POLITECNICO DI TORINO & EURECOM - TÉLÉCOM PARISTECH
Master Degree course in Electonic Engineering

Master Degree Thesis

Multimarket Feedhandler

Single FPGA solution to consume market data over multiple feeds

Supervisors
Prof. Luciano Lavagno
Prof. Sophie Coudert

Candidate
Alberto Anselmo

Novasparks
Company supervisor
Eng. Guillaume Taba

Academic year 2019-2020

Abstract

In the context of low latency solutions in the finance environment FPGAs stand as middle ground in both cost and timing. Despite being faster than software and cheaper of custom hardware the latter can still be a strong deterrent for customers. In particular one of the main constraint which renders these solutions less cost effective, and thus less desirable, is the one to one relationship between a product and a market. On the other hand its flexibility allows to follow the continuous markets migration and evolution making it the perfect medium for a long lasting product.

This internship aims to mitigate this problem by adapting the current Feed Handler architecture, provided by Novasparks, to handle multiple data feeds coming from different markets. With this goal in mind this work will propose a solution with a good balance between scalability and latency while still trying to remain flexible to account for the needs of different customers.

Résumé

Dans le contexte de solutions à faible latence pour un environnement financier, les FPGA constituent un bon compromis en termes de coûts et de délais. Bien qu'il soit plus rapide que le logiciel et moins cher que IC personnalisé, ce dernier peut toujours être un puissant moyen de dissuasion pour les clients. En particulier, l'une des principales contraintes qui rend ces solutions moins rentables, et donc moins souhaitables, est la relation un à un entre un produit et un marché. D'autre part, sa flexibilité permet de suivre la migration et l'évolution continues des marchés, ce qui en fait le support idéal pour un produit de longue durée.

Ce stage vise à atténuer ce problème en adaptant l'architecture actuelle du Feed Handler, fournie par Novasparks, pour gérer plusieurs flux de données provenant de différents marchés. Dans cet objectif, ce travail proposera une solution avec un bon équilibre entre évolutivité et latence tout en essayant de rester flexible pour tenir compte des besoins des différents clients.

Contents

Chapter 1

Introduction

1.1 The Low Latency scene

High Frequency Treading is a financial sector that can be grouped under the umbrella of algorithmic trading. It employs powerful computers to carry out complex strategies and perform large number of orders in fractions of a second. The general idea is to make up for the low margin of such a short term investments with incredibly high volumes of trades, frequently numbering in the millions.

The start of this trend in finance can be traced back to 1998 when the SCE authorized computerized trading, which was capable of executing trades 1000 times faster than humans. At that time it was really a niche market, with latencies in the order of several seconds. But with coming of the 21st Century and its advancement in transistor technology a rush started to reach ever lower latencies. By 2010 HFT had already reached the microsecond mark and it was not a niche anymore, in the US 56% of the total equity trading could be attributed to HTF. During those years, while trying to hit the nanoseconds mark[1] different solutions were explored, among which custom ICs and FPGA.
The latter in particular can provide better and more consistent latencies. One problem of computer solution is the possibility of bottleneck formation during high market activity periods, this degrades performance and generate high delay in the decision process, which is the critical part for a strategy to be profitable. Using an FPGA shows instead good consistency even when faced with a huge amount of incoming data, assuring always at least a microsecond overall latency.

Despite the advantages, in order to reach such level of performance, top

[1]The goal was reached in 2012 with a custom IC

of the line FPGAs are needed and thus the cost remains pretty high. In order to mitigate this downside, I have been trying, during the course of the internship, to improve the existing solution developed by Novasparks. The focus of my work will a component called Feed Handler in the attempt to enable the monitoring of multiple market in the same board.

1.2 About Novasparks

Novasparks is a young company, specialized in market data packet processing, algorithmic trading and complex financial product valuation acceleration. Founded in 2007 by Eric Le Rolland (CTO) and Marc Battyani. The company, previously named HPC Platform, was rebranded as Novasparks in June 2010. The headquarters are located in Paris. The company also has facilities in Nantes and in the United States with offices in New York City (NY).

Novasparks is developing products to reduce processing latency to help high frequency traders of investment banks, hedge funds and proprietary trading firms. All products have both low power and low latency characteristics in common. To provide the best performance to their customers, and to achieve critical latency processing, the technology is based on FPGAs. With their modular design, Novasparks' technical solutions are always adapted to the client's needs.

Novasparks' major product is the Feed Handler (FH), which captures market data frames (Ethernet frames), processes them and accelerates the client's trading decisions. But it is also capable of offering custom solution in order to cover a full Tick to Trade cycle, which correspond to market data processing and order execution, based on client strategies, fully in FPGA.

Chapter 2

Feed Handler

2.1 Feed Handler Overview

Order Book

Top of Book

Bid		Ask	
Price	Size	Price	Size
37.06	100	37.09	100
37.05	800	37.10	200
37.02	300	37.11	100
37.01	200	37.13	400
36.95	1,600	37.14	100

Figure 2.1. Simplified Order Book.

Every market provide a data feed specialized in broadcasting messages regarding each and every movement of the market. The job of a feed handler is to decode these messages and keep track of the changes in order to provide a snapshot of the market at a specific moment in time. These photographs of the market are called Order books.

A book is composed of two side, ask and bid, moreover it is specific to an instrument or share, so one will be generated for each stock present in the market or for each one we intend to follow. The two sides keep track

respectively of the sell and buy orders which are aggregated based on the price, effectively creating bins or level. The number of level available is usually referred as market depth and it is in general limited when tracking a lot of instruments.

In Figure 2.1 for example, we have a market depth of five and the levels are dictated by the price. For the bid side the orders are placed in an increasing manner since the lowest price is the most desirable. In contrast for the ask side the order is decreasing as highest price determines bigger profits. Oddly enough this Order book has been designed to be read bottom up.
This book mechanism is the tool used to take trading decision as it allows to have an idea of demand and offer in real time. For such reason it is beneficial to always have the most recent version of it, especially when faced against other HFT strategies even a couple of nanoseconds can make the difference.

2.1.1 Market Feeds

Although the definition of book is unique there exists two type of market feed:

- By **level**, the messages it generates are bid or ask limits. They are the cumulated sum of all orders available at a specific price for a given stock. Limits are calculated by the exchange. An example of this kind of market is CME[1]
- By **order**, the messages it generates are commands to buy or sell shares at a specific price. These commands are created by traders.

The same two categories are reflected in the Feed Handler, two solution have been developed to handle the different feed types. An example are the US Equities, NASDAQ, BATS and NYSE.

Feed Handlers by level are in general faster to develop because the stock exchange publishes limits instead of orders. Usually, the level of the limit is given (Eurex, Xetra, CME, etc.). The order book's update is then only an add limit (if the limit does not exists), delete limit or modify limit. Whereas, for Feed Handlers by order, the stock exchange publishes orders. This fact complicates a lot the generation of the book, not only each order needs to be stored to account for changes or deletions, but the stock needs to be extracted and the limit of its book recomputed.

For the scope of the internship the focus will only be on markets by order

[1] Chicago Mercantile Exchange

but all the work performed could in theory be translated to the other type of markets.

2.2 Feed Handler by Order Architecture

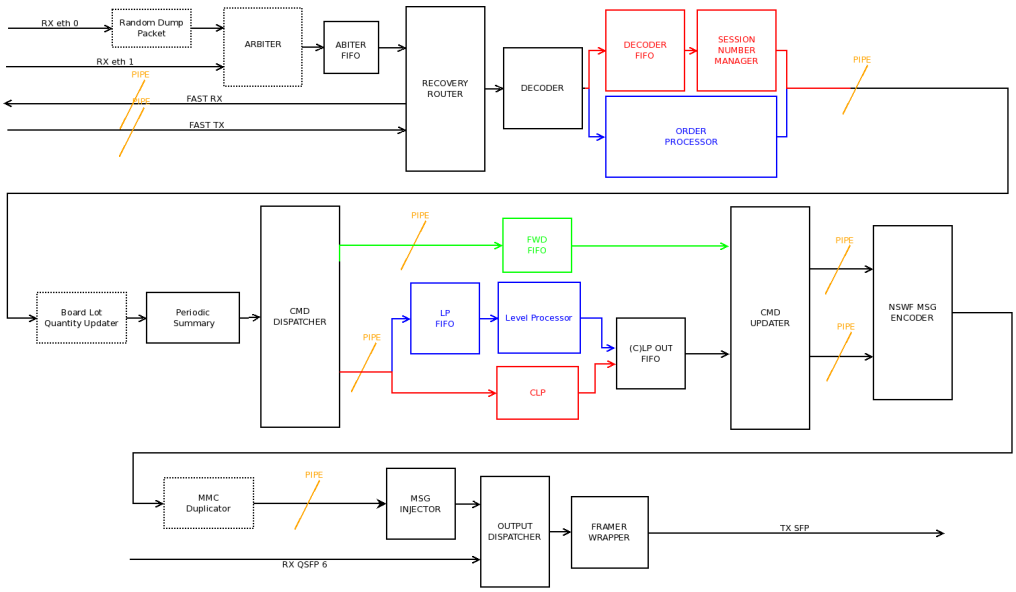

Figure 2.2. Current architecture of the feed handler.

The Figure 2.2 shows the current architecture of the Feed Handler by orders, which is the focus of this work. It is structured as a long chain of components with little to no branching inside of it. Even if each component has its own functionality we can group them together to separate the FH into three section:

- **Input** section, goes from the beginning up to the decoder. It receives the IP/UDP packets, perform arbitration among the multiple inputs, decode the messages and normalize them into commands understandable by the feed handler.

- **Processing** section, goes from the Order Processor to the Command Updater. It processes the commands to create the order and level book.

- **Output** section, goes from the NSWF[2] Message Encoder to the output. It transforms the internal commands and the books in a standardized format called NSWF, encapsulate them in IP/UDP packets and output them.

Out of the three sections there will be extra focus on the first one as it is impacted more by the addition of another market. Nonetheless, if needed, a more in-depth analysis will be performed for components of the remaining sections.

2.3 Input Section

All components in the first part of the chain will be briefly described to understand their role and their current limitations.

Ethernet and arbiter

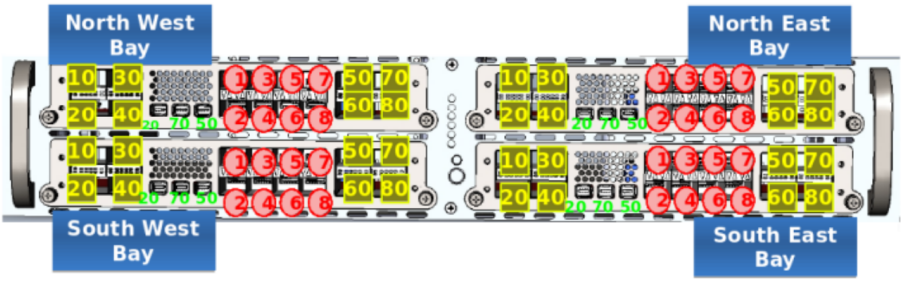

Figure 2.3. Front of an appliance with space for four FPGA.

