisation and definition (not shown)
5      initial begin
6          $display("Loading upstream memory.");
7          $readmemh("code/shared/rom_trade.mem", memory);
8      end
9      //defines read with delay 3 clock cycles triggered on mem_req.addr (not shown)
10
11       always @(mem_req.rw, mem_req.valid) begin
12         datardy = 1'b0;
13           if (mem_req.data[31:16] > 2'b01)
14             memory[mem_req.addr] <= {mem_req.data[31:15],memory[mem_req.addr][15:0] };
15           else begin
16             memory[mem_req.addr][15:0] <=  memory[mem_req.addr][15:0] + mem_req.data[15:0];
17           end
18              fork
19               begin  : write_wait_fork
20                 for (i = 0;i < 4;  i = i + 1) begin
21                   @(posedge clk);
22                 end
23               end : write_wait_fork
24             join
25             wait fork;
26             datardy = 1'b1;
27         end
```

Delays were implemented with the `fork` tool. This created a subprocess and allowed several blocks to run simultaneously. When running processor can only move onto the line below `wait fork` on line 31 once all the processes have terminated. For this implementation, that was equivalent to a 4 cycle delay. The control signal indicates that the data is unasserted from lines 18 to 32 when it is read. The unclocked modules are therefore delayed until this signal is asserted.

5.5 Cache implementation

5.5.1 Cache Components

The SystemVerilog implementation was inspired by Patterson and Hennessy design of write-back caches [33].

Caches were built on top of the memory model. The cache logic was split into 3 sub-modules: `dm_data, dm_cache_tag` and `dm_cache_fsm`. With this design, the data module is the same as the memory module as it handled the cache's data. The cache module was written to map tags to the data location. Figures 5.2 and 5.2 show the cache structure [33] and its block diagram, respectively. The `accumulated_orders` and `maximum_to_trade` share the same address and are written at different locations in memory.

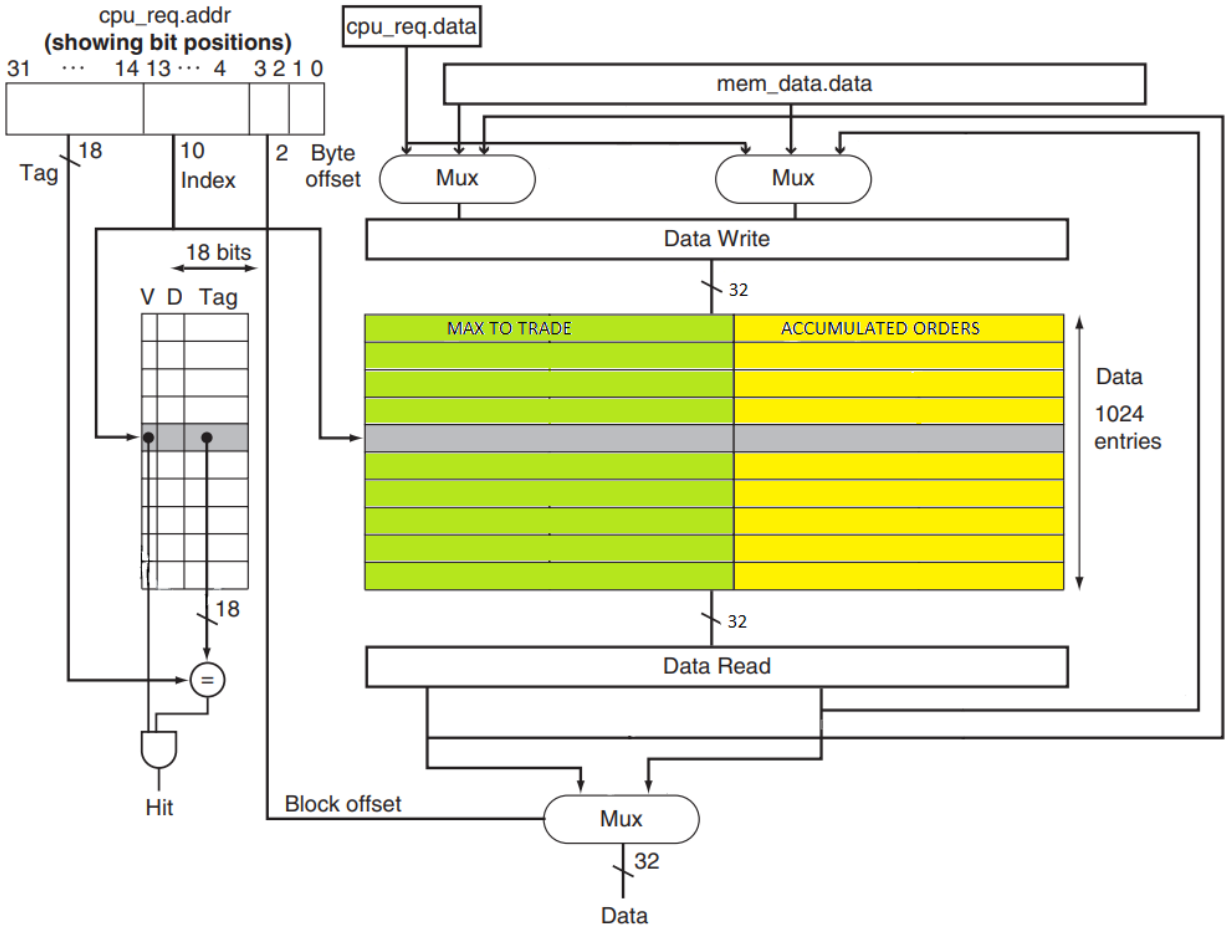
Some implementation choices can be outlined from figure 5.2:

1. Only 9 bits were used for indexing the cpu address, the index was defined as `cpu_req.add[13:4]`; the remaining bits were kept for the index and byte offset.

2. The continuous cache line of 32 bits was split into two 16 bits sections to split the maximum to trade and accumulated orders.

3. The cache is managed by the `dm_tag_upstream` module, which maps the tags and their stages `valid` or `dirty`.

Figure 5.2: Cache upstream module layout. Figure inspired from Patterson and Hennessy [33]



4. As a result, the cache consists of three main modules: `dm_cache_fsm`, `dm_tag` and `dm_cache_data`. Furthermore, the FSM connects both the tag and data modules to the memory controller: `mem_data.data` in this graph.

5. Data write and data read have a separate index in figure 5.2. This allows to process a read request and write request simultaneously for different cache lines. It is only used by the downstream processor, which reads from the `cpu_client_id` at the same time as writing to `exchange_client_id`.

The downstream cache was implemented comparably to the upstream cache, by combining the two blocks in one to create a one-way cache.

In this model, the inputs are the processor request and responses from memory – `cpu_req`, `mem_data`. The listing below displays all the default values for the states and variables. This ensured that

variables always go back to their baseline value after each clock cycle. For example, `tag_req` only needed to be asserted once to avoid several requests to the same cache tag.

5.5.2 Cache FSM

The cache FSM logic was implemented with a case statement based on the state on line 10, inside an `always_comb`. This allowed for a fully-combinational implementation. In this model, the states switch upon variable transition rather than clock rising edge. Registers inside the module trigger a state change. The FSM's implementation closely follows the diagram defined in chapter 4; the cache can be defined as write-back or write-through by adding a request to memory in the allocate state. Before any testing was performed, some sanity checks were run to ensure the data written in the cache and states were correct. Sample outputs are in the Appendix.

This module links the memory controller, cache and tag modules by passing an inferred list of variables on lines 16-18 of listing 5.9.

Listing 5.9: Code snippet for the cache FSM.

```
1  /*cache finite state machine*/
2  module dm_cache_fsm_upstream(input bit clk, input bit rst,
3    input cpu_req_type cpu_req, //CPU request input (CPU->cache)
4    input mem_data_type mem_data, //memory response (memory->cache)
5    // input change_max,
6    output mem_req_type mem_req, //memory request (cache->memory)
7    output cpu_result_type cpu_res //cache result (cache->CPU)
8    );
9    typedef enum {idle, compare_tag, allocate, write_back} cache_state_type;
10   /*FSM state register*/
11   cache_state_type vstate, rstate;
12   /*interface signals to tag memory*/
13   /*register definition and assignment*/
14   dm_cache_tag_upstream ctag(.*);
15   dm_data_upstream cdata(.*);
16   mem_controller cmem(.*);
```

The tag and cache data are stored and handled by separate modules. Their implementation

was very similar. The tag example is shown below. The maximum number of entries of the cache is set dynamically with `TAGMAX`, facilitating testing.

Listing 5.10: Code snippet for the cache FSM.

```

/*cache: tag memory, single port, 1024 blocks*/
module dm_cache_tag_upstream(input bit clk, //write clock
  input cache_req_type tag_req, //tag request/command, e.g. RW, valid
  input cache_tag_type tag_write,//write port
  output cache_tag_type tag_read //read port);//read port
);
  cache_tag_type tag_mem[0:TAGMAX];
  reg[6:0] i;
  initial begin
    for (int i=0; i<TAGMAX; i++)
      tag_mem[i] = '0;
  end
  assign tag_read.tag = tag_mem[tag_req.rdindex];
 //write to tag_mem on tag_req.we (not shown)
endmodule
```

In order to perform the write-back policy, the cache FSM stays in the `allocate` state until the memory is ready, then goes back to `compare_tag` and `write_back`. Once this loop is completed, the memory is set to "ready" to indicate the write is completed successfully. A reset signal forces the variable to go back to its default value shown on the code snippet above within a single clock cycle. This is a useful worst-case scenario to stop all trading if a problem is detected.

Listing 5.11: FSM in SystemVerilog, part II. This section describes the default value of all signals. This code snippet reset the variables to these values for the following clock cycle [33].

```

@always_comb begin

 /*-------------------------default values for all signals------------*/
/*no state change by default*/
  vstate = rstate;
  v_cpu_res = '{0, 0}; tag_write = '{0, 0, 0};
 /*read tag by default*/
 tag_req.we = '0;
 /*direct map index for tag*/
```