The appliance[3] is composed by eight port which can be configured independently to be either input or output.The arbiter can support a maximum of two input flux and its role is to provides recovery functionalities through redundancy. If enabled in advanced mode it will allow to detect missing packets in one stream and try to obtain them from the other stream. This kind of functionality reduce the necessity of retransmission and thus lower the possibility of latency penalty associated with it. In particular it can be very useful when paired with a mixed input composed by fiber optic

[2]NovasparkS Wire Format

[3]This term refers to the full product, motherboard and FPGA, integrated in a server rack

and microwave since The latter provides better latencies but it is less reliable. A secondary feature but still important consists in a first coarse filtering, based on the destination IP, to avoid overwhelming the following components.

Recovery Router

The recovery router purpose is to maintain a communication channel between the FPGA and the computer. It forewords all incoming packets through PCIe in such a way that the software is able to detect possible gaps into the data sent by the markets. When a gap is detected it will halt the flux of market messages until the software has recovered the correct message. This mechanism is in place to avoid breaking causality, meaning that we should always consider orders in the correct sequence to avoid scenarios like deleting an order prior to its submission. Currently the appliance is limited to one PCIe lane even if more are present on the board, thus limiting the possibility of communication.

Market decoder

This component is at the interface with the processing segment and its task is to convert the messages into an internal format understandable by the rest of the chain. It is divided into two part. The Market parser extracts all needed fields present in a message and assign an opcode to it in order to identify the type. The Normalizer arranges all the useful fields based on the opcode and through the aid of some hashtable it is able to correctly identify the instrument, if present, in order to perform a mapping between its name and two unique identifier, the instrument number and a prehash. Those, associated with the exchange id, assures that no collision is present when storing, processing an order and updating the ask or bid books.

The parser is fully auto-generated while the normalizer only in part[4] but in any case this ease the process of adoption of new markets and migration of older one. The automation process ensure a certain level of optimization, in particular for the encoding of the commands generated by the normalizer. But it also means that each market will have commands of different length, with the fields in slightly different positions.

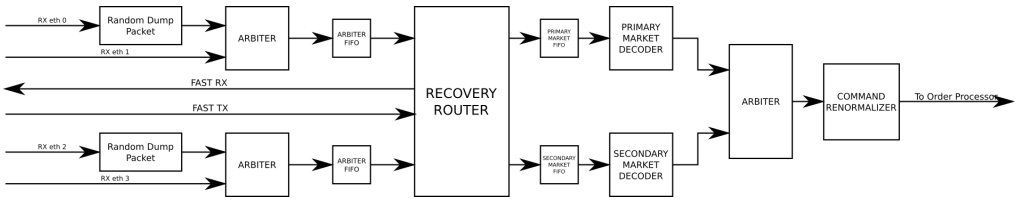

Figure 2.4. Modified architecture of the feed handler.

2.4 Proposed Architecture

The new architecture consists in duplicating the first part of the feed handler and recombining the two flows into a single stream. This approach is rather simple and naive but it carries some major advantages.

- There are only a few component which need to be developed from zero and thus there is more time left to testing and validation.
- Reusing most of the already existing components make the project easier to maintain and prevents it from diverging from other products.
- The parallel structure is easy to integrate in term of component instantiation and does not require a new dedicated wrapper.

Considering also the recent migration towards a larger FPGA, the Virtex-XCVU13P Ultarscale+, this solution becomes pretty viable. Thanks to the new free space, there is the possibility to accommodate the extra components that will be added.

2.4.1 Renormalizer

It is used to translate the different formats coming out of the two normalizer into a common format which will be used in the remaining part of the chain. This function can be realized by simply wiring bit x to bit y and apart form some cases in which a MUX will be used it will generate no logic. For this reason it will be fully combinational.

2.4.2 Command Arbiter

Its purpose is to merge the streams of incoming commands into one. Like a set of semaphore in a crossing it will prevent crushes by allowing only one lane at a time to go foreword.

[4]The prehash for example is market specific and optimized for use in the Order Processor

Since the feed handler is not a perfect pipeline[5] and data activity will change over time, this component is crucial. It represents a weak point as its forewarning method could lead to data loss inside the chain. For this reason multiple criteria were tested and validated in order to avoid the above mentioned scenario.

[5]Look at B for its communication protocol

Chapter 3

Development

The work was carried out using the vhdl language for the RTL description and python for verification and testing purposes. The focus will be the coexistence of two markets, NASDAQ and bats, on the same feed handler. However for generality purposes a python extension called Jinja has been used to grantee the possibility of changing the target markets with ease. For a better description of this templating language I remind to Appendix C.

3.1 Packages

Before approaching the development of any new component it is necessary to define a common reference. In case of the FH we need a VHDL package with the structure of the commands described in it. Such package will be referenced throughout all the chain and it is always specific to the market. Each command is identified by an opcode and associated with it there are a series of fields. All those information, along with size, encoding[1] and names, are stored inside a ***.json*** file. An example is sown in Appendix D where a delete order can be found.

The normal procedure to obtain this file and the package usually goes along those lines

- Reading the market data documentation and extrapolating all the relevant information
- Creating a ***.json*** with all the extracted information translated in the commands used internally
- Feeding the *market_commands.json* to a proprietary script

[1] Unsigned or Signed field

the result is the desired package with the minimum command length obtained through the overlapping of non concurring fields.

This flow is not applicable for the project due to the lack of a single documentation. Using one of the already existing *.json* is no solution either as a consequence of the difference of the markets. To accommodate for this diversity the new *nasdaq_bats_commands.json* was generated as the union of the two *nasdaq_commands.json* and *bats_commands.json*. It is a non optimal solution especially when dealing with the fields, some of them will be only used with bats and some others only with NASDAQ. This means that the final size will surely increase with respect to the one of the origin markets. As example here are the dimensions for the single and combined[2] version

	NASDAQ	BATS	NASDAQ + BATS
Bit Width	440	540	644

Having now the stepping stone the development can proceed with the first component, the Renormalizer.

3.2 Renormalizer

As stated before the already existing normalizer will be reused and thus a translation is necessary to pass from the standard their family use to the one defined above. According to the input market a different mapping will be needed and this can be solved in vhdl with the approach below

```
1  -- MARKET_NAME : string, declared inside generic
2  architecture rtl of renormalizer is
3  -- Signals declaration
4  begin
5      if MARKET_NAME = "bats" generate
6          process(opcode, command)
7          begin
8              case opcode is
9              -- Slice mapping
10             end case;
11         end process;
12     end generate;
13
14     if MARKET_NAME = "nasdaq" generate
```

[2]Note that command size change frequently, over the course of six month it went from 670 to 652, 664 finally landing to 644

```
15        -- Slice mapping
16        end generate;
17    end architecture;
```

The problem with this method, apart from being error prone, is the fact that markets evolve and so do their commands. This file would need to be constantly updated and to avoid it a Jinja template[3] has been used.

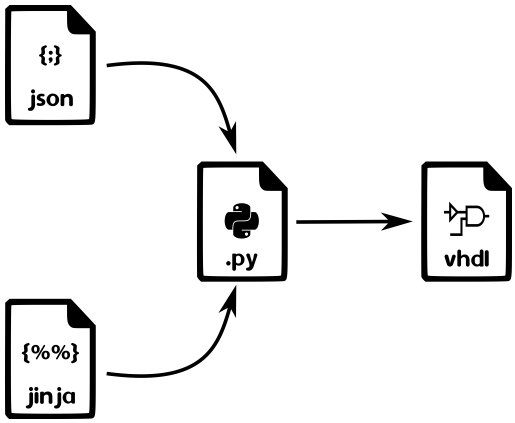

Figure 3.1. Diagram for Jinja template generation

In the diagram we can see how with the aid of python it is possible, given a series of family, to retrieve all the needed information and generate the desired mappings inside the renormalizer. Concerning the translation it mainly consists in moving slices of bits from a certain offset to anther.

There is only one exception, the *ns_order_id* field. This element, correspond to the prehash mentioned in Section2.3 and covers an important role in the Order Processor. It is used as key for the hash functions of the order hashtable and normally it is unique. For this reason it is stored together with the other information to correctly identify and order in case of hash collision. Due to the structure of the prehash the unicity cannot be assured between multiple markets and thus the renormalizer will assign a number to each family present and concatenate such number to the *ns_order_id*, forming a new field called *unique_ns_order_id*. The latter will take the role of unique identifier for an order solving all collision problems but at the same time it also degrades the total capacity of the hashtable.

The DQR memory used in the Order Processor has a data size of 144-bit and can be configured to be written and read in chunks of 36-bit since

[3]Refer to Appendix C for more details about Jinja

unique_ns_order_id needs at least 37-bit its size will immediately saturate to 72-bit in the QDR, reducing the memory space that can be used to store new orders.

3.2.1 Validation

The component has been validated with the following methodology[4]:

- The target family is specified and fed to a python script
- The python file recovers the informations for the origin families and randomly generate a series of N command, then dumps them to a *input.txt* file
- The N command are translated to the common standard and written to a *ref.txt* file
- A testbench loads the input from the first file and checks if the generated output is in line with the reference created by the software. It also saves the output of the renormalizer in case a check is needed if the simulation fails

3.3 Command Arbiter

The component's role is to merge the streams coming from the decoder stage avoiding any possible collision. It will thus be inevitable to stall some commands for one or more cycle. This fact provides not only a random increase in the overall latency but also a possible point of failure. The ability to buffer the incoming data is limited and we risk to lose some of it if a branch of the input section is halted for a relatively long time. To avoid the scenario two operating mode have been devised and thoroughly tested. Depending from the scenario the use of one of the other could provide some advantage but ultimately both of them were developed with different goals in mind. Before diving into them I remind that the possibility to have different configurable mode is allowed by the Avalon interface programmed into the FPGA, for further reference on this topic refer to A.

3.3.1 Latency

This operating mode is the simpler and its objective is to maintain timing as small as possible. It takes advantage of an internal timestamps, used to measure the point to point latency, and prioritize the least recent command.

[4]This will roughly be followed to test all future components

Such behaviour is actually not applied in all collision, the reason being that commands are grouped into units called packets and separated by an EOP[5]. It is important keep this logical units separated and once a side has been chosen it will keep consuming the following commands until we find an EOP. The figures display the state diagram of the FSM used to implement

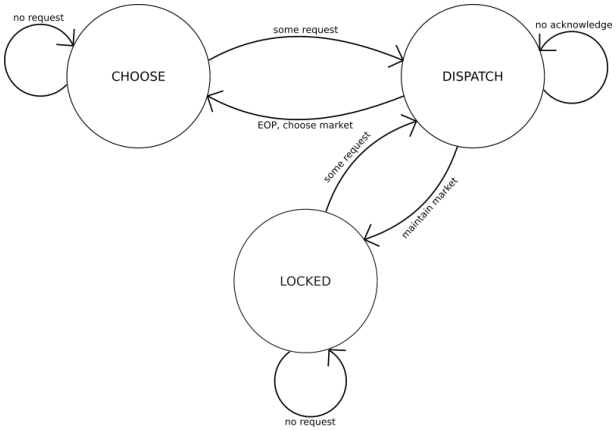

Figure 3.2. Finite State Machines for the Latency mode

this mode. Here is a brief explanation of each state

- **CHOOSE**, decided which command is acknowledged. In case of collision compares the latency field and takes the command with the higher one. If the latency is the same priority is given to the first market, NASDAQ in our case. Once the choice has been made, it is stored stored into a register and the same is done for the command
- **LOCKED** keeps consuming, according to the choice register, as long as the command in the pipeline is ready to be consumed by the following component
- **DISPATCH** oversees the transaction with the Order Processor assuring that no new command is accepted until the one stored has moved on

Overall the latency for this mode is two clock cycle. Despite the fact that the **DISPATCH** state actually wastes one cycle, stalling the input, this solution remain pretty optimal. From the simulators it emerged that at least one clock cycle is necessary between two following command, so in case of immediate reaction at the output everything remains well coordinated.

[5]End of Packet

3.3.2 Round Robin

This mode, as the name says, employs a round robin algorithm in case of multiple command ready at the same time. It keeps consuming commands until an EOP is present at all inputs. At this point only one of the End of Packet is forwarded while the others are discarded. The objective of this behaviour is to merges packets from the two markets to reduce the waiting time for both data streams and also the amount of commands sent to the next stage.

The main player used for this component is the logic involved into the decision process. The starting point is a fixed priority encoder. It is defined by this 3.1 kind of truth table

req_0	req_1	req_2	$\cdots$	req_n	$garnt_0$	$garnt_1$	$garnt_2$	$\cdots$	$grant_n$
1	X	X	$\cdots$	X	1	0	0	$\cdots$	0
0	1	X	$\cdots$	X	0	1	0	$\cdots$	0
0	0	1	$\cdots$	X	0	0	1	$\cdots$	0
0	0	0	$\cdots$	1	0	0	0	$\cdots$	1

Table 3.1. Truth table for fizzed priority logic.

and can be can be summarized with the following equation

$$grant_n = \overline{\left(\sum_{i=1}^{n-1} req_i\right)} \cdot req_n$$

To make the priority variable it is sufficient to replicate the above logic and feed it with a rotated version of the input. An AND gate is the added to each fixed priority at the output and each of the is ored together. In this way there is the possibility to choose which of the rotation is being used. To conclude a ring counter is used as select signal for the multiplexer, so at each clock cycle the higher priority input is shifted by one. This component will be referred from now on as root arbiter. On its own this component would be sufficient for two markets but to accommodate for the possibility of joining four or more[6] a general solution has been developed. For it a variation of the base arbiter it is needed. It will be called base arbiter and the only addition is an output consisting of the OR between all the input, to detect if the component is active or not.

In Figure 3.3 we can see how it is used to generate a tree structure for the final arbiter. The root arbiter are used to generate the grant output in the

[6]This would be feasible only by reducing the number of instruments per market, to avoid excessive ram usage/

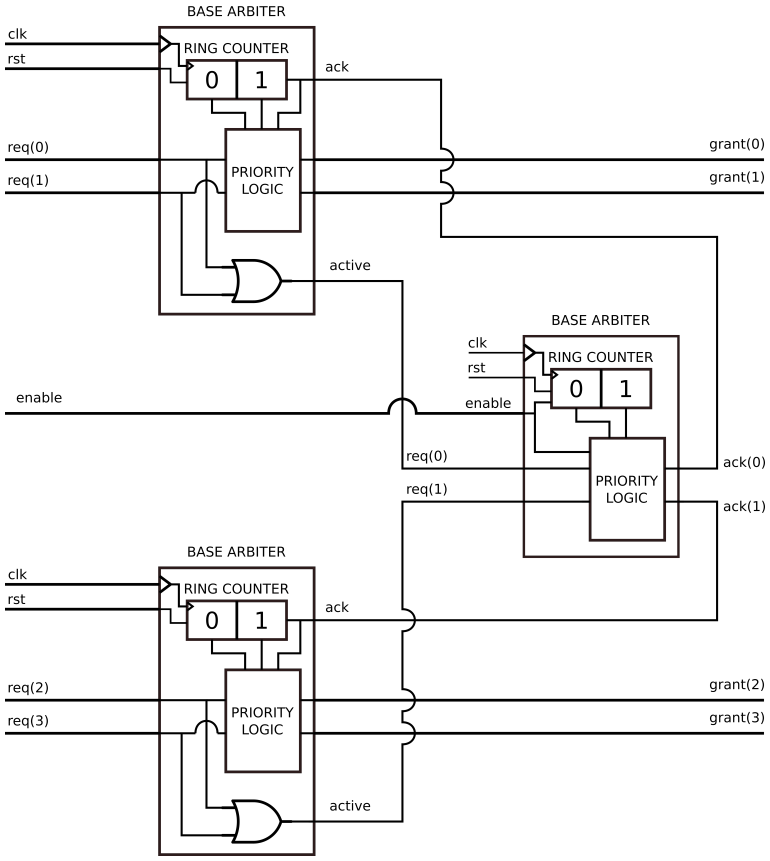

Figure 3.3. Example of a four input arbiter.

first layer while in the middle one they use the active signal as input and their output are fed back as enable signal to the previous layer. The last layer is made of a root arbiter since it is not necessary the active signal. The result is the desired Round Robin behaviour. The output is delivered as one hot encoded signal but to be used as select signal of multiplexer it is also converted into an unsized representation. The logic for the decoding is quite simple and by observing the 4-bit case in 3.3.2

req_0	req_1	req_2	req_3	sel_0	sel_1
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	1	0
0	1	0	0	0	1
1	0	0	0	0	0

Table 3.2. Part of truth table for one hot encoding to two's complement.

we can derive some generic rule

$$sel_n = \sum_{i \in A} req_i \text{ and } A = \{x \mid x \in [0, N-1] \wedge x \bmod 2^{n+1} \geq 2^n\}$$

where N is the length of the input signal and $n \in [0, \lceil \log_2 N \rceil - 1]$.

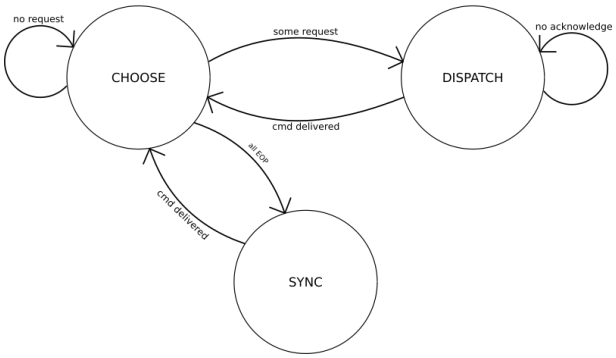

Figure 3.4. Finite State Machines for the Latency mode

The Round Robin logic needs to be integrated with some registers and control logic to properly foreword commands. On the first iteration a final state machine was used. Its structure was pretty smiler to the one used in the Latency mode, its state diagram is visible in 3.4, but instead of locking into a stream the choice was active each time. Moreover one state was dedicated at the synchronization. This solution, although working as intended, immediately showed its limitation when the component was integrated in the feed handler and tested with some real market data. We cannot expect a continuous and regular flux of packets at all time, there will be some moment of inactivity and precisely during those moment the FSM would remain stuck on the synchronization stage waiting for a packet to arrive in one of the two input. As a result incoming packets were lost.

The second iteration, available in Figure 3.5, replaced the FSM with just control logic. In particular, instead of pretending that all markets are active and for each of them a packet needs to arrive, it will register which one are active and base the synchronization only on them. This kind of behaviour is mainly due to the accumulation of incoming requests with the following relationship

$$req\_acc_{i+1} = valid_i \sum req\_acc_i$$

These registers will then be erased once the synchronization is reached and the accumulation process can restart. The final result has a much more flexible management of the inputs and it now allows to handle the case in which only one market is programmed into the feed handler.

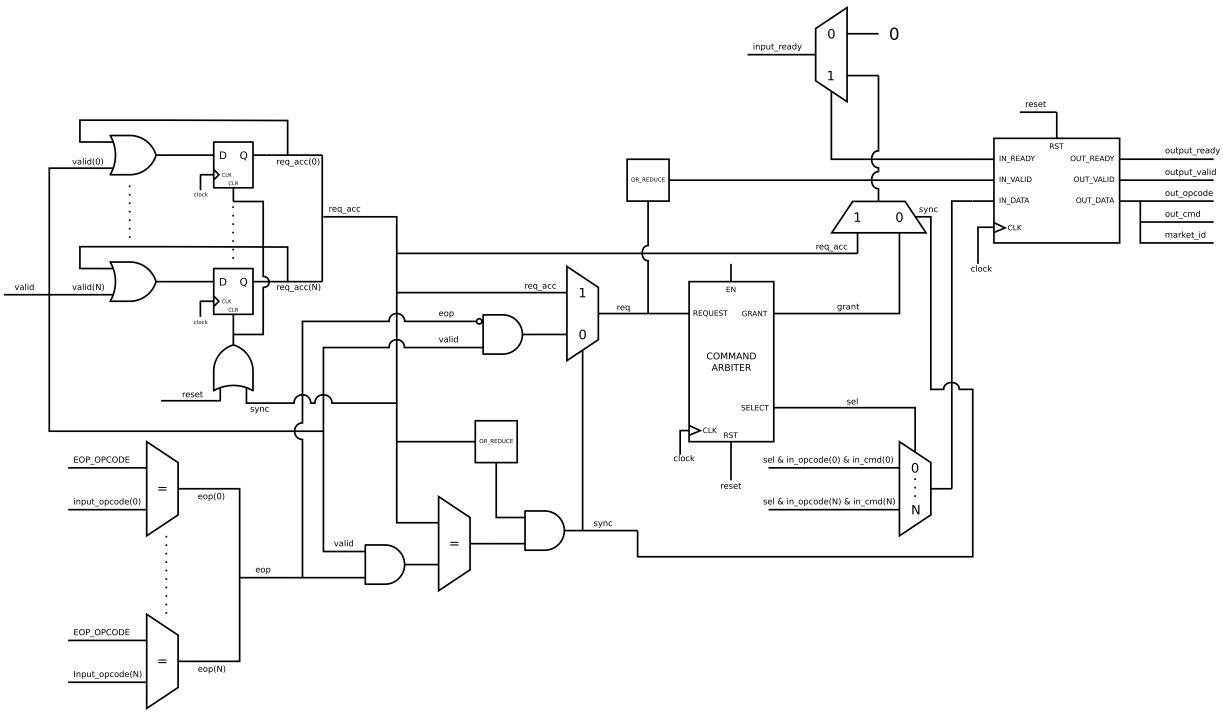

Figure 3.5. Block diagram of the Round Robin arbiter.

3.3.3 Validation

The correctness of the behaviour for the arbiter has been carried out in a similar manner to the renormalizer thus I remind to Section 3.2.1.

3.4 Market Decoder

The conceptual idea is very simple, as shown in the image all there is to do is just to instantiate the necessary component and test with some inputs that the decoder is producing the desired outputs. Unfortunately, since the existing code was not conceived with this future prospective in mind a series of technical challenges arose. The first one being entity conflict.