```
11 tag_req.index = cpu_req.addr[13:4];
12
13 /*read current cache line by default*/
14 data_req.we = '0;
15 /*direct map index for cache data*/
16 data_req.index = cpu_req.addr[13:4];
17 /*modify correct word (32-bit) based on address*/
18 data_write = data_read;
19 case(cpu_req.addr[3:2])
20 2'b00:data_write[31:0] = cpu_req.data;
21 2'b01:data_write[63:32] = cpu_req.data;
22 2'b10:data_write[95:64] = cpu_req.data;
23 2'b11:data_write[127:96] = cpu_req.data;
24 endcase
25
26 /*read out correct word(32-bit) from cache (to CPU)*/
27 case(cpu_req.addr[3:2])
28 2'b00:v_cpu_res.data = data_read[31:0];
29 2'b11:v_cpu_res.data = data_read[63:32];
30 endcase
31
32 /*memory request address (sampled from CPU request)*/
33 v_mem_req.addr = cpu_req.addr;
34 /*memory request data (used in write)*/
35 v_mem_req.data = data
```

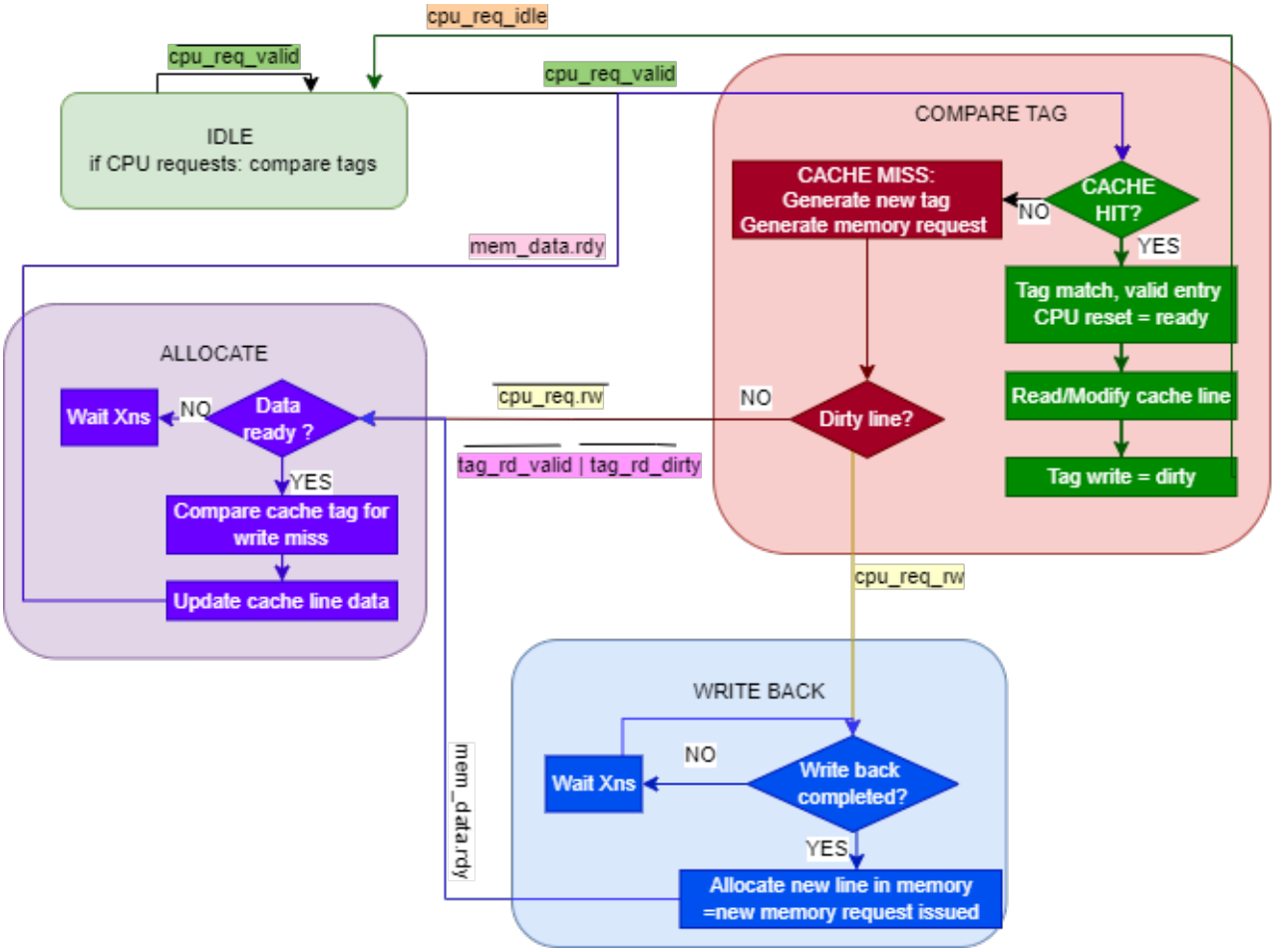
5.6 Timing implementation

SystemVerilog "always"

The variable change was defined for each module inside `always` blocks. Since the program is meant to react to any change in order requests, the goal was to have a highly sensitive system that was always well-defined.

The cache FSM was implemented within an `always_comb` block. Both caches are completely combinational, hence there was no need to define a sensitivity list. "In this way SystemVerilog ensures that the software tools will infer the same sensitivity list" [59].

Figure 5.3: Cache protocol FSM. This protocol involves MESI with snooping.



	always_comb	always @(*)
Triggered	Inferred sensitivity list	Start of program + Inferred sensitivity list + registers
Event/timing control, forks, blocking events	✖	✔
Multiple processes writing one variable	✖	✔

Table 5.1: Comparative table of always statements in SystemVerilog

For FSMs that are purely combinational, `always_comb` was favoured. It is sensitive to changes within the contents of the function, whereas `always @(*)` is only triggered by a new argument in the function. Furthermore, `always_comb` executes once at time zero, whilst `always @(*)` waits for a change in the inferred sensitivity list.

For processors, when different modules can be written/read simultaneously, `always @(*)` was chosen. The looser restrictions permitted the addition of forks in order to implement hardware delay, as well as writing multiple processes to the same variable.

Table 5.1 summarises the differences between the two methods. The cache FSM, downstream, upstream FSM and downstream processor were safer to be implemented with an `always_comb` block. On the other hand, the upstream processor could be updated by several modules; the top-level module also must account for the delay implemented in the cache and memory controller modules – they were implemented within an `always @(*)` block.

Chapter 6

Testing

To ensure successful verification, functional and statistical coverage were tested. Directed testing was used to verify the model's features and check the outputs based on predefined tests. Since this method left too much room for undefined behaviour, it was combined with coverage-driven constrained random testing. The random stimulus provided better coverage, constraints focused the testing on areas of interest only.

6.1 Risk Assessment

The model's coverage and behaviour were tested to ensure that all outputs were defined, in a hardware verification manner. For each risk, a test was created. As a result, one or more tests were added to the testbench.

List of possible risks assessed:

1. *Time*: Delays the module caused by the processor when it needs to correct, detect or go through extra states to perform the same operation. An example of a time risk is a longer critical path than intended that slows down the clock or gets stuck in a transient state.
2. *Function*: Possible error that impacts the behaviour of the whole system, for example, a design bug.

3. *Cost*: Risk that leads to a higher product cost, by using more hardware than needed, or more energy than required.

Key	*Risk*
A	Substantial deal breaker
B	Causes disruption to project timeline/delivery/budget
C	Inconvenient

Table 6.1: Legend for table 6.2

Table 6.2 shows all risks and measures taken for the top-level module of the baseline model

Table 6.2: Contingency plan for the the top-level module (1) showing time, performance and cost risks. Please see legend 6.1 for code colour.

Risk	**Description**	**Time**	**Perf.**	**Cost**	**Mitigation**
Client ID address out of range	Input `client_id` address for writing or reading to memory size is greater than 4 bytes in reality. As a result, the model would write the *Amount* value for the wrong client or lead to undefined behaviour since the client is unknown.	A	A	A	Constrained random testing and SVAs. Logic will also be added in relevant modules to only process orders if the client exists within bounds.
The amount is empty	Amount holds a wrong value when a new order or new maximum signal is detected. This is dangerous and could result in faulty memory lines. In the context of trading, an impossibly high amount may result in thousands of pounds of orders wrongly sent and lost to the exchange.	B	A	A	Risk check on the New_{order} upstream processor will only happen if *Amount* resides between predefined bounds. Similarly, the order can only be cancelled if its value is defined and between a minimum and maximum set by the system.
Data signals not set.	Undefined inputs. It is impossible to predict the system's behaviour in this situation	A	A	A	Upon the system start, both `client_id` and `amount` are given default values that won't affect the system to unsure they are defined at all times.

Table 6.2: Contingency plan – continued on next page

Table 6.2: Continued from the previous page

Risk	Description	Time	Perf.	Cost	Mitigation
Cancel + New Order, different clients	The testbench tries to cancel and send a new order concurrently for different clients.	C	B	A	In the worst-case scenario, the system should prioritise the cancelled order over new order. In the best-case scenario, both operations should be performed simultaneously. This is possible since each processor gets separate `client_{id}` and `amount` values.
New Order + New Max, any client	The testbench sets a new maximum and sends a new order at the same time for different clients or the same client.	C	B	A	Not allowed by the model. There is only one data signal to handle both new orders and new maximums. Future improvements could split memory blocks and add signals. This was considered here as an impossible combination.
Cancel + New Order, one client	The testbench cancels and sends an order simultaneously for the same client. This is problematic: risk check might wrongly fail after the cancel has been written to memory if the new order reads a dirty line.	B	A	A	The top-level logic always prioritises cancels over orders to ensure that no dirty lines are read. Improvements could involve passing data in registers. Testing extreme cases in constrained random testing ensures correct behaviour.
Concurrent Cancel + New Max, different clients	The testbench send new maximum and cancel order request simultaneously for different clients.	C	B	A	Not allowed by the model. There is only one data signal to handle the two variables. Future improvements could split memory blocks and add signals.
Order is not acknowledged	The memory rd/wr signal is never asserted, the system doesn't know if the order was correctly sent and stays stucks.	B	A	A	Testing checks latency per order. A timeout allows the system to reset the system to `IDLE` state and restore all values in case this ever happens.
False memory read-/write	Memory is said written/read when not. This could happen for both reduced and accumulated risk memory blocks.	C	A	B	If the memory holds a wrong value, there is no option for the system to know. Thorough functional testing should give a minimum of 99% certainty that the memory signals for read and write operations are always correct.

Table 6.2: Contingency plan – continued on next page

Table 6.2: Continued from the previous page

Risk	Description	Time	Perf.	Cost	Mitigation
Orders rate greater than logic	Incoming order latency is greater than module throughput, the modules cannot process most orders	C	A	A	Research and realistic testing should be performed as early as possible to avoid this to happen. The threshold for order rate is high since real-life models tend to process less than 40% of the orders received [19].

Table 6.2: End of Contingency plan table from the previous page.

6.2 Testing Coverage

Code coverage is a metric that helps to assess the quality of the test suite.

What percentage of coverage should be aimed for? Generally, the goal is to reach 100% code coverage. Error is not acceptable for brokers, basic checks are allowed. Any combination of random variables can help to uncover a problem. This model aimed to be robust over anything else. The usual balance between coverage and cost in testing did not apply here since coverage of, for example, 80% would mean that the model may lose billions of dollars 20% of the time.

To achieve such coverage, the range of random constraints range was reduced. In the scope of this project, dropping some new and cancelled orders was tolerated. The model can be manually reset, but this signal did not count in code coverage – resetting the input was kept for testing purposes only.

Since the goal was to cover the largest possible testing space to increase confidence in the model, different types of coverage were implemented.

1. Functional coverage: all functions must be called during testing.
2. Statement Coverage: all cases, except reset cases, must be executed.
3. Branch Coverage: Since the processors were implemented with FSMs and because top-level modules contain `if...else` statements, all branches must be tested. All states

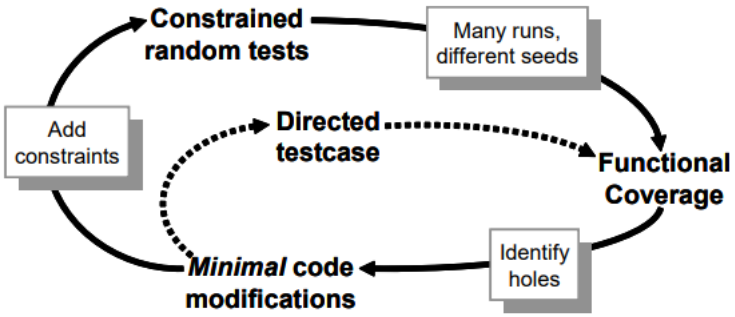


Figure 6.1: Code coverage convergence. Verification will be a step by step process with added constraints to ensure outputs coverage. Reference: [60]

must occur during training to be realistic. In the case where this may happen, the case statement or branch ought to be removed.

4. Conditional coverage: ideally, all boolean sub-expressions must be tested. This could lead to a prolonged testing period. Testing the maximum of conditions was important, but all combinations are not to be tested.

Figure 6.1 displays the testing strategy for coverage. Tests were gradually added from unit-tests, and constrained random tests to identify errors and problems. Functional code coverage was reached after several runs and new tests – when the code behaved as expected under all circumstances defined above.

6.2.1 Testbench Architecture

Two types of tests were implemented: unit-tests and constrained random tests. Coverage reports were generated from the constrained random testbench with the architecture depicted in figure 6.2. These reports identify critical misses in testing; more sophisticated reports list all variables' values and coverage throughout testing. Information about both behaviour and coverage could be extracted from these tests. Each variable could take any value, except the ones that were manually constrained to a value [43].

OOP was used to abstract each part of the testbench into classes shown in figure 6.2. An

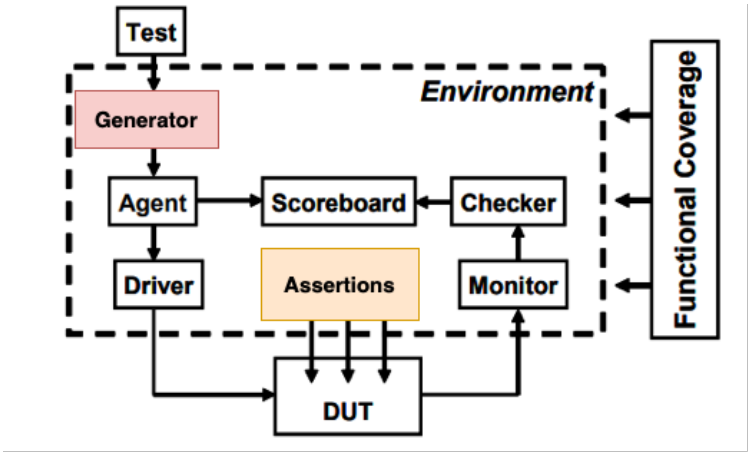


Figure 6.2: Testbench structure. Reference: [60]

environment was created and interfaced. The driver module linked the design under test to the rest of the testbench. Several modules were used to abstract the logic, as shown in figure 6.2:

1. **driver_*.sv**: The model's inputs are driven by the driver through the interface, which links the inputs and outputs to the main testbench.

2. **environment_*.sv**: Defines the scenario, functional and command layers to abstract the logic.

3. **generator_*.sv**: This module drives the functional layer. It narrows down the constraints for each parameter and generates random transactions to the driver iteratively.

4. **interface_*.sv**: The interface contains the IO for each of them, along with a clocking block that defines the logic clock and test clock.

5. **monitor_*.sv**: The model's outputs drive the monitor. It groups the signal transitions into commands.

6. **scoreboard_*.sv**: This module records the result of each transaction for the checker to compare with the actual result.

7. **tb_top_*.sv**: Defines the coverpoints for the design under test.

8. **tests_*.sv**: Initialises testing environment and start relevant threads.

9. **transaction_*.sv**: It records relevant information as a transaction class.

Coverage Testing

Each unit-test defined input values based on trading data that triggered a certain functionality – e.g. cancelling an order – and concrete output values that the model was expected to produce – e.g. adding the cancelled order value to the reduced risk memory. Sequences of trades were tested with different amounts and clients, on single modules and top-level modules, example results were copied in the appendix. These directed test inputs returned a pass or fail status and gave confidence in the dependability of the system. Whilst directed tests were sufficient to demonstrate dependability, they could not detect faults efficiently. By testing a small subset of the input space, there was no sound basis for extrapolating to untested input combinations. Each test was also written manually and studied to compute the expected outputs: it may be the victim of human error. The implementation cost to development cost ratio was particularly high.

As a result, constrained random testing was required on this safety-critical FPGA model [29]. It enabled the software developer to check that the model produced consistent outputs, providing it with constrained random test inputs. The model was duplicated: the device under test in figure 6.2 was compared and checked against a source-truth defined in the checker. The test generator should have been based on the requirements of chapter 2. However, the gap between English written requirements and constrained tests in SystemVerilog prevented translating directly between the two. To formalise the requirements, tests were based on risk table 6.1. The testbench checked each entry and checked output stability.

6.2.2 Input bounds

The top-level module defined in figure 2.7a was verified with constrained random testing. A minimal case was defined; some inputs could safely be constrained to reduce the number of test cases. By freeing the remaining inputs, the code coverage increased but functional coverage stayed the same. It was expected of constrained values not to affect the model's behaviour.

Input	*Can be constrained?*	*Value*
New Cancel	✗	✗
New Order	✓	NOT(New Max)
New Max	✗	✗
ClientId	✓	Under $2^4 - 1$
Amount	✓	Under $2^8 - 1$

Table 6.3: **Constrained inputs** for the top-level module. Please see figure for an overview.

Input	*Can be constrained?*	*Value*
New Cancel	✗	✗
Mem Write	✗	✗
ClientId	✓	Under $2^4 - 1$
Amount	✓	Under $2^8 - 1$

Table 6.4: **Constrained inputs** for the downstream processor top-level module. Please see figure for an overview.

Input	*Can be constrained?*	*Value*
Risk Check	✗	✗
New Order	✓	NOT(New Max)
New Max	✗	✗
Mem Write	✗	✗
ClientId	✓	Under $2^4 - 1$
Amount	✓	Under $2^8 - 1$

Table 6.5: **Constrained inputs** for the upstream processor top-level module. Please see figure for an overview.

Values from tables 6.4 and 6.5 were used on unit-level and top-level testing to test a smaller pool of possible input combinations during random testing. Some of these inputs were also set in advance during formal verification, as specified by the risk assessment table 6.2.

6.2.3 Output generation

For different models, outputs were generated by changing the number of iterations `CNRT` below, as well as the cache size `TAGMAX`, as described in chapter 5. For each output, the number of memory write operations to memory blocks, and their time to complete was recorded. All test conditions were kept the same. This guaranteed a fair comparison between models; only one variable could change at a time.

6.3 Testing logic

6.3.1 Basic checks

Listing 6.1: SystemVerilog fragment of environment.sv from the coverage testbench.

```
1  //constructor
2  function new(virtual updownstream_if updown_if);
3    this.updown_if = updown_if;   //get the interface from test
4    gen2driv = new; //creating the mailbox (Same handle shared)
5    mon2scb = new;
6    crt_n = 500;    //counter iteration
7    //creating generator, driver, monitor and scoreboard
8    gen  = new(gen2driv, crt_n);
9    driv = new(updown_if,gen2driv);
10   mon = new(updown_if,mon2scb);
11   scb = new(monscb);
12   // synchronise event between generator and driver; monitor and scoreboard
13   gen.drv_done = drv_done;
14   driv.drv_done = drv_done;
15   mon.mon_done = mon_done;
16   scb.mon_done = mon_done;
17 endfunction
```

For predefined sequences of inputs, the states and outputs were printed and checked against their expected values in unit-tests. Such strategy helps build confidence progressively in the model from the bottom-up: from each singular module to top-level.

The Questasim Graphic User Interface (GUI) was used to display signals for each clock cycle and ensure that all input signals are connected and trigger a change in the outputs. Screenshots can be found in appendix 10.5.

6.3.2 SystemVerilog Assertions

SVAs were added for three reasons:

1. Risk checking: ensures the correct output after risk checking is as expected.

2. States logic: ensures the correct order of states. Guarantees that only a single state is asserted at all times.

3. Trade logic: checks the amount written to memory after adding the new order, and verifies where the amount is written for the upstream memory – `accumulated_order` or `max_to_trade`.

These assertions were implemented in the top-level modules as well as within the RAM/cache modules.

Listing 6.2: SVAs for the upstream processor top-level module.

```
`define STATELOGC1 ((update_max == 1) && (check_risk == 1))
`define STATELOGC2 ((update_max == 1) && (send_order == 1))
`define STATELOGC3 ((check_risk== 1) && (send_order == 1))
`define SAFETOTRADE  ($signed({1'b0, max_to_trade_reg[15:0]})> $signed(result))
`define AMOUNTGOOD ((correct_amount == amount)||(correct_amount[31:16]== amount))


   trade_correctamount_cpu: assert property ( //correct amount was written
     @(posedge clk) // throws an error if correct amount != amount to trade
     `AMOUNTGOOD == 1'b0)
     else begin
       $error ("The amount was not indexed properly");
     end //

   trade_pass_checks: assert property (
   @(posedge clk) // throws an error if the trade is unsafe
   `SAFETOTRADE == pass_checks )
   else begin
     $error ("pass_checks was not computed properly for client %0d", client_id);
   end //

   trade_state_logic1: assert property ( // SVA to check state logic
     @(posedge clk) // throws an error if two states high
     `STATELOGC1 == 1'b0 )
     else begin
       $error ("two states, update_max and check_risk, are simultaneously high");
     end //
     // others SVAs for state logic
end
```

Chapter 7

Results and Evaluation

7.1 Threats to validity

Listing 7.1: Clock definition in the testbench top-level module. The time unit is 1 nanosecond, the precision of each operation is set to 1 picosecond.