3.4.1 Name Conflict

The problem lies in the fact that for each family the decoder always uses the same name/identifier for the entity and for the architecture. Since Modelsim, but also Vivado, allows only the existence of one entity under the same name, if we try to include two or more it would just overwrite the already compiled *market_decoder* each time a new one is encountered.

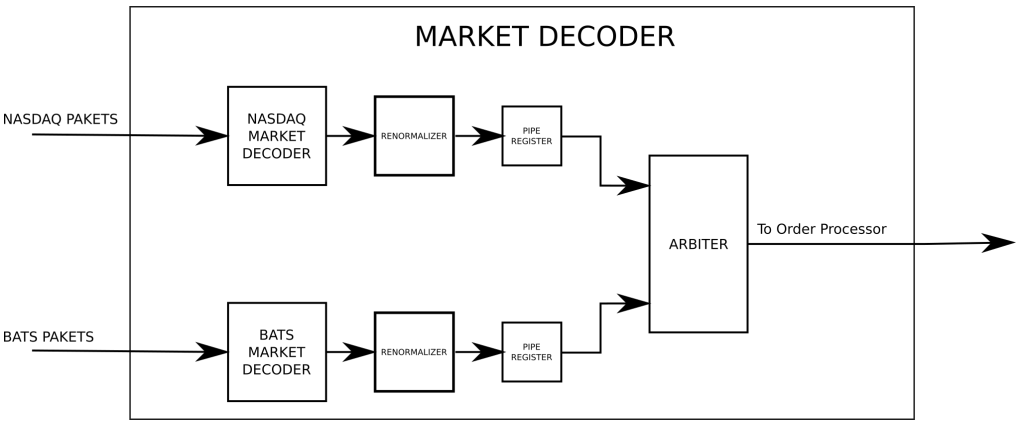

Figure 3.6. Block diagram of the market decoder.

In this way we would never be able to instantiate different one within the same design.

One possible solution is to just rename the architecture of each decoder. Since the interface is standard across all of them this is sufficient to reach the desired result, however it will require a big code refactoring and it would also break the current structure of the feed handler and some other automation tools which are centred around the *market_decoder* name. For such reasons the proposed and adopted solution was to utilize libraries. By compiling the necessary components into different libraries it is possible to use them even if they share the same name. The only caveat is that we cannot directly refer to them but we have to specify the their full path, *library_name.entity_name*, in order not to cause name conflicts.

3.4.2 Libraries Integration

In order to add such support, the way in which the compilation is handled was partially modified. The two main player in this step are:

- **info.txt** files, written in json format, contain a list of sources, components as well as packages which are used inside a specific *.vhdl* file. They are also used to record information on the available testbenches and to pass generics and commands to the simulation or synthesis environment.
- **simulate.py** script, which uses the above files in a recursive manner, to resolve dependencies across all the components used by the selected entity. Moreover it also parse all command directed to the simulation and correctly set up the software.

A neat little feature, already available into the info files, is the possibility to specify some attributes to sources or components by grouping them together in a list. It is possible to take advantage of this feature and expand it, intro ducting a new attribute.

```
1  {
2      "src":
3      {
4          "MyComponent",
5          "MyPakage",
6          ["MyFile.vhd", "example_lib"]
7      }
8  }
```

As shown in the example above it is now sufficient to use a string with the suffix *_lib* to specify the name of the library, while when not specified the *work* library will be the default one.

The main change to achieve this result was the passage from a list base storage to keep the sources to a dictionary based one. In this way each list of files is directly associate with library which is used as key of the dictionary. As for the algorithm used to detect dependencies it just needs to be directed to the right library before looking for the desired sources. The last adjustment was the addition of the same algorithm used to solve dependency but targeted to the libraries instead. This addition was necessary to determine the correct compile order.

3.4.3 Backpressure

Another problem which stood out after the first simulation was that regular decoder do not support backpressure. Unlike all other components inside the FH they need to receive a ready high signal before asserting their valid signal. This kid of behaviour renders impossible any kind of arbitration since we have no way of knowing when a command is available unless we already commit to accept that command. To correct this, a set of register has been applied to the output of each market decoder in order to buffer the commands and give the possibility at the arbiter to correctly exert backpressure. This solution is unfortunate since we are forced to add one cycle to the whole chain and the goal is to keep the latency low, but since modifying the decoder is currently not an option, this remains the best solution.

3.4.4 Validation

The testing of the *market_decoder* aims to reuse as much as possible the pieces of software used in the other families. Generating inputs and reference for each market separately is thus easy but since the output is generated from two streams we would need timing information on when the commands arrive at the arbiter in order to create a timing model. This task can although be avoided by sorting the output again into separate streams for each market and checking that each message arrives in the right order. For this reason a signal has been added to the arbiter[7] and also to the decoder in order to uniquely identify the origin market of a command, but it will only be used for simulation purposes.
Before actually arriving at a testing stage some other issue needed to be resolved. The normal validation procedure for the market decoders is to first define a *simulation functions* package. It contains the procedures necessary to write the registers[8] configuration. Since some procedures already existed for each market, a wrapper was created. This although generated also conflict, due to the library solution. It was discovered that if a custom type is compiled, even from the same source code, into two different libraries it is not recognised as the same type and thus it is impossible to make any assignment without defining a casting function. In this case the procedures used to configure each decoder were compiled in their own library and thus making it impossible to make them all work together with a single signal type. To solve this issue it was decided to refactor all simulation oriented packages into a library on their own. As second problem, in order for the configuration to currently work it is necessary that each instance of the same entity is assigned at compile time a different ID[9] to correctly address them. This prompted to some code refactoring and addition of generics which have now been integrated into the main developer branch.

3.5 Feed Handler core

The structure of the fh_core is rather simple in terms of logic and mainly consists of component instantiation. For this reason the bulk of the work consists into introducing *for generate* statements to replicate components and expanding the width of some signals. A constant was introduced to keep track of the markets used in a family. This new constant is called **NUMBER_OF_MARKETS** and used to correctly parametrize the

[7] The signal *market_id* is visible in 3.5

[8] The same one which will be software configurable, refer to A

[9] The identifier is used by the Avalon interface to detect a component.

structure of the fh_core according to the used family.

3.5.1 Problems encountered

During the integration process two major problem were encountered. The first one regards the market decoder, there is mismatch between the input port structure. Normally ready and valid signals are of type *std_logic* but for the *nasdaq_bats* family they are of type *std_logic_vector* since it has to receive at the same time data from multiple markets. Normally this would not be a big deal, nothing that a simple *if generate* statement cannot solve. However the structure of the fh_core has been consolidated over the years and there are multiple case in which components are addressed in simulation and synthesis using the absolute path from the top entity. This means it would be required to track and modify every occurrence regarding the market decoder. To avoid this error prone kind of work, the standard interface of the markets decoder were vectorized. This is indeed much simpler as a single modification is applied in the common template and refreshing the decoders solve the problem for all markets.

The second problem regards the input output interface of the core. In order to communicate with the software a connection is created through PCIe, and such link takes the name of fast path. There are only a limited number of PCIe lane available, and not all of them are used, but unfortunately the PCIe controller used inside the fh_core does not support more than one lane. After discussing the matter with the appropriate team it was decided to leave the problem in standby since adding support for recovery of lost packet is a secondary feature. For the moment the goal is to test that the chain is able to support the amount of data coming from two markets.
In any case the possibility are two:

- Maintain a single PCIe lane and split data internally.
- Add a second lane, and later expand it to three or more according to the number of markets.

3.5.2 Configuration

The validation procedure consists into feeding to the fh_core real captures of market messages registered using Wireshark. Then checking the messages at the output against a reference generated by a piece of software, developed internally, called Comparator.
Since the comparator is market specific it can process only one type of data for each source file. Due to this limitation the basic idea is the same used to test the market decoder. Start from two different captures, interleave the packets and then split the output messages based on the market of

origin. Unlike before we cannot add a signal to carry the information of the parent market. It would be very difficult to integrate this functionality into every component of the chain. The solution lies directly on the features embedded in the core. With the right configuration we can decide for each instrument the destination IP and port. This will allow us to split the output capture in two and use the respective Comparator to check the correctness.

Involved Components

All the following components are based on the same principle. In order to allow a configuration of part of the fh_core we use ram in order to store informations which can be used to generate different behaviours. So the configuration depends in general from the content of the memories and what is used to access them.

Instrument Hashtable

The first elements to configure are the instrument hashtables[10]. They are used to specify which instruments we are following and which we are ignoring[11]. They contain normally the id used by the market to identify the instrument, the name of the instrument, as well as the id which will be used inside the FH core. This translation is necessary to avoid collision in the following stages, especially when we are trying to track the orders to generate the books.
The hashtables are initialized through some *.mem* files. Luckily it is possible to directly use already existent *.mem* file coming from other tests. The only change we need to apply regards the last 4-byte in order to insure that the any instrument uses the same internal identifier.

In stream arbiter

The second component is the packet filter inside the in stream arbiter. As its name suggest it is used to filter incoming IP/UDP packets based on the destination multicast address. Each component offers a set of six pair of registers, used to define six different ranges of IP address. The registers are only 23 bit long but this is sufficient based on the market that the feed handler supports. In the Tables 3.3 and 3.4 there is an example for the configuration of both markets. Looking at table defined for NASDAQ it is evident that there are just three IP address, 233.54.12.101, 233.54.12.40 and 233.54.12.101, while for BATS there are proper ranges since it uses

[10] They are mapped as ram in the design like all the rest of configurable part

[11] All instrument mapped to a zero cell in the ram are ignored.

Register	Value
address-0-low	$0x360c65$
address-0-high	$0x360c65$
address-1-low	$0x360c28$
address-1-high	$0x360c28$
address-2-low	$0x360c2d$
address-2-high	$0x360c2d$
address-3-low	$0x000000$
address-3-high	$0x000000$
address-4-low	$0x000000$
address-4-high	$0x000000$
address-5-low	$0x000000$
address-5-high	$0x000000$

Table 3.3. NASDAQ IP range table

Register	Value
address-0-low	$0x003e02$
address-0-high	$0x003e0c$
address-1-low	$0x003e1e$
address-1-high	$0x003e20$
address-2-low	$0x003ec0$
address-2-high	$0x003ece$
address-3-low	$0x008240$
address-3-high	$0x008247$
address-4-low	$0x008200$
address-4-high	$0x008207$
address-5-low	$0x000000$
address-5-high	$0x000000$

Table 3.4. BATS IP range table

way more than six addresses, thirty one in my case. Note that not all IP considered in a range are actually useful, an example is range zero for BATS. It includes from xxx.0.62.2 to xxx.0.62.12 but in reality the important addresses are only half, the one terminating with an even number (xxx.0.62.2, xxx.0.62.4, xxx.0.62.6 etc.).