```
1    clk = 1'b0;
2    slowclk = 1'b0;
3    forever begin
4        #2 clk = ~clk;
5        #4 slowclk = ~slowclk;
6    end
```

The results and evaluation could have been skewed by some approximations from the testing protocol. Indeed, the constrained-random testbench was coded based on existing literature on LOBs:

1. The number of clients (stocks) considered for caching: **50 to 250** for any given interval.
2. **Exponential model** of each client's contribution, verified by Yousin Park at Morgan Stanley and Hendrick Bessembinder [19].
3. The overall trading frequency value agreed with Morgan Stanley HPC team was computed from the data provided by NASDAQ is **52,111** shares per second with a high standard

deviation.

4. The model's default time unit was **1 nanosecond**. According to line four of listing 7.1, one clock cycle took two nanoseconds for the device under test and **four nanoseconds** for the testbench. The longest cycle was used as a reference for the calculations below.

These values are only a reference for the test. The threats to the testbench validity are:

1. The data from table 1.1 is 12 years old. Morgan Stanley's trading team guaranteed that the current data is similar in its statistical metrics but different in quantity. It was therefore valid to use the client spread and ratio between orders. The LOB would only operate at a much higher rate. Mathematical models from 2020 studies [19] and [61] supported the idea that stocks' standard deviation and volume ratio tend to stay constant over time.

2. The model data was based on NASDAQ statistics, which has a lower density of orders than the US stock market, and the target environment. Results were assessed keeping in mind they had to be scaled up.

3. Data used by literature reviews consists of LOBs with a *start of the day*[1] of varying value. The maximum allowed to trade per client is different each day. Data initialization is a source of error.

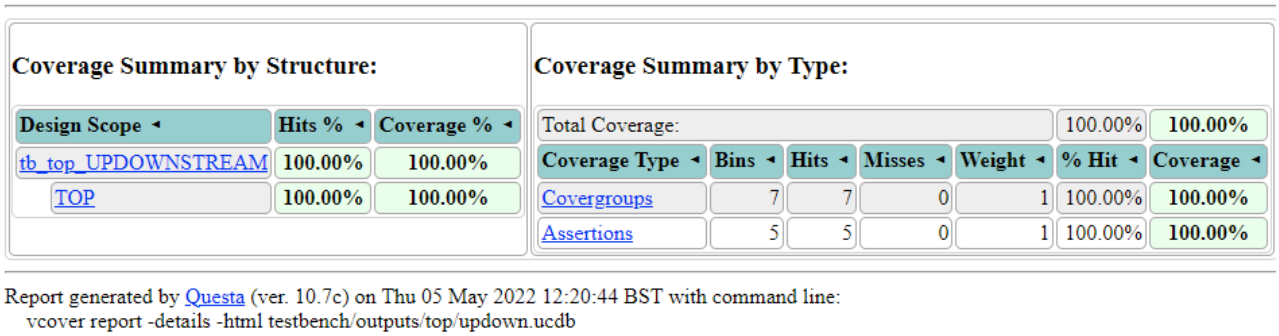
4. Orders tend to be sent repeatedly over a short period to the same clients before changing to another set of clients. While the constraints of a random test bench could narrow the pool of clients from which to choose, the random order in which they were chosen may not have followed a realistic pattern.

[1]Start of the day or beginning of the day is the initial state of the LOBs before trading opens at 7 am (UK). In this project, it was simulated as the start value of the memory or cache.

7.2 Baseline model performance

7.2.1 Coverage

Figure 7.1: Screenshot of the coverage report overview.



Coverage Summary by Structure:

Design Scope	Hits %	Coverage %
tb_top_UPDOWNSTREAM	100.00%	100.00%
TOP	100.00%	100.00%

Coverage Summary by Type:

Total Coverage:					100.00%	100.00%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Covergroups	7	7	0	1	100.00%	100.00%
Assertions	5	5	0	1	100.00%	100.00%

Report generated by Questa (ver. 10.7c) on Thu 05 May 2022 12:20:44 BST with command line:
vcover report -details -html testbench/outputs/top/updown.ucdb

While unit-level testing provided the first level of assurance that the baseline model was secure, the results of constrained-random testing gave supplementary confidence. The coverage report in figure 7.3 supports a full-coverage for the interval of clients, orders, states and values in memory defined in the code. This was crucial to the success of this project:

1. The model did not get locked in a state from any of the FSMs. It was safe in the case of sending an accumulated order, cancelling an order, and updating the maximum in a single transaction for the same client and different clients.

2. This model could prioritise tasks and drop the ones that could not be performed under time constraints.

3. A misguided broker desiring to lead the firm to bankruptcy would have a hard time since any order of any value can be handled.

7.2.2 Functional performance

The results of the SystemVerilog Assertions on the baseline model were as expected and gave certainty to the model. In figure 7.2, the outputs of $50,000$ pseudo-random input combinations

Figure 7.2: Screenshots of the results for the assertions on the baseline model.

Questa Assertion Coverage Report

Show All | Show Covered | Show Missing

Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
trade_pass_checks	0	1	-	-	-	-	-	Covered
trade_correctamount_cpu	0	1	-	-	-	-	-	Covered
trade_state_logic1	0	1	-	-	-	-	-	Covered
trade_state_logic2	0	1	-	-	-	-	-	Covered
trade_state_logic3	0	1	-	-	-	-	-	Covered

Figure 7.3: Screenshots of the results for the coverage on the baseline model.

Covergroups/Instances	Total Bins	Hits	Misses	Hits %	Goal %	Coverage %
/tb_top_UPDOWNSTREAM/cg_input	4	4	0	100.00%	100.00%	100.00%
/tb_top_UPDOWNSTREAM/cg_input	4	4	0	100.00%	100.00%	100.00%
/tb_top_UPDOWNSTREAM/cg_output	3	3	0	100.00%	100.00%	100.00%
/tb_top_UPDOWNSTREAM/cg_output	3	3	0	100.00%	100.00%	100.00%

always had the `pass_check` variable correctly computed – only safe orders were written in memory. The second line of figure 7.2 suggests that the correct block in memory was updated – the accumulated order block did not overflow in the maximum block for the same cache line. Lastly, the `trade_state_logic` verified the order of the consecutive states for each FSM. This ensured that no order was sent without checking its risk first, and that the cache write-back policy was implemented correctly – there was no discrepancy between the cache and memory.

7.2.3 Model efficiency

The relationship between orders sent and written to the upstream memory is shown in figure 7.4. As expected, most orders were dropped due to the high clock rate at which they are sent. The simulated hardware delay causes a ratio of one order sent to three orders dropped on the upstream processor. This trend was also followed by the downstream processor.

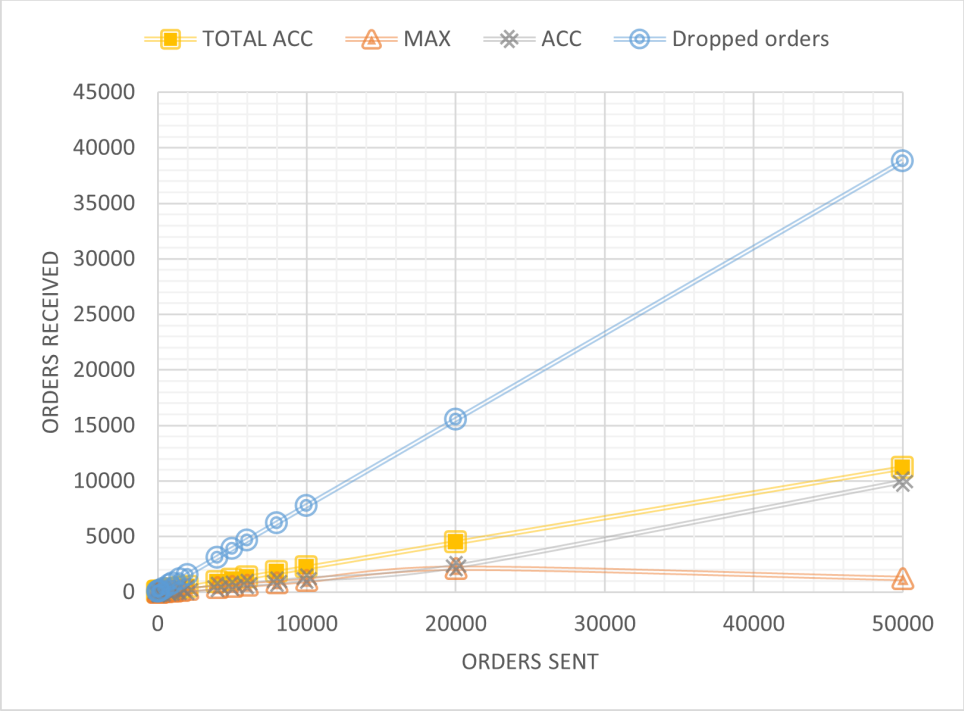
The rightmost point of the maximum to trade axis from figure 7.4 – for 50,000 orders – stands out. After a limit between 20,000 and 50,000, the accumulated orders are too high to update the maximum to trade. Any new maximum is lower than the cumulative amount traded.

Approximately one-fourth of the total orders were sent, with the majority of them being `new_orders` rather than updates in the maximum value. Table 7.1 displays the equations supporting these claims. In fact, the asynchronous model allows for a higher order rate than the minimum bound of three read delays plus four write delays. As a result, the time taken to process seven cycles corresponds to a delay of 21 nanoseconds for the baseline model.

The average time to process a single order can be found by taking the average total throughput for a different number of orders sent, in equation 7.1.

$$\frac{1}{6} \times \left(\frac{0.02}{400} + \frac{0.03}{800} + \frac{0.05}{1000} + \frac{0.07}{1500} + \frac{0.11}{2000} + \frac{0.14}{4000} \right) \times 10^{-9} = 45.694s \tag{7.1}$$

Figure 7.4: Orders received vs orders sent for the 128-bit RAM SystemVerilog implementation.



In sub-optimal testing conditions, with delays due to internet connection and server response, this corresponds to a rate of 45694 nanoseconds per order. This value is only a reference; the model is not optimised, there are several outliers in the timing data. By taking the weighted average of the total accumulated orders sent in figure 7.4, the ratio of orders sent and received was computed for the baseline model in equation 7.2. Since all orders were penalised by a

memory request delay, the efficiency value is relatively stable regardless of the number of orders sent. About $\frac{1}{4}$ of all orders were sent. This is in line with the delay of 7 cycles for a testbench that sends an order every 2 clock cycles.

$$BaselineEfficiency = \left(\sum_{i=1}^{NbSamples} \frac{received_i}{sent_i} \right) \times NbSamples = 22.34\% \tag{7.2}$$

7.2.4 Computing the penalty coefficient

The penalty coefficient for the missed orders by the cache was computed linearly based on the results above. It was assumed that the performance would be the same as the baseline model for any extraneous cache line: 22% [2].

$$Penalty\ Coefficient = 1 - 0.22 * \frac{size_{cache} - 64}{size_{cache}} \tag{7.3}$$

For example, the penalty coefficient for a 128-bit cache is $1 - 0.22 * \frac{128-64}{128} = 0.9278$, for 150-bit cache, 0.90613, and 0.8851 for 180 bit-cache. As the cache size increases, the coefficient decreases: additional orders are impacted by the hardware delay.

7.3 Write-back cache examples

7.3.1 Coverage and Functional performance

For both coverage and assertions, the results for cache coverage were identical to those for memory. SVAs remained unchanged. The code coverage for $50,000$ iterations was 100%, and all SVAs successfully passed.

[2] 22% is the percentage of orders sent from the orders received. This value was taken from the results of the baseline model in chapter 7

7.3.2 32-bit cache, 50 clients

Figure 7.5 varies greatly from the baseline model in figure 7.4. The number of dropped orders closely follows the number of accumulated orders sent until 1000 total sent orders. As the number of total orders increases in equation 7.4, so does the discrepancy between dropped and sent orders in equation 7.5, suggesting the cache is more efficient than the memory at processing orders. Cache line write rate can be modelled by a linear function.

$$Total\ Acc_i = 0.9538 \times sent_i - 17.988 \tag{7.4}$$

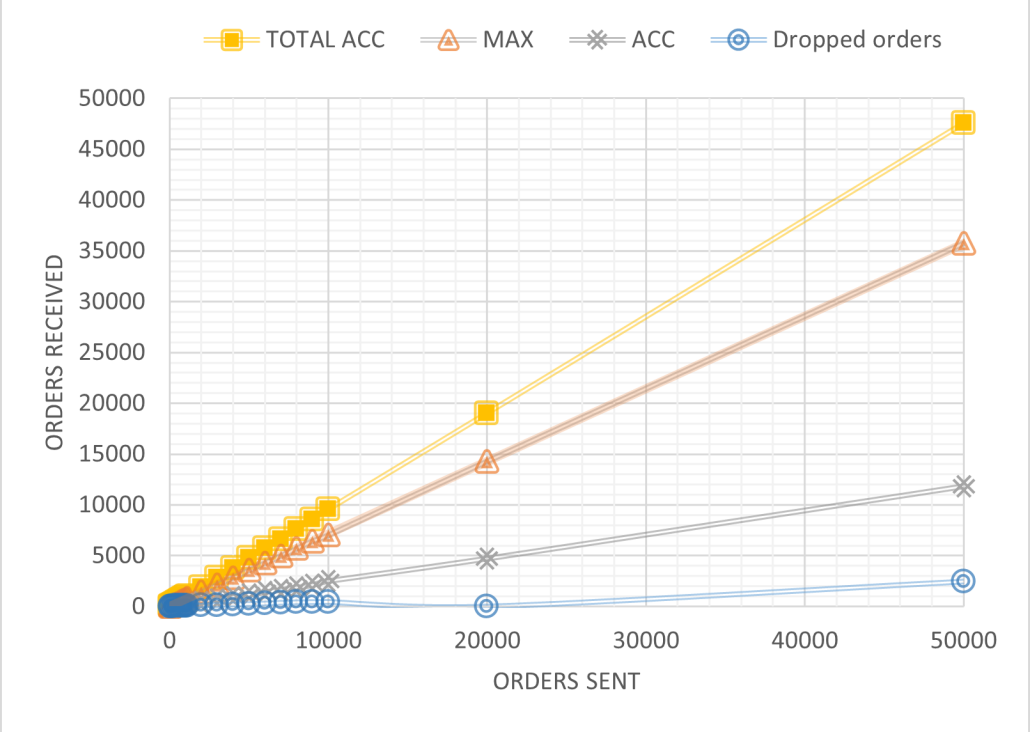
$$Dropped\ Orders_i = 0.0462 \times sent_i + 17.988 \tag{7.5}$$

It can be inferred from these equations that the cache added-value increases for a longer duration where only a few clients are traded on the LOB. For a share frequency of 52,111 shares/second, 25,000 to 50,000 shares would be traded every half a second up to one second. Assuming less than 50 clients are traded during this time frame, almost 90% of the trades would be sent instead of 22.3% in the baseline model.

The downstream cache follows the same trend as the upstream cache. From these results, the cache increased the sent order rate by **66.7%** compared to memory. This value is only for the time frame where the orders are repeated for a subset of clients, the total increase in orders sent will be lowered when writing a new subset of clients in the cache. For the given order rate of $52,111$ shares/seconds, this corresponds to an extra **34,794** shares each second in the optimal case of repeated orders for 50 clients as computed in equation 7.6.

$$ModelA \rightarrow 66.77\% \times 52,111 = 34794.5\ extra\ shares\ traded\ in\ one\ second \tag{7.6}$$

Figure 7.5: Orders received vs orders sent for a 32-bit write-back cache based on a pseudo random sequence of 50 clients.

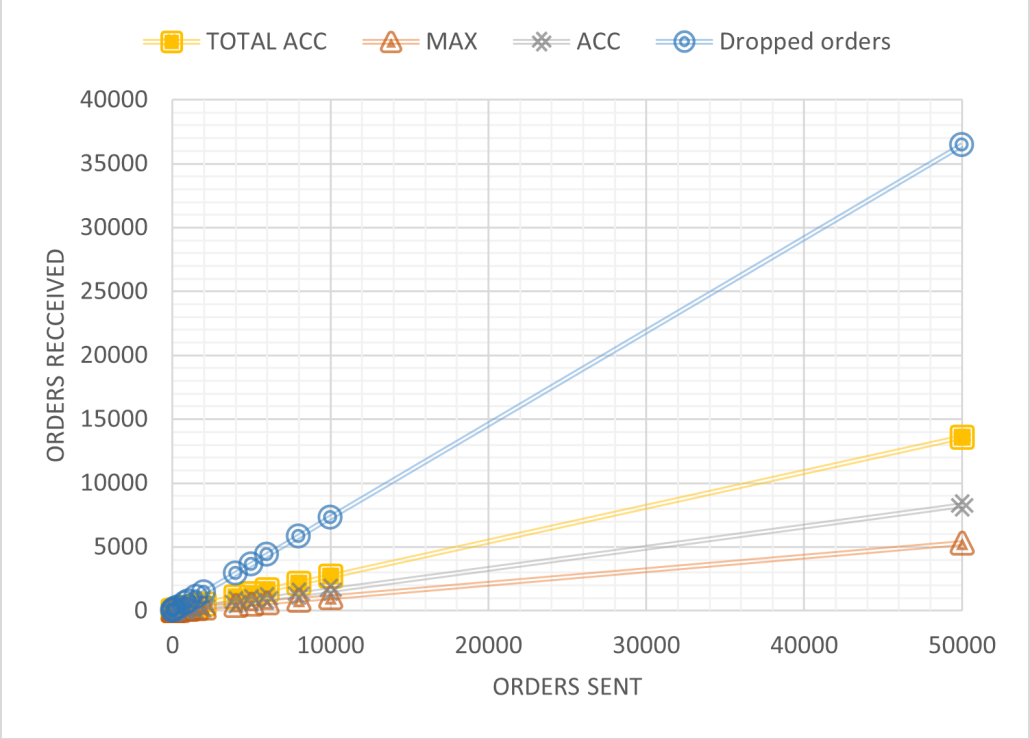


7.3.3 12-bit cache, 250 clients

A hypothesis was made that the caches may have some hardware delay due to the large amounts of data stored in each cache line. In chapter 4, the maximum bandwidth and clock rate for the FPGA logic limits the size of the data retrieved to 64 bytes. A cache line of 8 bits for the cache tag corresponds to a maximum cache size of 64 bits. Above this value, the penalty can be linearly computed. Small caches were tested to avoid any hardware delay.

For the upper limit of 250 clients, figure 7.6 showing the ratio of orders sent to received is similar to the figure representing output from the baseline model. Dropped orders dominate the result; the discrepancy linearly increases with the number of orders sent. The percentage of shares traded in addition to the baseline model is lower than 5%.

Figure 7.6: Orders received vs orders sent for a 12-bit write-back cache based on a pseudo random sequence of 250 clients.



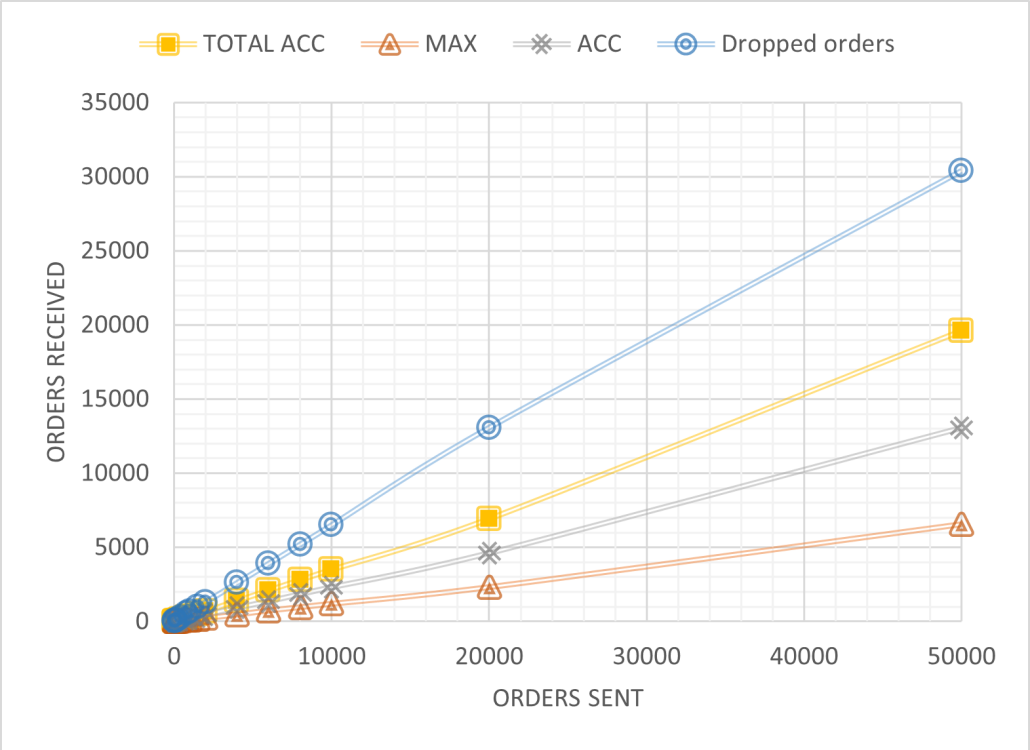
7.3.4 12-bit cache, 20 clients

A smaller scale of clients traded and cache size was compared with the 32-bit, 50-clients model. The conjecture was that a smaller cache would be easier and quicker to implement than the one described in the previous section. In a future development, there could be several caches for different groups of clients traded together. The hypothesis for trying this configuration was based on the project's definition of a proof of concept—a better performing small cache could be a better compromise than a single big cache.

As shown in figure 7.7, there was a slight improvement compared to the baseline model. For example, 30,000 orders were dropped for the maximum value of 50,000 instead of the 37,000 for the latter. Applying the same formula as in equation 7.2, the average percentage of shares sent was **66.64%.**. This corresponds to an increase in order of $66.64 - 22.34 = 44.3\%$. Given that 52,111 shares were traded in a second, this model theoretically offers 23,085 extra shares traded every second.

$$ModelB \rightarrow 44.3\% \times 52,111 = 23,085 \; extra \; shares \; traded \; in \; one \; second \tag{7.7}$$

Figure 7.7: Orders received vs orders sent for a 12-bit write-back cache based on a pseudo random sequence of 20 clients.



7.3.5 Write-back cache performance

Comparison of the three models

Each line from figure 7.8 was mapped to its equation in table 7.1. These outcomes are similar to the previous ones. The model for a 32-bit cache with 50 clients performs the best out of the subset shown in figure 7.8. This model is similar to the baseline model when the cache is small in comparison to the subset of clients traded. Notably, the two black lines are close to each other in figure 7.8; they correspond to the baseline model and 12-bit, 255 clients model. The accumulated orders follow the same pattern. In fact, the blue and black dashed lines cross after 30,000 orders were sent: caching has proved detrimental for incorrect parameters. On the other hand, the green lines for the 32-bit, 50 clients cache stand out since its performance is

Model	*Total orders sent*	*Accumulated orders*	*Percentage of orders sent*
Baseline (Memory)	$0.2239x - 2.115$	$0.1911 - 244.57$	22.4%
32 bits, 50 clients	$0.9519x - 12.57$	$0.2367x + 13.541$	90%
12 bits, 250 clients	$0.2713x - 2.3084$	$0.1655x - 6.8829$	25.7%
12 bits, 20 clients	$0.3879x - 135.62$	$0.258x - 96.615$	66.6 %

Table 7.1: Results of different models on constrained-random testing and their efficiency. x represents the number of orders sent.

almost 100%.

Write-back cache performance

The results above suggest a linear relationship between the number of orders sent and orders received for different clients and cache sizes. The cache tested was not affected by a hardware delay: its size is under the 64-bit limit calculated in chapter 5. In figure 7.9 the total number of orders sent for a wider spectrum of models shows that the linear trend breaks for caches size greater than 96 bits.

Combining the results given by the model and the penalty coefficient from equation 7.3 gives the results in figure 7.9. In figure 7.9, The trendline's slope is steeper between 24 and 64 bits on the x-axis. It then decreases for a cache size greater than 96 bits. The subsets of clients tested are close to 64 and 96 (50 and 100), the cache can hold almost all the data for these cases. It can also be noted that the x-axis values double at each tick; the increase in the number of orders sent is linear. From these results, the maximum of orders sent is between 64 and 96 bit-caches. For few clients traded, the latter is preferable, while for more than 100 clients traded, 96 bit-cache outperforms the others.

In table 7.9, cache capacity stalls at a size greater than the number of clients traded. For example, the limit of a 16-bit cache was reached for ten clients. To save space on the FPGA and avoid hardware delays, the smallest cache size that reaches maximum capacity is preferred over larger caches.

Given the bandwidth of 24 bytes and clock rate of 500MHz of the model – c.f. chapter 4 – the number of orders being processed decreases for caches of a size greater than 96 bits. This is

Figure 7.8: Orders written to the accumulated memory block of the upstream cache for different models.

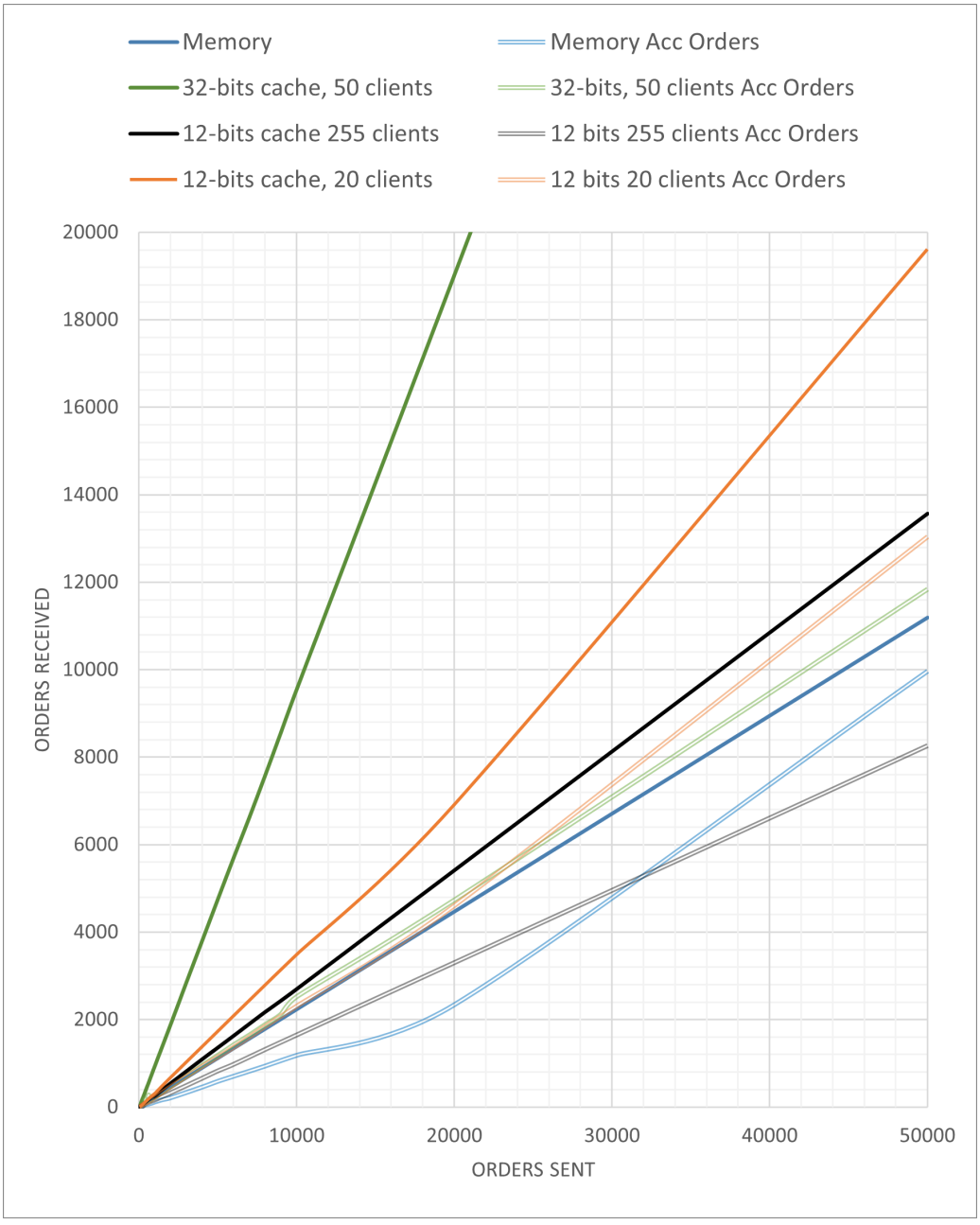


Figure 7.9: Comparison of different cache sizes results for a total of 5,000 orders sent.

	0	2	4	8	16	24	32	64	96	128	150	180
10 clients	1134	1194	1364	2321	4774	4774	4774	4774	4654	4430.3	4326.79	4226.41
50 clients	1116	1213	1131	1188	1329	1496	1886	4652	4458	4321.75	4220.77	4122.85
100 clients	1114	1127	1136	1153	1250	1395	1652	4522	4596	4316.18	4215.33	4117.54
200 clients	1114	1127	1130	1163	1251	1381	1469	3374	4626	4292.06	3889.18	3442.36
500 clients	1114	1129	1130	1158	1209	1406	1449	4579	4549	4248.45	4149.18	4052.92

due to the penalty coefficient.

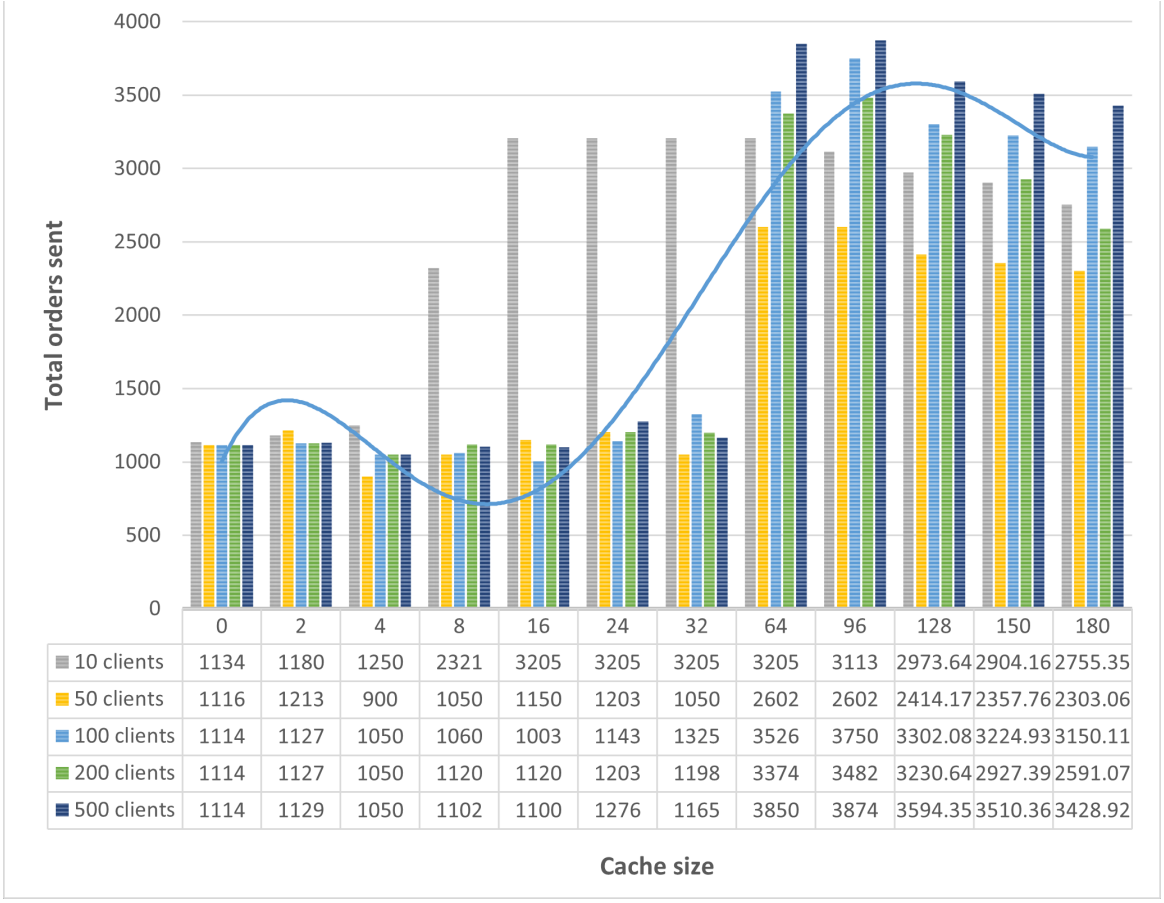
Lastly, the large cache sizes in figure 7.9 tend to perform better than small cache sizes. In this model, hardware delay has fewer impacts than it has on the memory since the cache has already loaded the values in registers. Instead of unpacking and transferring the address and value to read, the delay is caused by computing the read and write tags.

7.4 Write-through cache performance

The average percentage of orders processed by the write-through cache in figure 7.10 is 62%. This is low compared to the three cases developed in the previous sections.

The polynomial trendline in figures 7.10 and 7.11 is similar to the write-back cache trendline.

Figure 7.10: Write-through orders received against the cache size for an example of 5,000 orders sent.

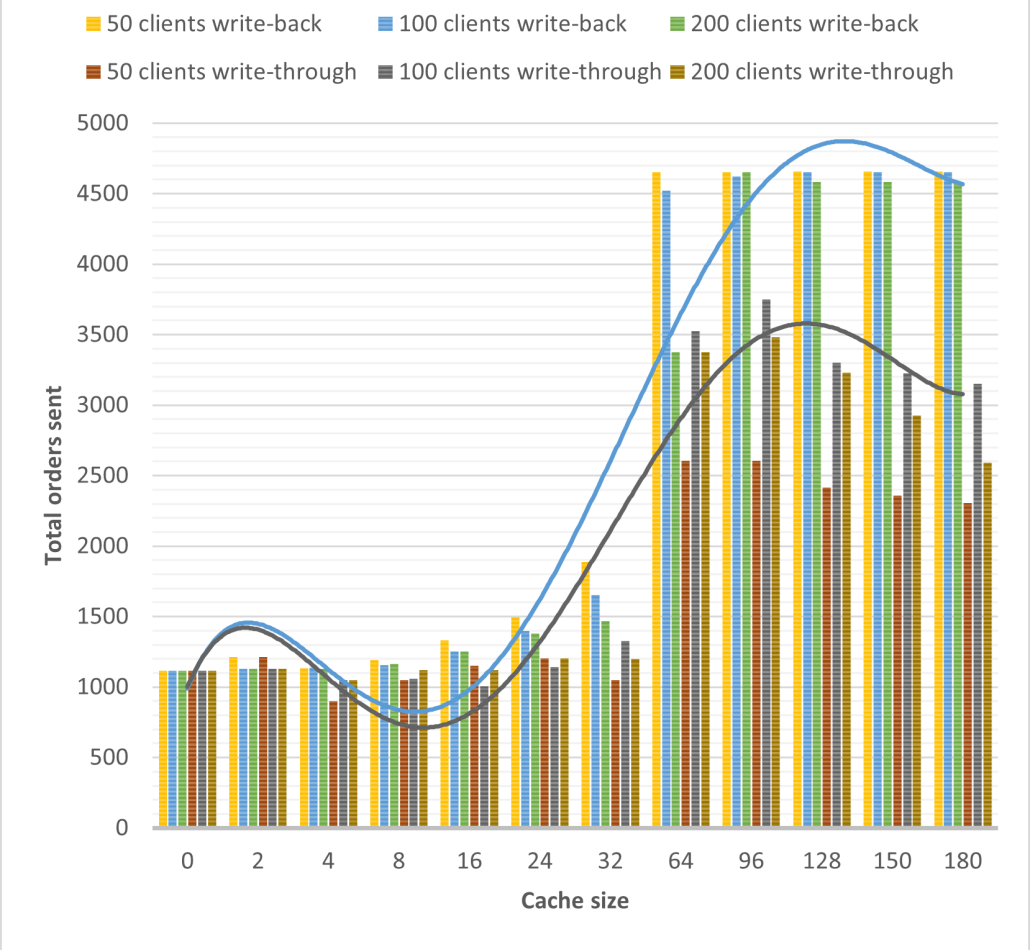


	0	2	4	8	16	24	32	64	96	128	150	180
10 clients	1134	1180	1250	2321	3205	3205	3205	3205	3113	2973.64	2904.16	2755.35
50 clients	1116	1213	900	1050	1150	1203	1050	2602	2602	2414.17	2357.76	2303.06
100 clients	1114	1127	1050	1060	1003	1143	1325	3526	3750	3302.08	3224.93	3150.11
200 clients	1114	1127	1050	1120	1120	1203	1198	3374	3482	3230.64	2927.39	2591.07
500 clients	1114	1129	1050	1102	1100	1276	1165	3850	3874	3594.35	3510.36	3428.92

As the diversity of clients increases compared to the cache size, caching tends to be less effective.

Memory is accessed more often. Whilst it was expected for the write-through cache to perform poorly for small sizes and better for large sizes, the write-through suffers from more delays than the write-back protocol for the same large cache sizes. The difference between the number of orders sent by the two caches seems minimal for small sizes. Such behaviour is explained because both models are inefficient for small cache sizes – they behave like memory. As the y-axis values increase, the differences between the two methods become more prevalent.

Figure 7.11: Comparison of write-through and write back cache efficiency for a subset of 50,100 and 200 clients traded.



Write-back performed better than write-through for all cases tested in figure 7.11. The data was sent at a fast rate, and the same cache line had to be accessed multiple times.

Write-through caches are particularly slow for sequential, recurring data. For example, if the testbench adds £100 to the same client five times in a row, the write-through cache must write

the cache data back to memory continuously. This consumes significant bandwidth and results in significant delays. In this example, the write-through cache suffers from an extra $4 \times 7 = 21$ cycles penalty from writing to memory compared to write-back.

Testing focused on a small subset of repeated orders, for which a write-through cache is not optimised. The former always outperformed the latter when writing to the same memory location frequently. The possible risks of losing data when using write-through apply in the case of a power cut or multiple processes writing to the same memory; this is not the case for this project.

Effectively, write-back performed better than write-through cache since it can write to the cache at the logic rate rather than having to wait for this to be written to the main memory every time a cache line is dirty.

7.5 Parameters studied

The size of the cache, the method of initialization, and the cache layout were all investigated. The results were identical for different values in the cache at time zero for any maximum greater than the minimum order size. This is due to the downstream processor's ability to send cancellations at any time, allowing traders to send more orders and complete operations. As a result, if the memory block layout at the start of the day is well-defined and greater than a single order, the cache performance is unaffected. When the maximum was set to 0, there was a slight increase in orders sent across all models because some orders were skipped until the maximum was set and orders could be sent.

Model	*Performance*	*Explanation*
Baseline	22.34%	7 cycles delay for every order
Write-back/through <64 bits	20-40%	Majority of orders must be written in memory.
Write-through > 64 bits	60%	Consecutive orders for the same clients require a memory operation.
Write-back cache >64 bits	70-90%	The cache can hold most orders
Write-back cache zero-initialised >64 bits	70-90%	The maximum and order history does not affect performance if greater than zero. The processor has to wait for a new maximum to start sending orders.

Table 7.2: Overview of different parameters and their effect on their performance: rate of orders sent/received.

7.6 Best performing model

From these results, caching speeds up the baseline model. Table 7.3 shows suggested minimum and maximum bounds for the target interval of 50–250 clients repeatedly traded in one second. The safest and best-performing cache size is 96 bits.

Number of clients	*Cache lower bound*	*Cache upper bound*	Recommended size
Minimum: 50	64	96	64
Median : 150	64	128	96
Maximum: 200	64	128	96

Table 7.3: Estimated lower and upper bound for a write-back cache, and recommended size rounded to the nearest power of 2 .

Chapter 8

Conclusion & Further Work

8.1 Summary of Thesis Achievements

A complete LOB was implemented in SystemVerilog based on assumptions and models discussed with Morgan Stanley. The use of an OOP testbench, FSM for cache control, and SystemVerilog structures were all challenges that resulted in a well-designed and compact final model. The 96-bit write-back cache model outperformed the baseline model by a factor of 2.5 for a period of 1 second when repeated trades were made on the same subset of 50 to 150 stocks (clients). When compared to the memory model, smaller caches of up to 32 bits provide a speedup of a factor of two. The results of the constrained-random testing provided confidence in all models because all input combinations tested resulted in defined expected behaviour with correct outputs.

8.2 Applications

The direct application of this project will occur in the following year, as the model will be tested on an FPGA board and modified accordingly. This opens other possibilities in the domain of graphic processing or live video processing where data has to be read quickly. Caching could be used to speed up the recognition of hand gestures or facial recognition on FPGAs, where the

background is constant and can be stored in memory while the object of interest is accessed often and changes position.

8.3 Future Work

The future workload can be split into two major categories: improvements to the LOB model and improvements to the testbench.

Indexing

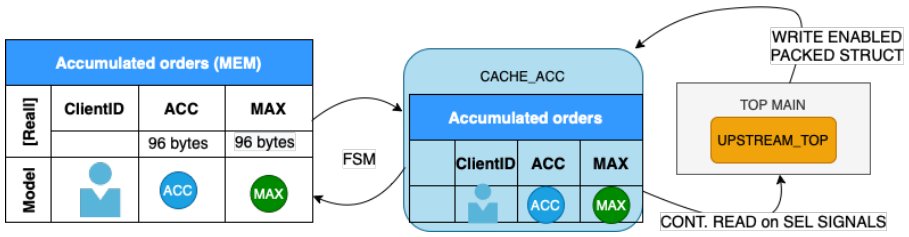
The maximum to trade and accumulated order values are currently stored in the upstream cache. It would be possible to read and write to both of these caches if they were split in half. The added complexity of managing extra logic between caches may or may not outweigh the added value of parallelism between the two groups of data, possibly changing the conclusions of this project.

It may also be useful to extra an entire line with the same index for the upstream memory which always needs to read both the maximum to trade and accumulated orders value at all times. Figure 8.1b shows an example of the different designs.

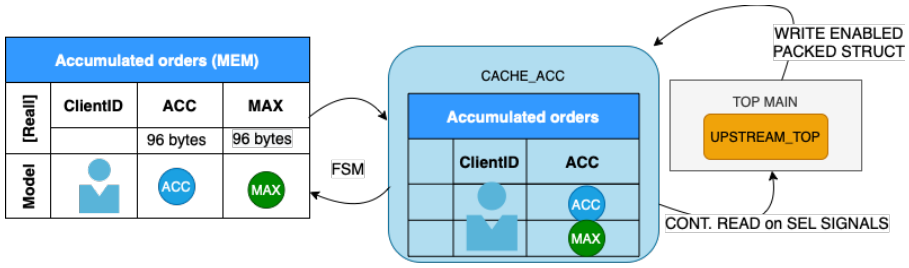
Splitting upstream cache and memory

A more complex model

It would be particularly useful to modify the model and remove some simplifications to be closer to the structure of an investment bank. The arbiter, optional in this project, adds redundancy to the cache. Companies such as CitiBank, Morgan Stanley, and J.P. Morgan use an abiter to pass data that has just been written from one processor to another and from one order to the next. Implementing it would be useful to study its effect on the performance gap between



(a) Same index cache



(b) Different indexes cache

Figure 8.1: Cache layout within the trading moel

the models. If the data mostly repeats itself, this may reduce the performance of the cache compared to memory only.

Repeating caches

Other improvements would have the aim of creating the optimal LOB memory and cache control.

1. A similar improvement consists in repeating existing caches. Instead of one of each – upstream and downstream – cache, several instances could be implemented. Each instance would represent a set of frequently traded clients, and a logic module could switch between the caches when it detects a new set of clients. The memory could be updated during off-hours since the caches would hold values for different clients. Rather than the caches themselves, the processors would interact with the cache handling module. This could result in a significant speedup.

2. An improvement based on the point above would add data exploration using machine learning to plan beforehand which set of clients must be loaded in the cache. The model could use reinforcement learning and continuously learn from trades, improving day by

day. As a result, the set of clients would be predicted for each change in sociopolitical factors. While this may sound like one of the most complex additions, it would be easier in practice to add it to the existing machine learning model used by firms to study and predict the market, which is already connected with the FPGA LOB.

Formal Verification

Morgan Stanley plans to make some improvements to this project. Formal Verification is the final step in the model testing process, as complete input freedom provides complete confidence in the model. As a first step, JasperGold will be used, and then the model will be written in TLA+ for optimal confidence.

Whilst constrained random testing tests several combinations, it has set constraints and a reduced number of iterations. The last stage at which the design is tested could assure that all possible combinations produce the correct output. This uses the same SystemVerilog Assertions as functional verification, without the constraints.

From the stimulus assumptions, formal verification could test all combinations but the ones set in the risk assessment tables for each SystemVerilog module (the testbench coverage will not be 100%).

To conclude, when compared to software alone, programmable hardware provides significant benefits. Engineers can now visualise the results of FPGA design and synthesis using a variety of tools. It might be helpful to compare this cached model to the existing one by fully implementing it on custom hardware. The current software and programmable hardware trading model could be replaced by software, programmable hardware, and custom hardware. Several banks are considering this option, as two electronics labs have been added to their offices in the last two years.

Chapter 9

User Guide

9.1 Prerequisites

Compilation: Questasim simulation software is needed to run most of the project. It can be downloaded or run through one of the labs machines:

1. Access machine `login@ee-mill3.ee.ic.ac.uk`, through ssh (enable X-11 forwarding),
2. Alternative - asking to the department for access to `login@ee-beholder1.ee.ic.ac.uk`
3. Run `source /usr/local/mentor/QUESTA-CORE-PRIME_10.7c/settings.sh` to load the software.

On windows, X-11 forwarding can be enabled with Mobaxterm. X-11 forwarding is used to open a Questasim graphics window. Links:

1. Icarus verilog (for basic module tests): http://iverilog.icarus.com/
2. Verilog Compiler (Questasim) : https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/
3. MobaXterm : https://mobaxterm.mobatek.net/

9.2 Accessing the project

The project can be cloned from Github:

1. Navigate to the main page of the repository, https://github.com/chocovore17/fpga-caching-fyp
2. Above the list of files, click Code.
3. To clone the repository using HTTPS, under "Clone with HTTPS", click . To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click Use SSH, then click . To clone a repository using GitHub CLI, click Use GitHub CLI, then click to copy.
4. Type git clone, and then paste the URL copied earlier.