NSWF encoder

This component is close to the output of the chain and performs a translation of the internal instrument id, defined in the hashtable, into a series of information. Here is the mapping of the data inside the decoder:

4 bit bitmap	4 bit exchange_id	4 bit channel_id	24 bit out_id

- **bitmap**, used to specify to which output interface the packet will be sent. Each bit correspond to one Ethernet port.
- **exchange_id**, identifier assigned to the exchange[12].
- **channel_id**, used in the framer to decide the destination IP and port of the packet.
- **out_id**, identifier used at the output for the instrument corresponding at this address.

[12]Each market is in general composed by sub-markets called exchange

Framer Wrapper

This component uses the **channel_id** specified above to retrieve the all information necessary to a IP/UDP packet. It is possible to have a maximum of 16 channels and the information stored into the session memory are:

16 bit checksum	16 bit ip port	32 bit ip address	42 bit mac address

3.5.3 Validation

Once all configuration files were generated the design was tested at first using captures containing 5000 packets coming from a single markets and the output was validated with the comparator. This step was necessary to identify possible bugs related to only one side of the chain. Then a joint capture was created through the use of Wireshark. The first step is to modify the initial captures and set the timestamps in such a way that the time frame is the same, then the following command generates the final capture

```
mergcap inputs.pcap -w outputs.pcap -F pcap
```

With a proper input the fh_core was tested for both mode of the command arbiter. To separate the output capture Wireshark is used again with the following command

```
tshark -r input.pcap -w output.pcap -F pcap -R "ip.dst==192.168.1.10" -2
```

and at last the resulting capture can be fed to the Comparator.

Chapter 4

Synthesis

4.1 FPGA

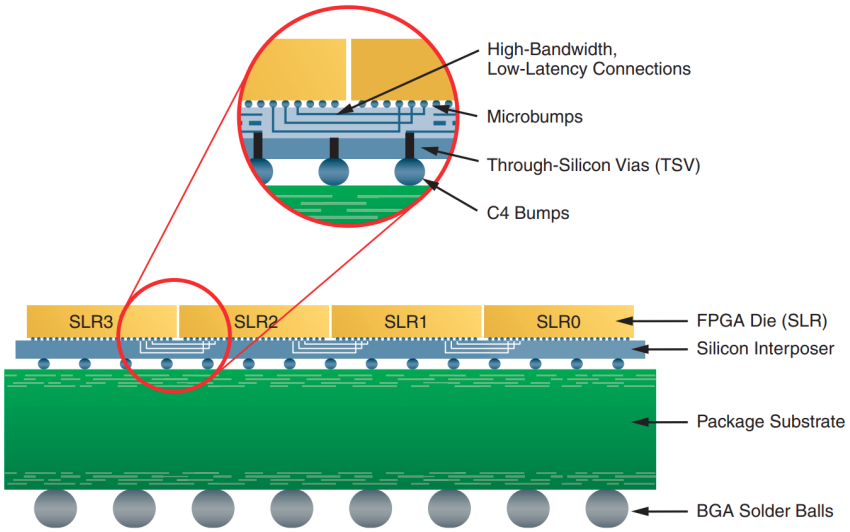

Figure 4.1. Diagram for an SSI structured FPGA

As mentioned in the introduction the FPGA used for the implementation is part of the Virtex Ultrascale+ from Xilinx. It is one of the top product available in this market and for good reasons. Starting by its size[1] this FPGA is huge, to put things in prospective there is around an order of magnitude in difference between the logic element present in the Virtex

[1]Packaging size is 52×52 mm, close and even bigger then some modern CPU

compared to more normal FPGAs, like the Zynq 7000 or the Cyclone V series (millions against hundred of thousands at best). But the differences does not stop here, either if we are talking about memory (BRAM, DDR4, QDR), DSP or IO this FPGA will have plenty to spare. It even offers a category of its own the UltraRAM, which are quite large, fast and reconfigurable memories directly integrated into the die, to be specific dual-ported 288Kbit SRAM.

In order to fit all this feature a multi-die approached is used. As shown in the figure each die corresponds to a SLR (Super Logic Region) which is connected to the adjacent die through a SLL (Super Long Line). These connections resides on the Silicon Interposer, a layer used in-between the IO component of the package and the dies. This kind of structure is called SSI (Stacked Silicon Interconnect) and it strongly influence the design methodology for this kind of FPGA, since it allows to cram more silicon together but at the same time it can hinder performance and thus where and what is passing through this communication layer needs to be carefully planned.

4.2 Block Placement

To correctly implement your design in this kind of device it is necessary to roughly assign a zone of the die to the major building blocks of our project. This process constitute a coarse floorplan and it is needed since the software, Vivado, used to synthesized has some difficulty in exploring the design space, due exactly to the dimension of the FPGA. It does a much better job when we restrict the area and position of the design's component.

4.2.1 Floorplanning basic

As starting point an identifier is assigned to specific region of the FPGA through this command

```
create_pblock pblock_name X4Y3:X7Y11
```

The arguments correspond to the name and the identifier of a clock region. Such regions are distinct part of the FPGA characterised by a fixed number of resources. They also correspond to the basic building block that you can select[2]. By using the two clock region joined with a colon, a rectangular

[2]It is also possible to select directly the resources, LUT, BRAM, DSP, etc. in order to finely define your selected area

space is generated with those two regions at opposite corners. Once the necessary pblock are crated a reference to the components in the design can be obtained with this kind of command

```
get_cells -hierarchical -filter {NAME =~ "regexp_expression" ||
                                 REF_NAME =~ "regexp_expression"}
```

One can use either the **NAME** or **REF_NAME** the difference is simply that the latter is unique. Moreover in this case due to the use of libraries, a *_libName* suffix is added to the **NAME** in case of the same entity being present in different libraries. The bizarre fact is that when a library is not explicitly defined for a component it gains *_default* ending, meaning that contrary to Modelsim the default library is not called work. The last step is then to assign all instances of the component to the pblock with the following command

```
add_cells_to_pblock pblock_slr1_left_bottom  [get_cells $inst$]
```

where the variable ***$inst*** contains the path to a specific component instance in the design.

4.2.2 Floorplan evolution

This initial placement is stored into a ***.xdc*** file and the one already available for the normal FH was used as starting point. Figure 4.2 shows where the component are usually placed.

Farts Iteration

This configuration allows to reach a slack of roughly -2.213 ns. But even to do so some initial constraint needed to be relaxed, in particular the areas assigned to the market decoder and NSWF encoder were not sufficient and thus they were expanded.

From the figure it is obvious that the FPGA is not fully occupied and a lot of space is still available. SLR0 is not used at all and also part of SLR3 remains unused, while SLR1 and SLR2 result very congested and the routing proves to be the major player in the timing violation. Considering the worst path from the instrument hashtable to the arbiter, it shows that of the total delay 0.355 ns is due to the actual logic and 4.821 ns to the routing, which corresponds at 93% of the whole delay.

Second Iteration

After trying to adjust this configuration with little improvement it was obvious that some change was needed. The first step was to move the

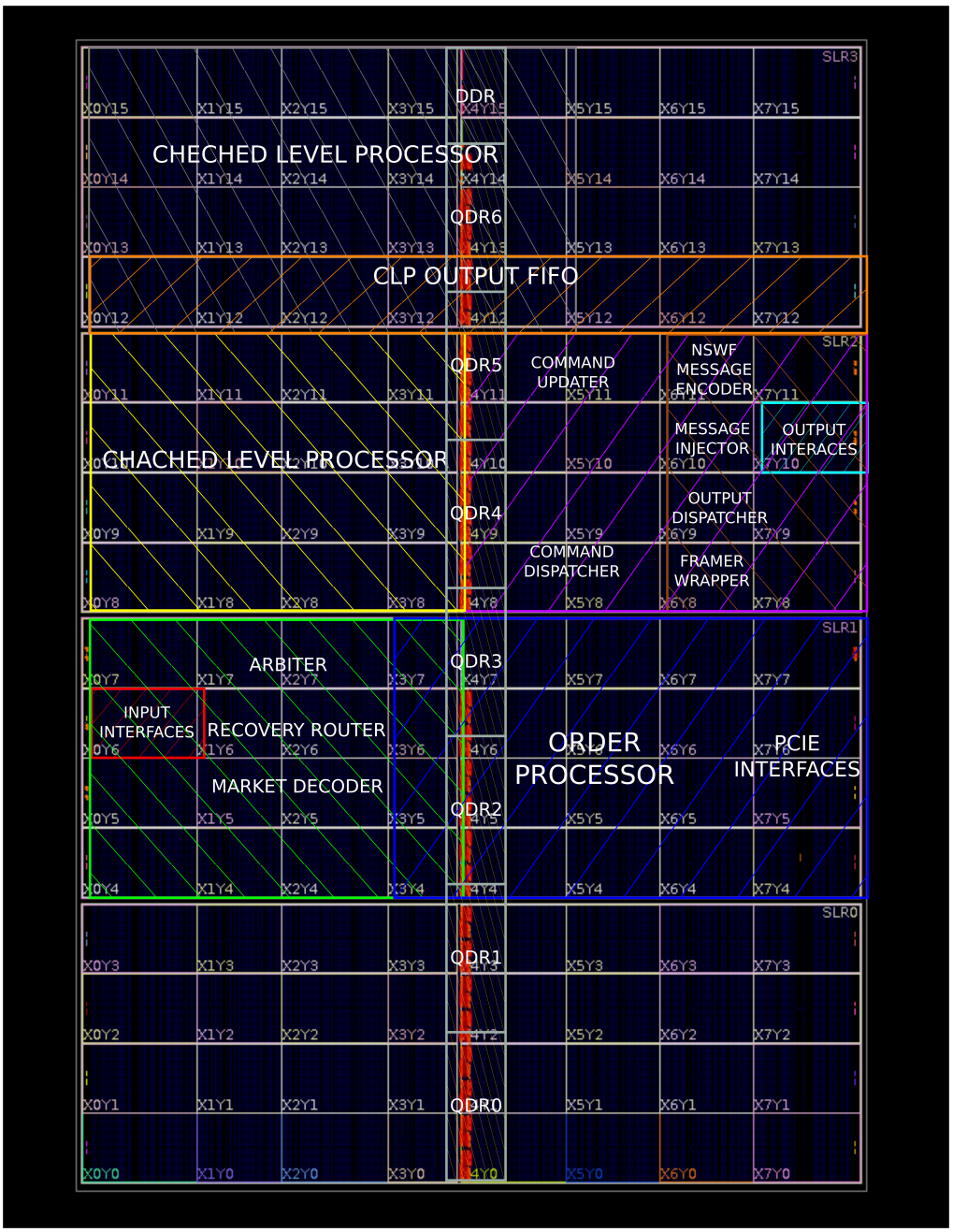

Figure 4.2. Schema of default placement for FH core

market decoders into the SLR0. This move was done not only to relieve congestion on SLR1 but also with a regard for the future. In case a new market is added to the project there would be no space for it in the SLR1 while there is plenty in the SLR0.

This solution demanded an addition of two pipeline register, between the market decoder and the in stream arbiter, in order to use some specialized registers, called Laguna, to efficiently cross SLR. The results of this change were not immediately evident, the slack got worse, $-2.510\,\text{ns}$, but now the timing inside the decoders were much better and close to zero.

Third Iteration

The problem was now with the component after the decoder, the Order Processor. Due to their connection the solution was to move also this component to the SLR0 and an additional division was made. The market decoders would take the left half while the order processor the right one. This displacement also demanded a change in the QDR used, the third was deactivated and replaced by the zero in order to keep the memory as close as possible to the order processor. This time the improvement were quite significant with a slack of $-1.598\,\text{ns}$.

Forth Iteration

Despite the improvement the major problem, the NSWF encoder, still remained. This component is at its core a big RAM with also a bit of logic in it. Since its memory cell are scattered in the right part of SLR2 and the area is fairly congested, the interconnection delay results too high. To solve the problem as many components as possible were moved into the newly available space in the right half of the SLR2. Again a good result ($-0.844\,\text{ns}$) but nowhere near to an acceptable one.

Fifth Iteration

After some more experimentation the area assigned to the NSWF encoder was reduced a bit and shifted away from the output interfaces. Moreover a new area was created in the middle of the right half of SLR2. It was then assigned to the message injector[3] for two reason:

- Having a more chain like structure in the floorplan, instead of using a single area for more components
- Correcting a bad tendency shown across all synthesis iteration, which was placing the message injector in the clock region X5Y8, which is in the bottom-left corner of SLR2 right half

[3]This component visible in Figure 2.2 and 4.2 collects various periodic messages from the software, like packet statistics or status of the board, and foreword them to the output

This solved almost every problem in that zone and It allowed to reach a −0.124 ns slack. The tool was now consistent and always hitting around the −100 ps mark. The best so far reached the −60 ps. This is how the best run looks like: Although not positive it is possible to state with a certain degree

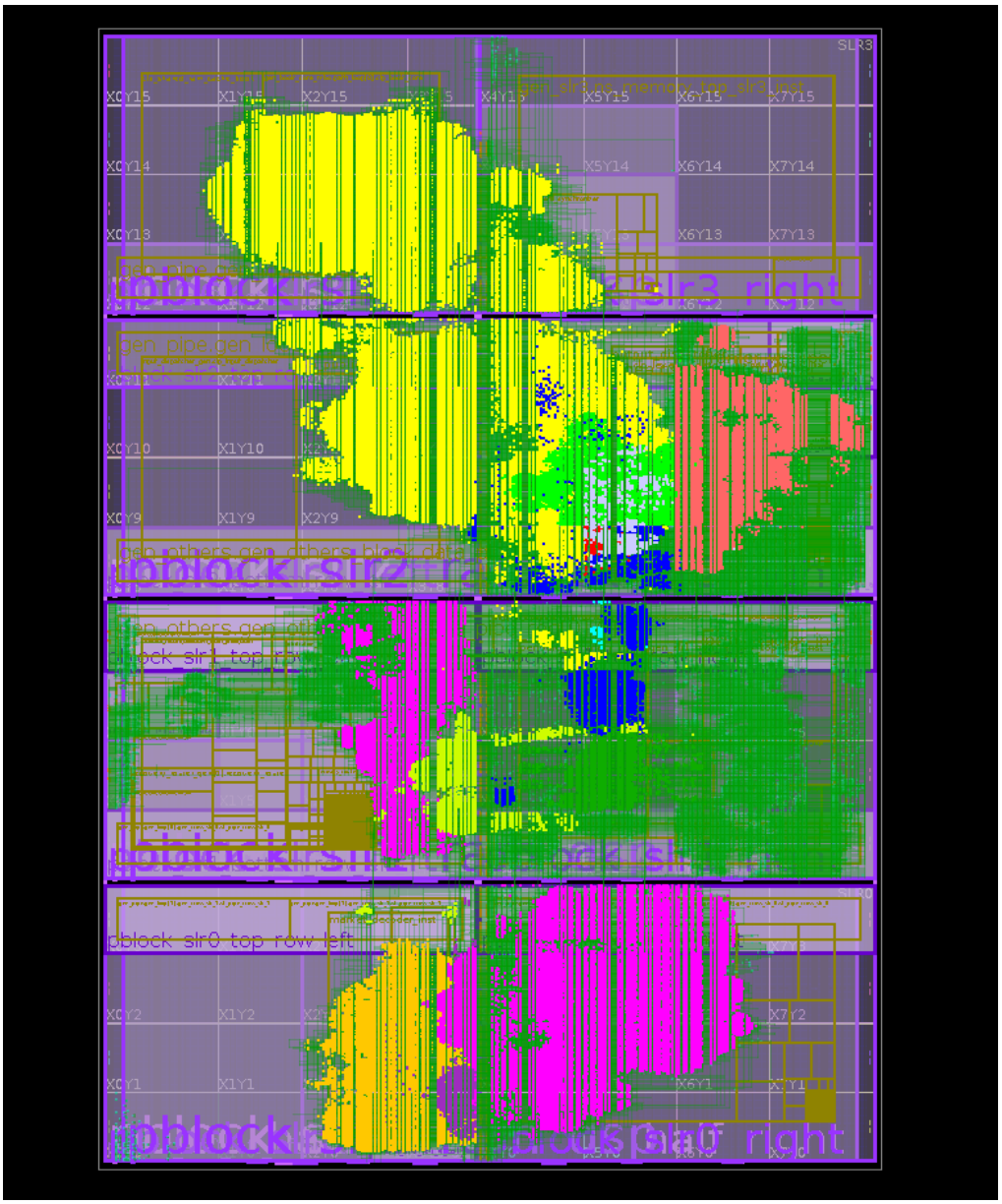

Figure 4.3. Image of the final synthesis

of confidence that anything below the −100 ps threshold will work without a problem. This is mainly due to the pessimistic estimate performed by the software and related to temperature, clock uncertainty etc.

Chapter 5

Software

The software is currently not ready to support two market together and due to the mole of code involved I am currently unable to propose an effective solution[1]. Nonetheless after some reverse engineering and debugging it has been modified to crudely configure in the correct way the appliance.

The main limitation derives by the fact that no configuration model currently exists with the presence of multiple markets and thus the only way is to trick the software into thinking that is dealing with a single one. This cause the information to mix together and renders sometimes impossible to associate them to the correct market, making necessary the use of ad hoc coding just to re-establish the boundaries between the markets.

5.1 Global structure

A file called *application_description.json*, which contains all of the components configurable by the software, is parsed and for each element its possible settings are recorded. All the settings are merged together with the *patches.json*, corresponding to common properties and market specific variations, to create a file called *apg.json*. Such file represent a template used to generate actual *settings.json*, which are records of a configuration. The apg file is then compiled together with the C++ sources to create the *nsapg* binary which takes three file as input

- **feeds.csv**, contains the exchanges involved and the IP/PORT coordinates.

[1]Due to delivery time of other projects no one in the software department could develop the code alongside my work.

- **instrument_dictionary.csv**, contains a list of the instruments we want to follow, as well as to which exchange they belong to and what output id they will have.
- **settings.ini**, list of value used to configure the component of the FH,

The *nsapg* compiles the information in those three files into a *settings.json*, then goes over all the cores and configure them through the Avalon interface.

5.2 Software changes

The only areas that needed to be touched were the one used to configure the hashtables inside the normalizer. The reason being that the *exchange_id* is involved in both the hash generation to recognize an instrument and the key generation which will be used in the order hashtable. Currently such field uses 4-bit inside the FH but depending from the market a slice of 2 or 3 bit is used for the two processes mentioned in the previous statement.

This behaviour derives from the general amount of exchanges being below four and also from avoiding the number zero as exchange id. Taken into account these details it is obvious that at most three of the four bit actually carry important information. Since the *nasdaq_bats* family will have at most seven exchanges all the four bits become important and some care needs to be taken in how the exchange ids are assigned.

The exchange id is supposed to be a unique field and the same applies to the slices used in the normalizer, at least inside the same market. To respect these rules, the solution is to force the assignment of this field to NASDAQ and then to BATS. In this way they will have range 1-3 and range 4-7. The next step is to count how many of the NASDAQ exchanges are present and subtract that number to wherever exchange id is used for BATS exchanges. In such way the mapping used in the hashtable will consider the last three bit and derive the usual range of 1-4.

The last remaining step is to hard code a filtering process before the configuration of both the hashtable and the in stream arbiter. For the first the aim is to avoid passing instruments which do not belong to the renormalizer of a specific market and for the second instead is to correctly compute the IP ranges based on the exchanges of a market.

This solution is very simple but has also big problems. The software is able to correctly configure the FPGA only in two cases

- Only one market is present
- Both market are present, but NASDAQ has all the exchanges

The remaining case, both market present but with only some exchanges, risks to not work correctly as there is the possibility that orders from the different markets could be confused and mixed inside the order processor.

Chapter 6

Testing on FPGA

With both software and firmware ready the work done so far can be validated on an actual board.

6.1 ATST

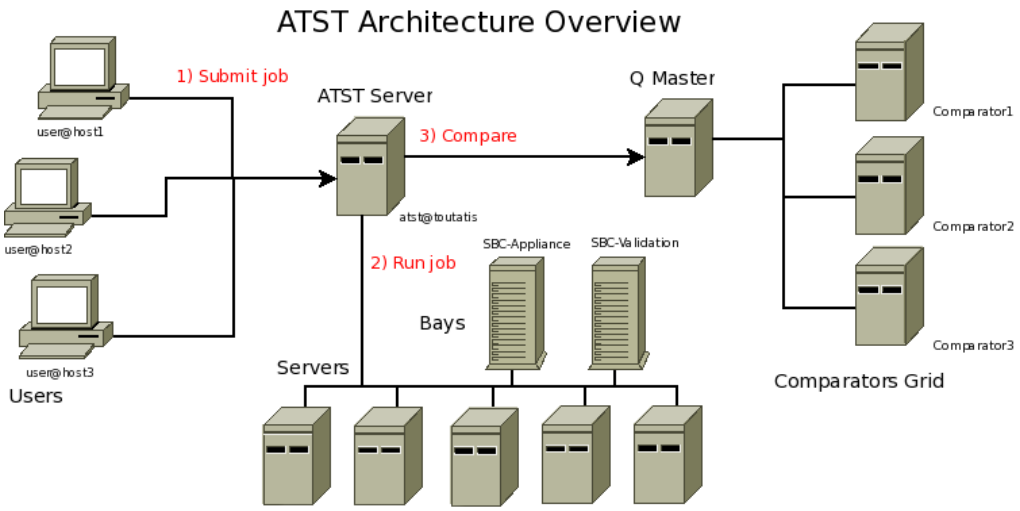

Figure 6.1. Example architecture of the ATST platform

All test conducted on real hardware pass through the ATST platform. It is an automated server which lets users submit jobs easily, without worrying about competing for resources or inappropriate hardware configurations. It is a very useful platform and this are its main features:

- automated job queueing system

- management of hardware resources
- allocation of bays[1] to a job based on its requirements
- allocation of CPU resources for running comparators
- email notification upon test completion

6.2 Testing Methodology

The methodology remains very similar to Section 3.5.3, but with two main differences. The main one is the size of captures that the appliance allows to run. To be more precise this difference is dictated by time. It is possible to use big captures containing million of packets in Modelsim but it is not practical. For a better prospective here a simple comparison. The test used in the above mentioned section, takes around 20-30 minutes with 10000 packets, while the average time for a 10 million capture is between 8-11 minutes on the FPGA. This diversity motivates the use for real hardware as an in depth validation tool, while software simulation is better for debugging, corner cases exploration and small regression testing.