```
git clone https://github.com/YOUR-USERNAME/fpga-caching-fyp
> Cloning into ‘fpga-caching-fyp‘...
> remote: Counting objects: 10, done.
> remote: Compressing objects: 100% (8/8), done.
> remove: Total 10 (delta 1), reused 10 (delta 1)
> Unpacking objects: 100% (10/10), done.
```

9.3 Running the project

Details on how to change the cache size can be found in chapter 5. To run the model under `code/*` and print the output on the command line. This command can be run to call the main bash script: `questasim_run_top.sh`. The `-c` option will additionally generate a coverage output webpage to print out assertions and coverage statistics. The listing below shows the command called by this script. The `gui` line can be added back to open questasim in a separate window and see the waveforms.

Listing 9.1: Script to run the project

```
vlib work
if [  -z $1 ]
then
vlog -work work +acc=blnr -noincr -timescale 1ns/1ps
    testbench/unit_level/UPDOWNSTREAM/tb_top_UPDOWNSTREAM.sv
    code/shared/top.sv
vopt -work work tb_top_UPDOWNSTREAM  -o seg_work_opt
vsim seg_work_opt -c -logfile testbench/outputs/top/UPDOWNSTREAM_sim.txt
    -do "run -all; quit"

else
# run converage check on questasim
rm -r covhtmlreport_toplevel/
rm testbench/outputs/top/UPDOWNSTREAM_cov_report.txt
vlog -work work +acc=blnr -noincr -timescale 1ns/1ps
    testbench/unit_level/UPDOWNSTREAM/tb_top_UPDOWNSTREAM.sv
    code/shared/top.sv
vopt -work work tb_top_UPDOWNSTREAM  -o seg_work_opt
vsim -coverage seg_work_opt -c
    -logfile testbench/outputs/top/UPDOWNSTREAM_sim.txt  -do "run -all;
    coverage report -file
    testbench/outputs/top/UPDOWNSTREAM_cov_report.txt -byfile -assert
    -directive -cvg -codeAll; coverage save -onexit -assert -directive
    -cvg -codeAll testbench/outputs/top/updown.ucdb; quit"

# vsim -coverage seg_work_opt -gui -do "run -all; run 1500ns"
vcover report -details -html testbench/outputs/top/updown.ucdb
# firefox covhtmlreport_toplevel/index.html
fi
```

Chapter 10

Appendix

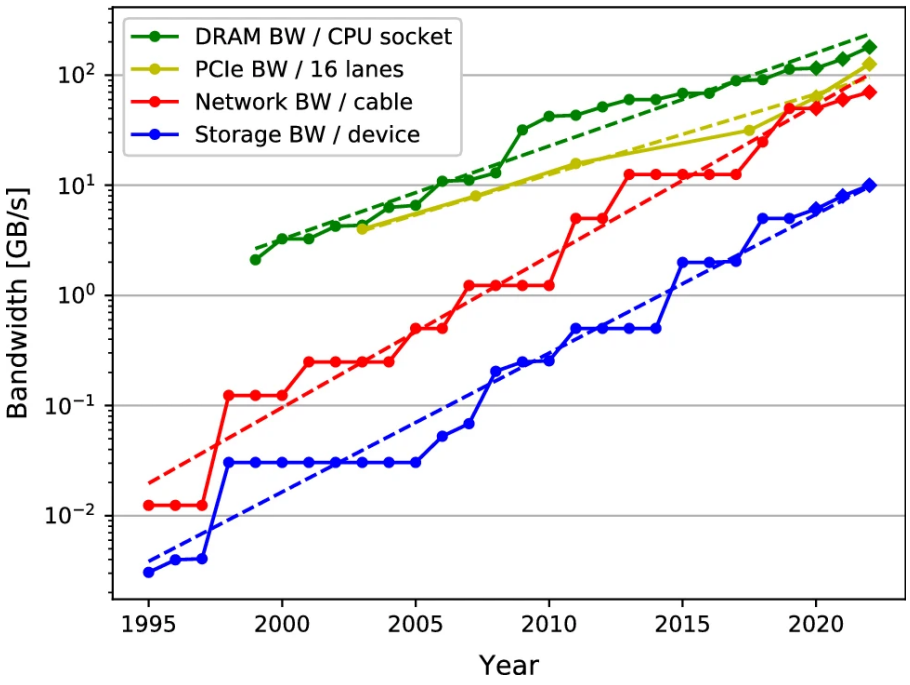


Figure 10.1: Bandwidth trends at device-level. Data points were approximated from the referenced figures in order to add the PCI Express standard bandwidth and represent all bandwidths in GB/s[8]

Mem source	Mem type	BW (GB/s)	Latency (ns)	Capacity (MB)
Internal	BRAM	$\geq 10^3$	10^0	10^0
	URAM	$\geq 10^3$	10^1	10^1
On-board	HBM	10^2–10^3	10^1–10^2	10^3
	DRAM	10^1–10^2	10^1–10^2	10^4
Host	DRAM	10^1	$\geq 10^2$	$\geq 10^5$

Figure 10.2: FPGA-related bandwidth and latency. [8]

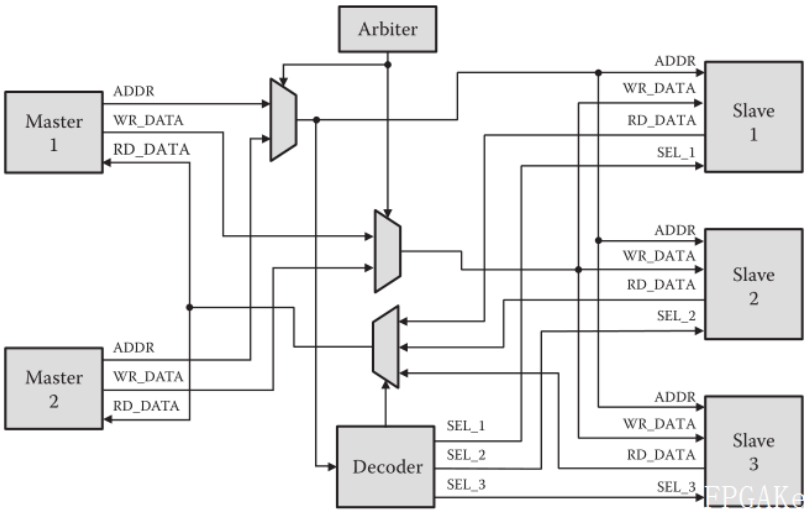


Figure 10.3: AHB bus structure according to AMBA 2 specification. Source FPGA Key website

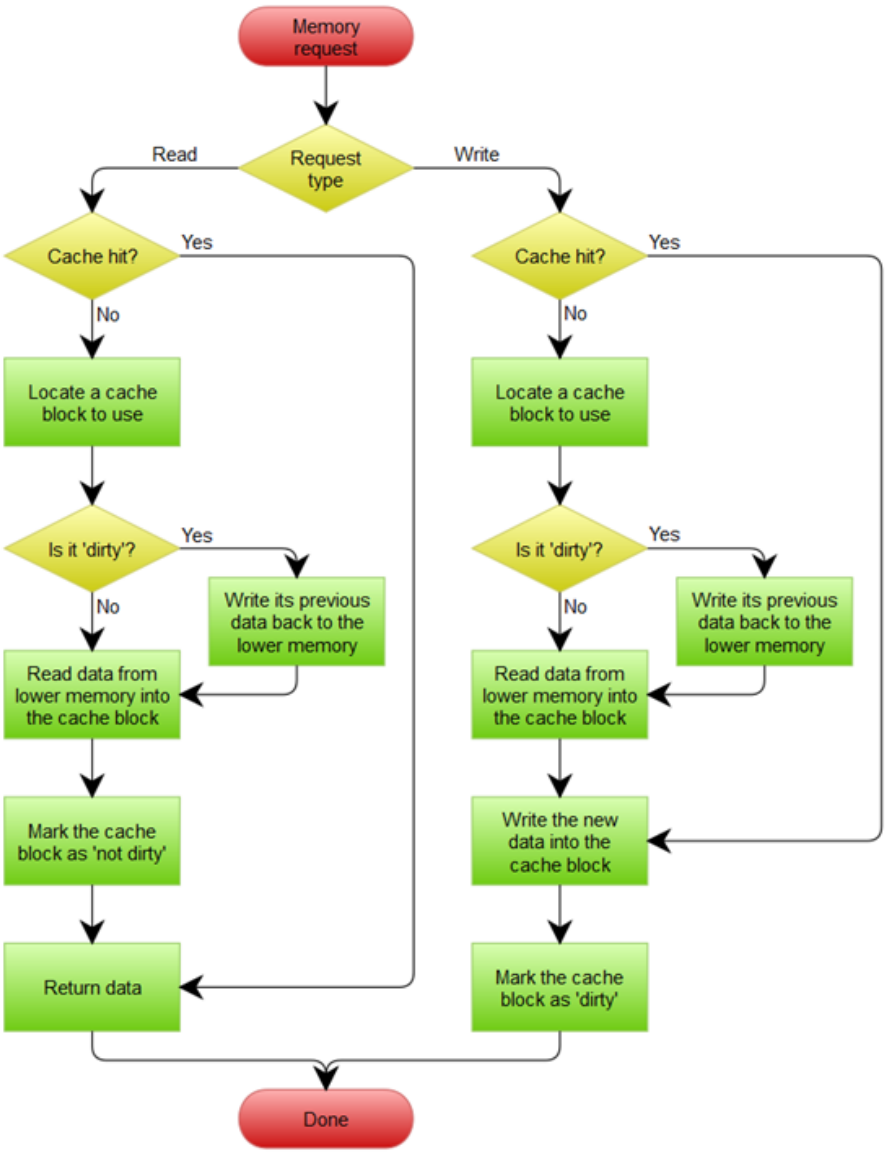


Figure 10.4: Flowchart for a write back cache with write allocate method to handle write miss. Source: Huawai

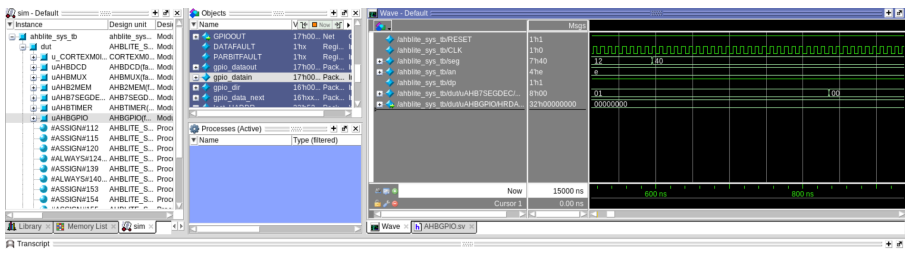


Figure 10.5: Questasim GUI output for the memory model

10.1 Project development

Figure 10.6 contains a detailed GANTT chart with each phase of the project matching the deliverables shown in table 3.1. The colour code matches the importance of each entry, from crucial to the project (red) to optional (green).

TASK	PROGRESS	START	END
Initial Phase			
Propose project	100%		
Find supervisor	100%		
Presentation, GANTT Chart, flowchart, diagrams	100%		
Organisation and structure of project	90%		
2. Research and baseline implementation phase			
Research MESI protocol & snooping	50%	9/15/21	9/18/21
Familiarise myself with Verilog	100%	9/18/21	9/20/21
Research on past caching attemps in verilog	90%	9/20/21	9/24/21
Code development and debugging for the baseline model	80%	9/20/21	1/3/22
Testing	50%	10/4/21	1/13/22
Verification	10%	9/19/21	9/21/21
3. Testing Phases			
Unit tests for single modules	100%	10/4/21	10/8/21
Constrained Random testing testbench with driver	50%	11/23/21	11/28/21
Formal and showing alternatives	0%	1/3/22	2/2/22
Creating and collecting data + graphs	10%	9/20/21	6/17/22
Conclusions and comparison (e.g same cycle with different memory model)	0%	2/2/22	2/22/22
4.Cache implementation phases			
Block diagrams, state machines, logic	25%	2/22/22	3/9/22
MESI protocol research & questions	2%	2/22/22	3/9/22
Code development & debugging	0%	3/9/22	4/2/22
Testing	0%	4/2/22	4/16/22
Review & change (e.g same cycle with different cache model)	0%	4/16/22	5/21/22
4. Producing the reports			
Gathering diagrams for the reports	8%	1/13/22	4/16/22
Meetings/emails with Morgan Stanley for clarifications	14%	9/15/21	7/20/22
Meetings/emails with supervisor	20%	9/15/21	7/20/22
Writing Interim Report	12%	1/3/22	1/23/22
Writing Final Report		5/21/22	6/15/22

Figure 10.6: GANTT chart of the project

There are 2 to 3 cycles of implementation for this project shown in figure 10.7

1. Building the baseline model and baseline testbench to collect the reference data and

graphs.

Critical to the project.

2. Modifying the model to add caching with a cache coherency protocol and testing it to compare to the baseline

   Needed for the project.

3. Try to split the accumulated risk limit memory into accumulated risk and maximum to trade blocks to assess the performance when these are separate DPRAMs or separate cache blocks.

   Optional to the project.

Success is to be evaluated based on these phases; assessing how faithful the baseline model is compared to the existing one, then drawing conclusion from the comparison of the memory and cache(s) model performance.

Figure 10.7: Cycles of Implementation. The rightmost elements are less essential to the project than the leftmost ones.

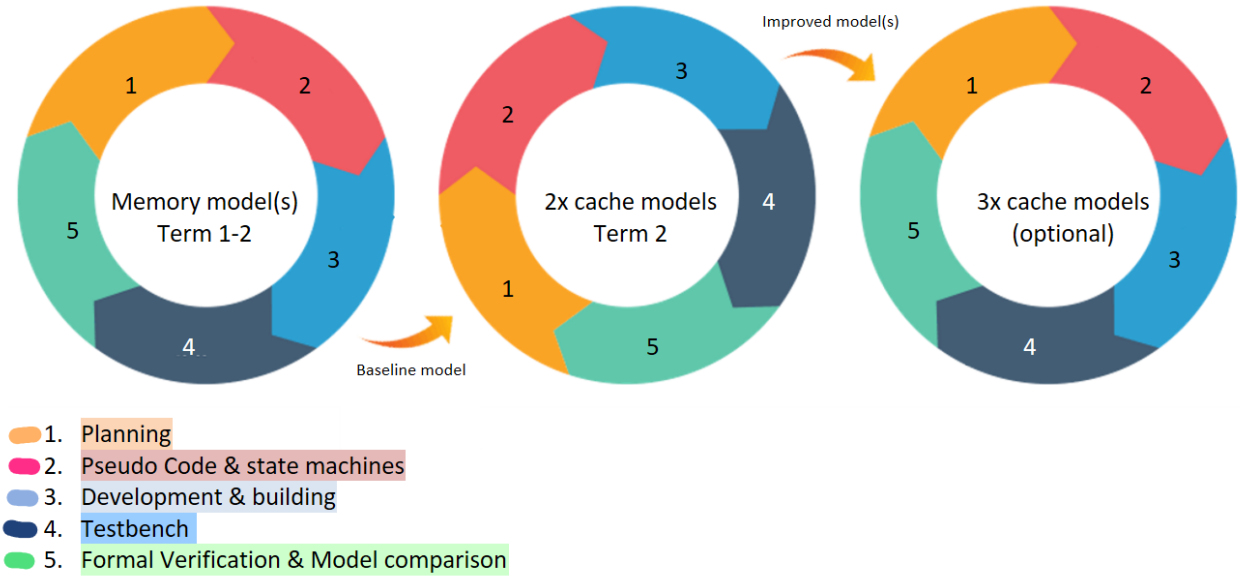


Figure 10.8: State transition table for the downstream processor. The states are defined on the Moore design 4.7

Curr State	Cancel ACK	MemWr	NextState
0	0	X	0
0	1	X	1
1	X	0	1
1	X	1	0

Listing 10.1: Example of a unit test using Icarus Verilog.