The other change concerns the channel id. While validating the correct functioning of both markets some problems emerged with three class of messages:

- **Version** messages, used to broadcast the version for the output protocol, i.e. the proprietary NSWF
- **Gaps** messages, used to signal the presence of a gap at the input. Meaning that the books published after such messages may be incoherent with the real state of the market
- **Exchange** messages, used for delivering some information regarding a single exchange, like temporary interruption of the treading services

The common factor among these three types is that they are mapped to the zero address of the instrument ram used in then NSWF encoder. By default this address contains the value 0xF00000000, referring to Section 3.5.2 the extracted information result in

$$\text{bitmap} = \text{F}, \quad \text{exchange\_id} = 0, \quad \text{channel\_id} = 0, \quad \text{out\_id} = 0$$

Among the four identifiers *out_id*, *bitmap* and *exchange_id* do not pose any problem. The latter is not even used, a separate memory stores the values for the translation. The *channel_id* value instead correspond to a special mapping and forces the MMC Duplicator to broadcast the message

[1] This terms refers to an appliance

on all the configured multicast group. This means that on the IP address reserved for NASDAQ there will be some messages belonging to BATS and vice versa. The comparator is unfortunately unable to cope with those messages and generates an error, making the validation of the output capture impossible.

The simple solution would be filtering the unwanted messages but only in case of small captures, around the 1G mark or 10 million packets. When dealing with more consistent tests, in the order of 40 to 70 gigabytes which represents half or a full day, this approach is really time consuming and from my experience not fully stable[2] with the current tools.

The best solution would be changing the behaviour of the MMC Duplicator in order to tie an exchange with the channels actually used by it. A proposal regarding this topic already exists and it implies the passage from a number, to encode the *channel_id*, to a mask representation. It means not only a rise in the field size, from 4 to 16 bits, but also a revision of the whole output section. Since the development for this idea requires a considerable amount of time[3] and this feature is not currently in demand by any customer, the use of the *channel_id* was dropped entirely.

The appliance has at its disposal four different outputs and it gives the possibility to associate an instrument to an ethernet port through the use of the *bitmap*. Although the problem for the three category of messages remains, due to the default value 0xF for instrument zero, it is easier to fix. Moreover with the used of multiple output the sorting of the output capture based on the IP address is no longer required.

6.2.1 Exchange Bitmap

A comparator is used to detect when a message requests the zero element of the instrument ram. In such case the internal *exchange_id* is used to access a small ram which contains the real bitmap. The substitution between the default and real value happens through the use of a multiplexer. To regulate this behaviour on a user level a flag, called **exchange_msg_sorting_enable** has been added to the Avalon registers of the NSWF decoder. Note that depending on the type of market the internal *exchange_id* may or may not assume the value zero, for this reason when not used it is set to 0xFFFFFFFFFF as in Figure 6.2. That value should aid with the debug of the output capture in case address assumes the unexpected zero value.

[2]When trying to edit large capture crashes are quite frequent

[3]The estimate made by the manager is around three months

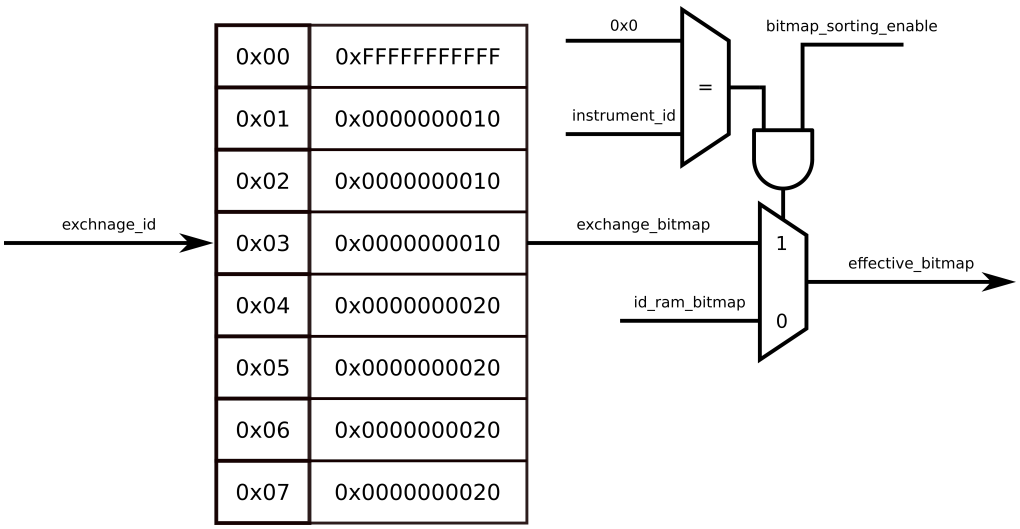

Figure 6.2. Exchange bitmap block diagram

In the image below is shown an example of how the feature is enabled together with other parameters.

```
[processing]
book_shift_enable = false
trade_id_enable = true
delta_prices_enable = true
publication_depth = 5
decross_enable = false
exchange_msg_sorting_enable = true
```

The configuration of the small ram uses the already existent address and data interfaces dedicated for the instrument ram but an enable signal needs to be added to regulate the access specifically for the exchange bitmap ram. The value stored inside the ram are determined at boot time. The bitmap field is stored in an optional row inside the *instrument_dictionary.csv* file, of which an example is available here.

```
market_instrument_id,  exchange_name,  out_instrument_id, bitmap_enable
A,                     inet,           1,                  0x0000000010
BAK,                   bx,             664,                0x0000000010
CDI,                   psx,            1226,               0x0000000010
GURI,                  byx,            3289,               0x0000000020
MLAB,                  bzx,            4763,               0x0000000020
REX,                   edga,           6176,               0x0000000020
WGA,                   edgx,           7896,               0x0000000020
```

By oring each bitmap belonging to the same exchange the result is a cumulative bitmap of all the outputs used by the exchange and it can be stored inside the newly added ram. This process takes advantage of the configuration code for the NSWF encoder, it exploits the iteration over all the instruments to compute the real bitmap and thus it does not have a significant impact on the boot up time.

6.3 Effective instruments

Before discussing the results it is necessary a small digression on the hashtable used in the Order Processor. The order hashtable is split in smaller memories (QDRs) and it employs six different hash functions in parallel on the same key. This kind of redundancy assures in general the capability to find a free cell inside the table. However depending on the type of keys and the filling of the hashtable it is possible that no place is available. In these cases an external memory can be used but not only it degrades performance when looking for an instrument, it also carry the risk of becoming full and thus it is better to avoid its use.

This is exactly the case for the BATS market. From the tests emerged that during high activity period some order were lost. The cause can be traced to the use of the *uniqie_ns_order_id* coupled with the *ns_order_id*. This latter field is composed in different manners depending on the market but it is always used as key for the hash functions. Its value is very important and directly connected to the distribution of the orders inside the hashtable. Normally it is constructed in such a way to be unique but in some cases this condition may be impossible to respect with only 36-bit so the *uniqie_ns_order_id* comes into play to fill this role with its 72-bit. This field is than stored inside the hashtable instead of the *ns_order_id* to correctly identify an order. The doubling in dimension degrades the total capacity of the hashtable and thus reduce the maximum number of orders that can be stored inside of it.

This downside become even more accentuated when both markets are active at the same time but since the study of new key values optimized for this structure is outside the scope of the internship the only available solution is to restraint the maximum number of instruments. In this way number of live orders is reduced and the table never incurs in any space problem. In particular NASDAQ is reduced from its theoretical limits of 27000 to 13600 and BATS, which is heavily effected by this issue, goes from 36000 to 13000. In general the conclusion is that 26000 can be considered the current limit when mixing the two markets, independently on the proportion.

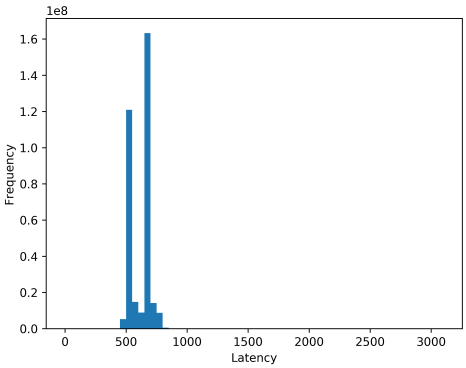

Figure 6.3. Timing NASDAQ only configuration

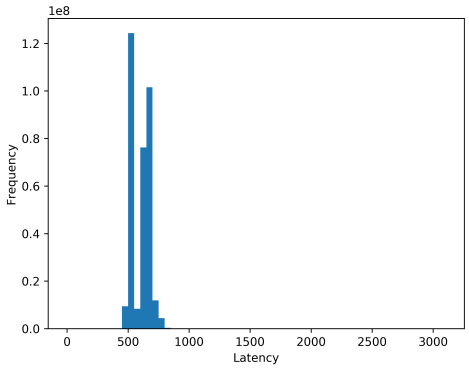

Figure 6.4. Reference timing for NASDAQ test

6.4 Results

The performance metric used to evaluate the NASDAQ/BATS feed handler will be latency. As stated before the objective is to maintain it as low as possible. Considering the changes made to the architecture a minimum shift of six cycles can be expected. Two cycles due to the command arbiter, another two to cross SLR and last two pipeline stages at the output of the normalizers to allow backpressure. This corresponds to roughly 18 ns which is nearly the case for an only NASDAQ configuration.

In Figure 6.3 and 6.4 are available the histograms for a full day capture. Around 330 million packets are registered at the output and for each of them the latency is computed with the aid of two synchronized counters, one at the input and one at the output. Although the possible granularity of the data is really fine the tails results invisible a cause of the main spikes around the 600 ns mark. Nonetheless it is still evident that a shift has occurred. Comparing the averages reveals a difference close to 17 ns, from 594.64 ns to 611.62 ns.

Conducting the same comparison for a BATS only configuration results in a similar outcome but it yields a greater shift in the overall distribution. The average moved from 611.775 ns to 635.64 ns, a total displacement of roughly 24 ns.