```
// Testbench
module test;

  reg  clk, ack, memwr;
  wire out;

  // Instantiate device under test
  fsm DOWNSTREAMPROCESSOR(.clk(clk),
          .ack(ack),
          .memwr(memwr),
          .out(out);

  initial begin
    // Dump waves
    $dumpfile("../outputs/Module/upstream.vcd");
    $dumpvars(1, test);

    clk = 0;
    ack = 0;
    memwr = 0;
    $display("Initial out: %0h", out);

    toggle_clk;
    $display("IDLE  out: %0h", out);

    ack = 1;
    toggle_clk;
    $display("STATE_1  out: %0h", out);

    memwr  = 1;
    toggle_clk;
    $display("IDLE  out: %0h", out, " memwr is 1, ack 1");

    ack = 0;
```

State=0	Risk check	New order	New max	MemWr	NextState
000	0	0	0	0	0
000	0	0	0	1	0
...	0	0	1	0	3
	0	0	1	1	3
	0	1	0	0	1
	0	1	0	1	1
	0	1	1	0	4
	0	1	1	1	4
	1	0	0	0	0
	1	0	0	1	0
	1	0	1	0	3
	1	0	1	1	3
	1	1	0	0	1
	1	1	0	1	1
	1	1	1	0	4
	1	1	1	1	4

(a) State 0.

State=1	Risk check	New order	New max	MemWr	NextState
001	0	0	0	0	2
001	0	0	0	1	2
...	0	0	1	0	4
	0	0	1	1	4
	0	1	0	0	5
	0	1	0	1	5
	0	1	1	0	4
	0	1	1	1	4
	1	0	0	0	0
	1	0	0	1	0
	1	0	1	0	3
	1	0	1	1	3
	1	1	0	0	1
	1	1	0	1	1
	1	1	1	0	4
	1	1	1	1	4

(b) State 1.

State=2	Risk check	New order	New max	MemWr	NextState
010	0	0	0	0	2
010	0	0	0	1	0
...	0	0	1	0	3
	0	0	1	1	3
	0	1	0	0	1
	0	1	0	1	1
	0	1	1	0	2
	0	1	1	1	4
	1	0	0	0	2
	1	0	0	1	0
	1	0	1	0	3
	1	0	1	1	3
	1	1	0	0	1
	1	1	0	1	1
	1	1	1	0	2
	1	1	1	1	4

(c) State 2.

State=3	Risk check	New order	New max	MemWr	NextState
011	0	0	0	0	3
011	0	0	0	1	0
...	0	0	1	0	3
	0	0	1	1	3
	0	1	0	0	4
	0	1	0	1	1
	0	1	1	0	4
	0	1	1	1	4
	1	0	0	0	3
	1	0	0	1	0
	1	0	1	0	3
	1	0	1	1	3
	1	1	0	0	4
	1	1	0	1	1
	1	1	1	0	4
	1	1	1	1	4

(d) State 3.

State=4	Risk check	New order	New max	MemWr	NextState
100	0	0	0	0	4
100	0	0	0	1	1
...	0	0	1	0	4
	0	0	1	1	4 (priority to max)
	0	1	0	0	4
	0	1	0	1	1
	0	1	1	0	4
	0	1	1	1	4
	1	0	0	0	4
	1	0	0	1	3
	1	0	1	0	3
	1	0	1	1	3
	1	1	0	0	4
	1	1	0	1	4
	1	1	1	0	4
	1	1	1	1	4

(e) State 4.

State=5	Risk check	New order	New max	MemWr	NextState
101	0	0	0	0	5
101	0	0	0	1	1
...	0	0	1	0	5
	0	0	1	1	4
	0	1	0	0	5
	0	1	0	1	1
	0	1	1	0	5
	0	1	1	1	5
	1	0	0	0	2
	1	0	0	1	0
	1	0	1	0	x
	1	0	1	1	3
	1	1	0	0	2
	1	1	0	1	1
	1	1	1	0	x
	1	1	1	1	4

(f) State 5.

Figure 10.9: State transition tables for the upstream processor. The states are defined on the Moore design 4.8.

```
35      toggle_clk;
36      $display("IDLE  out: %0h", out, " ack back to 0");
37    end
38
39    task toggle_clk;
40      begin
41        #10 clk = ~clk;
42        #10 clk = ~clk;
43      end
44    endtask
45
46  endmodule
```

Glossary

contingency plan live streaming of trade-related data. It encompasses a range of information such as price, bid/ask quotes and market volume. Trading venues provide reports on various assets and financial instruments, which are then distributed to traders and firms. IG Glossary.. 80, 132

Dow Jones Industrial Average price-weighted measurement stock market index of 30 prominent companies listed on stock exchanges in the United States. 1

Field Programmable Gate Array definition by Xilinx: A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term field-programmable. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC). Circuit diagrams were previously used to specify the configuration, but this is increasingly rare due to the advent of electronic design automation tools.. 19

functional verification the task of ensuring that the logic design meets the specifications in electronic design automation. In most large electronic system design projects, this is a difficult task that consumes the majority of time and effort. Functional verification is a subset of broader design verification, which includes non-functional aspects such as timing, layout, and power, in addition to functional verification. This effort is comparable to program verification, and it is NP-hard or worse – and no solution that works well in all cases has been found.. 3

high-frequency trading a method of trading that uses powerful computer programs to transact a large number of orders in fractions of a second. It uses complex algorithms to analyze multiple markets and execute orders based on market conditions. Typically, the traders with the fastest execution speeds are more profitable than traders with slower execution speeds. *Source:* Investopedia High-Frequency Trading *What Is High-Frequency Trading (HFT)? By James Chen, Updated August 25, 2021 Reviewed by Michael J Boyle*. 11

market data live streaming of trade-related data. It encompasses a range of information such as price, bid/ask quotes and market volume. Trading venues provide reports on various assets and financial instruments, which are then distributed to traders and firms. Market data is available across thousands of global markets, including stocks, indices, forex and commodities. IG Glossary. 2, 19

systematic risk refers to the risk inherent to the entire market or market segment. Systematic risk, also known as "undiversifiable risk," "volatility" or "market risk," affects the overall market, not just a particular stock or industry.. 29

systematic risk it is the risk that a company- or industry-level risk could trigger a huge collapse. Systemic risk refers to the risk of a breakdown of an entire system rather than simply the failure of individual parts. In a financial context, it denotes the risk of a cascading failure in the financial sector, caused by linkages within the financial system, resulting in a severe economic downturn.. 29

tick size the minimum price amount a security can move in an exchange. It's expressed in decimal points, which in U.S. markets is $0.01 for stocks. 10

tick-to-trade time interval between receiving a market 'tick' (a price movement in the market) presenting the opportunity to the algorithm, and processing the buy or sell order. The time taken to respond to incoming market data – tick to trade – determines how competitive trading can be.. 2

Transmission Control Protocol handshake TCP uses a three-way handshake to establish a reliable connection. The connection is full duplex, and both sides synchronize (SYN)

and acknowledge (ACK) each other. The exchange of these four flags is performed in three steps—SYN, SYN-ACK, and ACK. 19

User Datagram Protocol (UDP) one of the core members of the Internet protocol suite. With UDP, computer applications can send messages, in this case referred to as datagrams, to other hosts on an Internet Protocol (IP) network. Prior communications are not required in order to set up communication channels or data paths.. 19, 20

verification testing Verification testing is a black-box testing strategy at either the system or subsystem level, and is performed to ensure that the system under test meets its requirements"[43].. 30

write-back cache type that enables control of the time and the frequency at which data is written or copied into the source backing store. In a write-back operation, any new, requested processor data is written to the cache, but not in the memory. The memory write process is only performed when the cache data needs to be edited or purged for new content. Source. Technopedia. 21, 75

write-through cache type where the data is simultaneously copied to higher level caches, backing storage or memory. It is common in processor architectures that perform a write operation on cache and backing stores at the same time. In each write-through operation, data that is brought into the cache is also written into the backing store, which is the primary memory (RAM, in most cases). Write-through cache also helps with data recovery, as the data in operation is written to both cache and memory.Source: Technopedia. 21, 22

Bibliography

[1] C. F. Institute, "2010 flash crash, the stock market crash of march 6, 2010." https://corporatefinanceinstitute.com/resources/knowledge/trading-investing/2010-flash-crash/.

[2] J. Breckenfelder, "How does competition among high-frequency traders affect market liquidity?," 2020. Research Bulletin No. 78, https://www.ecb.europa.eu/pub/economic-research/resbull/2020/html/ecb.rb201215 210477c6b0.en.html 1/5.

[3] N. Shiroha, "A brief history of algorithmic trading," *Capital*, 2020. hrefhttps://altbtc.cc/story/a-brief-history-of-algorithmic-tradingsources.

[4] "High-speed finance looks to flexible hardware," 2019.

[5] M. Dvořák and J. Kořenek, "Low latency book handling in fpga for high frequency trading," *17th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, pp. 175–178, 2014.

[6] C. He, H. Fu, W. Luk, W. Li, and G. Yang, "Exploring the potential of reconfigurable platforms for order book update," *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, 2017.

[7] M. Asiatici and P. Ienne, "Large-scale graph processing on fpgas with caches for thousands of simultaneous misses," 2021. EPFL, DOI: 10.1109/ISCA52012.2021.00054.

[8] J. Fang, Y. Mulder, J. Hidders, J. Lee, and P. Hofstee, "In-memory database acceleration on fpgas: a survey," *the VLDB Journal*, 2019. 29:33–59 https://doi.org/10.1007/s00778-019-00581-w.

[9] A. N. SLOSS, D. SYMES, and C. WRIGHT, "Chapter 12 - caches," *ARM System Developer's Guide - The Morgan Kaufmann Series in Computer Architecture and Design*, pp. 402–459, 2004.

[10] S. Circuit, "Write-through vs write-back," 2018. Website year 3 Embedded Systems lecture.

[11] B. Jacob, S. W. Ng, and D. T. Wang, "Memory systems, chapter 4 - management of cache consistency," *Science Direct*, pp. 217–256, 2008.

[12] R. Cofer and B. F. Harding in *Embedded Technology, Rapid System Prototyping with FPGAs* (Newnes, ed.), ch. 16, pp. 227–236, 2006.

[13] K. E. Murray, K. Petelin, O. Zhong, and S. Wang, "Vtr 8: High-performance cad and customizable fpga architecture modelling.," *ACM Transactions on Reconfigurable Technology and Systems.*, 2012. 13. 1-55. 10.1145/3388617.

[14] L. Kekely, J. Cabal, V. Pus, and J. Koˇrenek, "Multi buses: Theory and practical considerations of data bus width scaling in fpgas," *23rd Euromicro Conference on Digital System Design (DSD)*, 2020. 978-1-7281-9535-3/20 ©2020 IEEE DOI 10.1109/DSD51259.2020.00020.

[15] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "High bandwidth memory on fpgas: A data analytics perspective," *IBM Research*, 2020. Systems Group, Department of Computer Science, ETH, Switzerland.

[16] Xillinx, "High-performance, lower-power memory interfaces with the ultrascale architecture (wp454)," 2015.

[17] NASDAQ, "Nasdaq Total view report on how to build a fast limit order book in c++." Overview of the stock market for traders contaning buy orders and sell orders. This is a software platform containing information about opening and closing balances.

[18] C. Cao, O. Hansch, and X. Wang, "The information content of an open limit-order book," *The Journal of Futures Markets*, vol. 29, no. 1, pp. 16–41, 2009. Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/fut.20334.

[19] H. Bessembinder and K. Venkataraman, "Does an electronic stock exchange need an upstairs market?," *Journal of Financial Economics*, vol. 73, pp. 3–36, July 2004.

[20] M. Avellaneda and S. Stoikov, "High-frequency trading in a limit order book," *Quantitative Finance*, vol. 8, no. 3, pp. 217–224, 2008. Mathematics, New York University, 251 Mercer Street, New York, NY 10012, USA.

[21] M. Glantz and R. Kissell, "Multi-asset risk modeling: Techniques for a global economy in an electronic and algorithmic trading era," 2013. ISBN 978-0124016903.

[22] T. R. Electronics, "A historical constant: The importance of data quality," 2012.

[23] V. Bhagat, "The growth and future of algorithmic trading," *Quant. Insti*, 2018. hrefhttps://blog.quantinsti.com/growth-future-algorithmic-trading/sources.

[24] N. Y. S. Exchange, "New york stock exchange data," 2018. https://www.nyse.com/index.

[25] "Xilings accelerated algorithmic trading." This is an internal company webpage for customers to assess the performance of FPGAs over fully-software limit order books.

[26] A. Cartea, Sebastian, J., and J. Penalva, "Algorithmic and high-frequency trading.," *Cambridge University Press*, 2015.

[27] W. Kenton, "Order book definition."

[28] WK, "Git post on how to build a fast limit order book in c++."

[29] D. Kane, A. Liu, and K. Nguyen, "Analyzing an electronic limit order book," *The R Journal Vol. 3/1*, 2011. ISSN 2073-4859.

[30] "The ultimate guide to fpga architecture." https://hardwarebee.com/the-ultimate-guide-to-fpga-architecture/.

[31] OLME and H. Company, "Lmesource, client interface specification v4.04," *The London Metal Exchange*, 2015.

[32] CSPI, "Tick-to-trade Latency definition and example.the influence of latency and importance of tick to trade in the market.."

[33] D. A. Patterson and J. L. Hennessy, "Computer organization and design," *The Hardware/Software interface.*, vol. 5th edition, 2014. CD5.9: Advanced Materials. Implementing Cache Controllers. ISBN 978-0-12-407726-3.

[34] R. Oshana, "Chapter 3 - multicore system architectures," in *Multicore Software Development Techniques* (R. Oshana, ed.), pp. 39–52, Oxford: Newnes, 2016.

[35] J. Gómez-Luna, E. Herruzo, Benavides, and J.I., "Mesi cache coherence simulator for teaching purposes," vol. 12, 2009.

[36] Nandland, "Blocking vs. nonblocking in verilog.." source: https://www.nandland.com/articles/blocking-nonblocking-verilog.html .

[37] S. Circuit, "Asynchronous memory embedded system lecture," 2019. https://www.student-circuit.com/learning/year3/embedded-systems/asynchronous-memory/.

[38] P. Milder, "A brief introduction to systemverilog.," *CSE 502 – Computer Architecture*, 2015. Stony Brooks University.

[39] S. Williams, "Icarus verilog compiler, described by its creator as "the compiler proper is intended to parse and elaborate design descriptions written to the ieee standard ieee std 1364-2005. this is a fairly large and complex standard, so it will take some time to fill all the dark alleys of the standard, but that's the goal."," 2021.

[40] M. G. Corporation, "Questa sim user manual," 1999-2001. Software Version 10.0d, Siemens product.

[41] "Systemic risk and management in finances." https://www.cfainstitute.org/en/advocacy/issues/systemic-risk.

[42] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *IEEE Computer*, vol. 34, pp. 134–137, 2001.

[43] I. A. Dongjiang You *et al.*, "Practical aspects of building a constrained random test framework for safety-critical embedded systems," *Cardiac Rhythm Disease Management, Medtronic, Inc., USA*, 2008. Paper link doi: http://dx.doi.org/10.1145/2593783.2593791.

[44] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[45] S. Pittet, "An introduction to code coverage," 2020. Software development section of Altassian webpage https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

[46] P. P. Harrod, "Hardware and software verification," 2021. ELEC97106/97107, Autumn term, Lecture 1.

[47] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," *In Testing Software and Systems*, vol. 1, p. 98–130, 2010.

[48] M. Krichen and S. Tripakis, "Conformance testing for real-time systems.," *Formal Methods in System Design*, vol. 34, no. 1, pp. 238–304, 2009.

[49] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 1, pp. 63–86, 1996.

[50] E. J. Weyuker, "On testing non-testable programs.," *The Computer Journal*, vol. 25, no. 4, pp. 458–474, 1982.

[51] I. Aldridge, "How profitable are high-frequency strategies?," 2010. http://www.huffingtonpost.com/irene-aldridge/how-profitable-are-high-f˙b˙659466.html.

[52] R. Wagner and D. K. T. Bhala, "High frequency trading." https://sevenpillarsinstitute.org/case-studies/high-frequency-trading/.

[53] U. Government, "The markets in financial instruments (amendment) (eu exit) regulations 2018," 2018. https://www.legislation.gov.uk/uksi/2018/1403/contents/made.

[54] ""IEEE Standard for Software Safety Plans.,"," 1993. doi: 10.1109/IEEESTD.1994.122165.

[55] Y. Park, "High performance computing team of morgan stanley." face-to-face interaction to collect information about the requirements of the project.

[56] "Github blog: Synchronous vs asynchronous processing," 2021. https://dsinecos.github.io/blog/Synchronous-vs-Asynchronous-processing.

[57] P. Zhang, "Chapter 5 - microprocessors," in *Advanced Industrial Control Technology* (P. Zhang, ed.), pp. 155–214, Oxford: William Andrew Publishing, 2010.

[58] ChipVerify, "Verilog always block." https://www.chipverify.com/verilog/verilog-always-block.

[59] M. K. Patel, "Systemverilog for synthesis," 2017. source : Read The Docs, https://verilogguide.readthedocs.io/en/latest/verilog/systemverilog.html.

[60] C. Spear, "SystemVerilog for Verification, a guide to learning the testbench language features.," *Synopsys, Inc. Ed. Springer*, 2016. TLibrary of Congress Control Number: 2006926262, ISBN-10: 0-387-27036-1.

[61] K. Huang, M. Gungor, S. Ioannidis, and M. Leeser, "Optimizing use of different types of memory for fpgas in high performance computing," *National Science Foundation under grant CNS-1717213*, 2020. 978-1-7281-9219-2/20.

Quotes

"The working frequency of current FPGAs can reach several hundreds of MHz, however, it is unusual to see any complex FPGA logic running at more than 500 MHz. The need to increase the achievable frequency is documented by works of major FPGA manufacturers in their latest chip families. Xilinx came up with the Time Borrowing concept for UltraScale+ FPGAs [1]. It promises average maximal frequency (Fmax) increase of 5.5 changes. Intel introduced the HyperRegisters [2] inside its Stratix 10 FPGAs. These registers are spread across the FPGA routing fabric and they are used to balance critical paths during routing (when retiming is performed). The promised Fmax improvement is up to 2× in some cases. Even with these advances, practical FPGA firmware running at frequencies over 1 GHz is not expected in the foreseeable future."
Multi Buses: Theory and Practical Considerations of Data Bus Width Scaling in FPGAs [14]

'an electronic list of buy and sell orders for a specific security of financial instrument organised by price level. An Order book lists the number of shared being bid on or offered at each price point, or market depth. It also identified the market participants behind the buy and sell orders. These lists help traders and improve market transparency because they provide valuable trading information'
Investopedia, High Frequency Trading [41]

'Details of the order-book information is available to brokers and institutional investors , whereas the aggregate version is representative of what online traders would be able to see'
'Morgan Stanley for providing us the equity market microstructure research grant for this study'
"The information content of an open Limit Order book", Charles Cao, Olivier Hansch and Xiaoxin Wang (2009)

'For hardware, random fault injection on constrained hardware blocks enables the evaluation of the architectural vulnerability factor (AVF), which is a measure of the effect of single event upsets (SEUs) on system behavior. Each SEU is simulated through a fault injection at a random hardware location at a random time, and is checked against an oracle'
"Practical Aspects of Building a Constrained Random Test Framework for Safety-Critical Embedded Systems", Dongjiang You, , Isaac Amundson (2014)

'In this way SystemVerilog ensures that the software tools will infer the same sensitivity list'
"Systemverilog for synthesis", M. K. Patel,2017.

‘In addition to invalidating outstanding copies of a cache block that is being written into, we also need to locate a data item when a cache miss occurs. In a write-through cache, it is easy to find the recent value of a data item, since all written data are always sent to the memory, from which the most recent value of a data item can always be fetched. In a design with adequate memory bandwidth to support the write traffic from the processors, using write-through simplifies the implementation of cache coherence. ’
“Computer organization and design, 5th edition“ D. A. Patterson and J. L. Hennessy (2014)

“While FPGAs have high intrinsic parallelism and very high internal bandwidth to speed up kernel workloads, the low interface bandwidth between the accelerator and the rest of the system has now become a bottleneck in high-bandwidth in-memory databases. Often, the cost of moving data between main memory and the FPGA outweighs the computational benefits of the FPGA. Consequently, it is a challenge for FPGAs to provide obvious system speedup, and only a few computation-intensive applications or those with data sets that are small enough to fit in the high-bandwidth on-FPGA distributed memories can benefit.”
“In-memory database acceleration on FPGAs: a survey”, Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee and H. Peter Hofstee, 2020

“The authors are grateful to the Australian Stock Exchange and the Securities Industry Research Centre Asia-Pacific for providing the data used in this study. The authors also thank Morgan Stanley for providing the equity market microstructure research grant for this study.”
“The information contents of an Open-limit order book”, Charles Cao, Olivier Hansch and Xiaoxin Wang (2010). Acknowledgements.

List of Tables

1.1 Daily characteristics of the 100 Most Actively Traded Stocks on the Australian Stock Exchange [18] . . . 7

2.1 Write-back and write-through cache features. . . . 21
2.2 MESI protocol transitions. . . . 23
2.3 Pros and Cons of using Moore vs Mealy design in Algorithmic trading. . . . 27

3.1 Deliverables and their importance for the completion of the project. . . . 36
3.2 Legend for table 3.1 . . . 36
3.3 Ethical considerations in HFT. Summarised from Seven Pillars website [52]. . . . 39

4.1 Causes of delays within the scope of the trading model and simplifications. . . . 45
4.2 Comparison of using data and control bits versus packed SystemVerilog structure for design coherence. . . . 49
4.3 Simultaneous write risks, ✎=simultaneous write. Red indicates that the block has to be locked, green shows that the data can be written safely . . . 61
4.4 Simultaneous read and read write risks, 👓=simultaneous read Red indicates that the block has to be locked, orange shows that both items will be read/written at the same time; they share the same address in memory . . . 61

5.1 Comparative table of always statements in SystemVerilog . . . 78

6.1 Legend for table 6.2 . . . 80
6.2 Contingency plan for the the top-level module (1) showing time, performance and cost risks. Please see legend 6.1 for code colour. . . . 80
6.3 **Constrained inputs** for the top-level module. Please see figure for an overview. . . . 86
6.4 **Constrained inputs** for the downstream processor top-level module. Please see figure for an overview. . . . 86
6.5 **Constrained inputs** for the upstream processor top-level module. Please see figure for an overview. . . . 86

7.1 Results of different models on constrained-random testing and their efficiency. x represents the number of orders sent. . . . 99
7.2 Overview of different parameters and their effect on their performance: rate of orders sent/received. . . . 105

7.3 Estimated lower and upper bound for a write-back cache, and recommended size rounded to the nearest power of 2 105

List of Figures

1.1 Overview of the project's aims. 4

1.2 Effect of Bus width on data rate. Adapted from Benjamin F. Harding and R. C. Cofer [12] . . 5

2.1 Contribution of Algorithmic trading to other global trading volumes, 2015. Chart 1 and 2 – Morton Glantz, Robert Kissell [21]; Chart 3 – Thomson Reuters.[22] 9

2.2 Timeline showing events in Algorithmic Trading relevant to this project. Green represents political events, black refers to technical improvements. full chart: https://time.graphics/line/642882 10

2.3 Different data types considered for the trading model. Any change from one to another requires a thorough assessment of the development costs against the plus-value. Figures were taken from Xilinx website. [25] 11

2.4 LOB of an example security. The quoted spread and midprice are indicated in the figure. Source: Cartea, A., Sebastian, J. and Penalva, J. [26] 11

2.5 Layout of an FPGA. Source: HardwareBee 14

2.6 Simplification of an Electronic trading model. On the left-hand side, the bank or trader combines software and hardware to find the best order to send at a competitive speed. 15

2.7 Block diagram of the memory and cache models implemented. Some simplifications have been applied. 17

2.8 Due to floating-point precision and large data width, the data for each client has to be stored at different addresses. When reading from this client, the data from each address has to be combined to retrieve the initial value. 18

2.9 Simplified flowchart of the data path for write-through and write-back caches on a cache miss. . 21

2.11 Time diagram of an asynchronous master/slave design. In this context, the processor drives the memory read/write. Data can be returned between clock edges. source: github.io 25

2.10 Asynchronous memory and logic designs allows to pipeline tasks and have a better throughput. source:github.io. 25

2.12 Moore vs Mealy design. 26

2.13 Verification and Synthesis cycle in FPGAs. Figure taken from Xilinx website. [25] 30

2.14 Visualising Directed Testing. Source: Imperial College London EE department [46] 32

2.15 Visualising Random Testing. Source: Imperial College London EE department [46] 32

2.16 Visualising Constrained Random Testing. Source: Imperial College London EE department [46] 33

4.1 Memory model after applying design choices. See figure 5.1 for the model before simplifications. 43

4.2 Caching model after applying design choices. See figure 2.7b for the cache model before simplifications. . . . 44
4.3 Risk check flowchart. . . . 47
4.4 Different data types considered for the trading model, block diagrams 48
4.5 Upstream processor block diagram. . . . 50
4.6 Downstream processor block diagram. . . . 51
4.7 Downstream processor state machine. . . . 52
4.8 Upstream processor state machine. . . . 52
4.9 Upstream state machine state diagram comparison between two models. . . . 54
4.10 Upstream state machine state diagram comparison between two models for the case of **two consecutive new orders**. . . . 55
4.11 Upstream state machine state diagram comparison between two models for the case of a **consecutive maximum update and new order**. . . . 55
4.12 Downstream processor block diagram. Visualising the tasks performed by this processor. The FSM is indicated as a purple box. . . . 57
4.13 Different memory modules for the trading model 58
4.14 FSM for the cache coherency with state transition. The variable names are the same as in the code. Source: Computer Organisation and Design, D. Patterson and Henessy [33] 60

5.1 Baseline Model block diagram. . . . 63
5.2 Cache upstream module layout. Figure inspired from Patterson and Hennessy [33] 73
5.3 Cache protocol FSM. This protocol involves MESI with snooping. . . . 77

6.1 Code coverage convergence. Verification will be a step by step process with added constraints to ensure outputs coverage. Reference: [60] 83
6.2 Testbench structure. Reference: [60] 84

7.1 Screenshot of the coverage report overview. . . . 91
7.2 Screenshots of the results for the assertions on the baseline model. . . . 92
7.3 Screenshots of the results for the coverage on the baseline model. . . . 92
7.4 Orders received vs orders sent for the 128-bit RAM SystemVerilog implementation. . . . 93
7.5 Orders received vs orders sent for a 32-bit write-back cache based on a pseudo random sequence of 50 clients. . . . 96
7.6 Orders received vs orders sent for a 12-bit write-back cache based on a pseudo random sequence of 250 clients. . . . 97
7.7 Orders received vs orders sent for a 12-bit write-back cache based on a pseudo random sequence of 20 clients. . . . 98
7.8 Orders written to the accumulated memory block of the upstream cache for different models. . 100
7.9 Comparison of different cache sizes results for a total of 5,000 orders sent. . . . 101
7.10 Write-through orders received against the cache size for an example of 5,000 orders sent. . . . 102

7.11 Comparison of write-through and write back cache efficiency for a subset of 50,100 and 200 clients traded. . . . 103

8.1 Cache layout within the trading moel . . . 108

10.1 Bandwidth trends at device-level. Data points were approximated from the referenced figures in order to add the PCI Express standard bandwidth and represent all bandwidths in GB/s[8] . 113
10.2 FPGA-related bandwidth and latency. [8] . . . 113
10.3 AHB bus structure according to AMBA 2 specification. Source FPGA Key website . . . 114
10.4 Flowchart for a write back cache with write allocate method to handle write miss. Source: Huawai115
10.5 Questasim GUI output for the memory model . . . 115
10.6 GANTT chart of the project . . . 116
10.7 Cycles of Implementation. The rightmost elements are less essential to the project than the leftmost ones. . . . 117
10.8 State transition table for the downstream processor. The states are defined on the Moore design 4.7 . . . 118
10.9 State transition tables for the upstream processor. The states are defined on the Moore design 4.8. . . . 119