Passing now to the captures of both market together, Figure 6.7 shows the latency histogram for a joint configuration of 13K instruments per market, with again roughly 330 million packets. The performance in this case is exceptionally good with an average of 610.303 ns. This value is better, even if by a small margin, than the NASDAQ only test and clearly it is this market which dominates the distribution. Even changing the selected

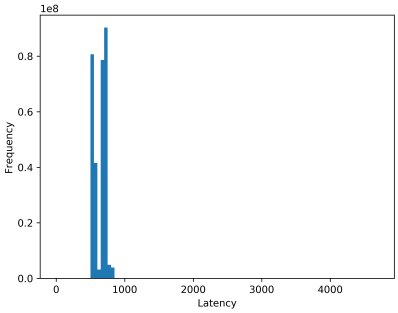

Figure 6.5. Timing BATS only configuration

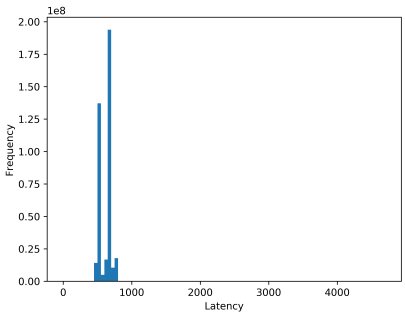

Figure 6.6. Reference timing for BATS test

instruments does not change much meaning that this result is mostly linked to the ratio of instruments between the markets.

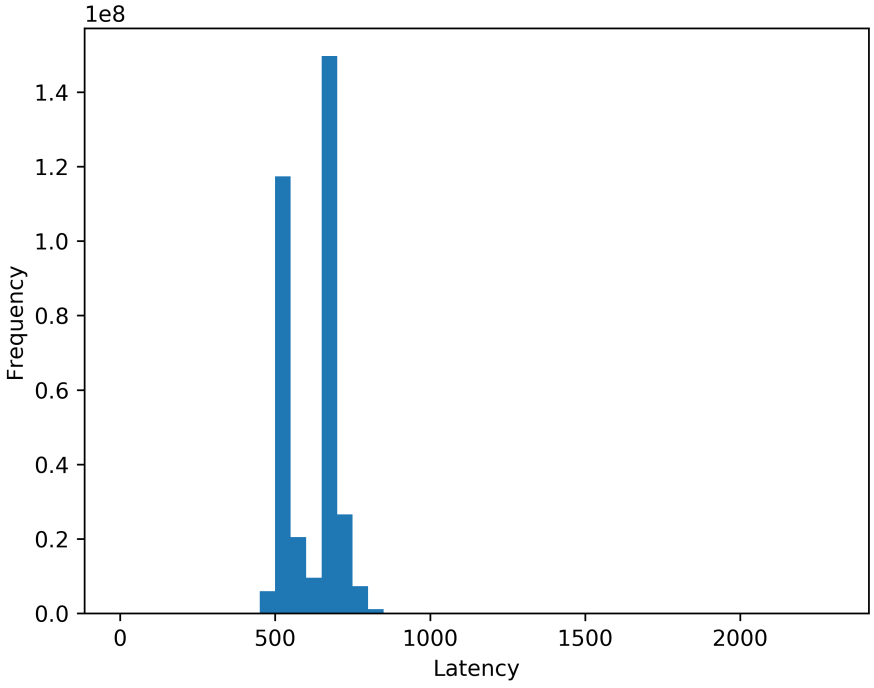

Figure 6.7. Timing for NASDAQ+BATS configuration

Regarding the uneven delay between the two markets it still has no clear origin. Since stressing the architecture revealed the complete absence of

collision on the command market.

Another small remark regards the worst possible timing. This information seems consistent and remains nearly unchanged for both sides, it usually accumulate only three or four nanoseconds. But when looking closely at the tails it is possible to see that they are actually less populated when compared to the reference distributions. This is overall a good factor but it must not be mistaken as an improve in performance since it is probably due to the additional pipes which gives a little more time to the chain during processing. To put it simply the extra delay makes the chain less clogged than usually is. But nonetheless it remains overall slower.

6.5 Future Improvement

As stated before the current prototype has may issue, but of course it is the first of its kind, and thus it means that there is still a lot of room for improvement. Here is a rundown of all the features which are missing or could get improved:

- support for data recovery in case of gaps, two possible routes have already been proposed in 6.2
- rework on the hash key or in alternative addition of unused QDR3[4] to the Order Processor to reach support of the maximum number of instruments on all markets
- proper generalization of the software and specific template definition for the settings of the multimarket family
- definition of standard validation flow which is automated and integrated with the ATST platform

Looking now at future prospects the most important aspect will be to explore different families combination and test how scalable this solution is. By looking at Figure 4.3 it is evident that there is definitely enough space left in the FPGA to fully fit a third market. One of the major limiting factor will be for sure the order hashtable. The more market are joined together the more difficult it will be to reach an even distribution on the hash functions, moreover if the peak activity grows the space will eventually reach critical capacity and orders will be lost. For these reasons for the four markets mark and onwards, the number of instruments will be limited and careful analysis should be performed to decide how the partition for each market influence the overall behaviour.

[4]Preliminary tests show that this is possible with non negligible performance hit of around 8 clock cycles

6.6 Conclusion

The purpose of this work was to cover the development and decision process adopted in the making of a new category of products for the company Novasparks. At the same time it shows how in the context of a consolidated architecture it is sometimes necessary to make some compromises and trade-off possible performance for development time.

Overall the goal for the internship was met. The Feed Handler is working as intended and it is true that there are stills some problems in the architecture, like the hash distribution and the recovery portion, but this first NASDAQ/BATS hybrid can serve as proof of concept as it shows the feasibility of an FPGA Feed Handler linked to more than one market.

This work will be a valuable asset for the company. In the recent years more small markets have appeared, like LTSE (Long Term Stock Exchange) and MEMX, and more are likely appear in the future. With the possibility to aggregate them together in a single FPGA Nova sparks will be able to offer a compelling product for trading company eager to engage in these new treading territory.

Appendices

Appendix A

Avalon

The Avalon interface is used to create an addressable space usable by the software. In this way we provide a mean to set some configurable parameters, once the FPGA has already been programmed, and to report possible internal error. In particular this interface is used during boot of the appliance to correctly setup the RAM and user defined parameter.

The section of the standard used is the Avalon-MM[1] and in general this are the signal it uses:

- ***amm_clk***: clock signal, it can be on another frequency domain with respect to the rest of the design. In such a case the incoming data needs to be resynchronized.
- ***amm_reset***: reset signal, restore the component to a user defined state. It is synchronous.
- ***address***: on the master side the default behaviour is to allow byte addressing while on the slave side we have a word addressing by default. In both cases the addressing unit can be configured, in general the word mode is used.
- ***byteenable***: each bit of this signal state which byte in the data is eligible to be written or read.
- ***read***: used by the master to initiate a read communication.
- ***readdata***: used by the slave to acknowledge a read transaction with the master.
- ***write***: used by the master to initiate a write transaction.
- ***writedata***: used by the slave to acknowledge a write transaction with the master.

[1] Memory Mapped

- ***waitrequest***: asserted when the lave is not capable to immediately answer to the master. It is a way to stall communications until the slave is ready.

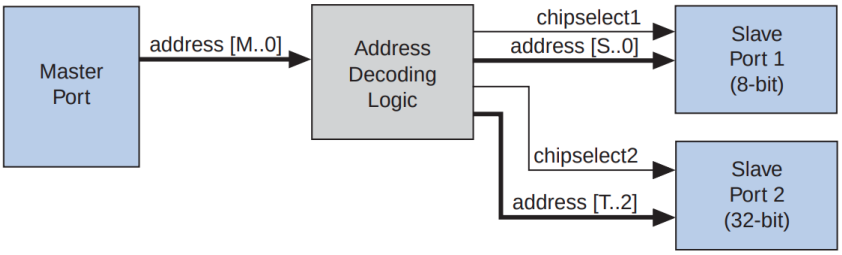

Figure A.1. Simple AMM tree like structure.

The AMM uses an hierarchical structure where one or more master initiates transaction with slave nodes. As shown in Figureẽffig:ammtree there could be different slaves and each of them can have its own addressing space. The address size of the master does not limit the one of the slaves and it is even possible for them to have a bigger address of the master but in this case a suitable decoding logic needs to be developed. Regarding the addressing capabilities the master is bound to some conventions. A byte alignment is forced in relation to the data width. Example, if the *readdata* port is 32-bit than addresses for a contiguous read must be spaced of 0x4.

For reading and writing operation many mode are available and I remand to the official documentation for some detailed information on all the specification and operating mode.

Appendix B

Component Communication

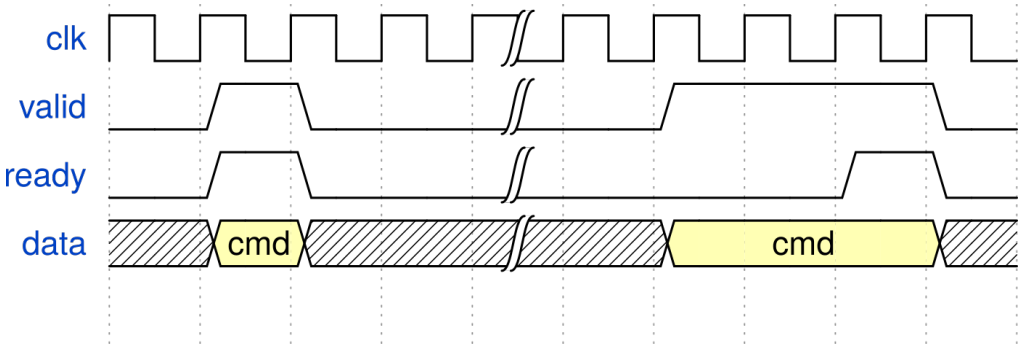

Figure B.1. Example of transaction with the *ready/valid* protocol.

Although all functional components work at the same frequency inside the FH, they do not create a perfect pipeline, meaning that some may have a variable latency. For this reason the communication among them is coordinated through a *ready/valid* handshake. This protocol is very starightforeward and simply data on the output is carrying meaningfull information when the *valid* siganl is asserted, while data is considered consumed when both signals are asserted at the rising edge of the clock. A sample waveform is shown on the Figure B.1. One advantage of this protocol is the possibility to exhert back pressure. This term refer to the fact that a component is able to regulate the incoming flux of data through the use of the *ready* signal. Whenever it is deasserted new data could be accumulate at the iterface or rirectly being halte, this behaviour can be seen as pressure acumulating at the input of the component. This property, for example, is crucial for the arbiter to function but it represents also a weak point. In case the incoming data cannot be saved or put on wait it ends up as discarded.

Appendix C

Jinja2

Jinja is a powerful templating language which allows to organize any data coming from a python script into text format. Its syntax[1] is encapsulated inside what we could call container in order to differntiate it from normal text which is part of a template. There are four different type of container:

- `{% ... %}`, is used for statements like loops and branches. It contains in general the logic of the template
- `{{ ... }}`, is used to evaluate expressions and print the result inside the template
- `{# ... #}`, is used for comments so the text inside of it will not be included in the template output
- `# ... ##`, is used for line stetements. They are a configurable and alternative way of marking some text as a jinja specific command

The philosophy of Jinja consists in using the language to place the data at the right place in a file and relegate the processing to the python source. Despite this the language does not enforce this behaviour and result very flexible providing also a lot of tools when some logic is needed. To the point that it can be as expressive as python itself.

As example the source file of the renormalizer is provided in the following pages. The code employees a wide varieties of the offered features, like variable assignment at line 47, loops, conditional statements, text formatting and also custom functions. At line 77, **log**, **ceil** and **int** are explicitly imported from python and the symbol | act as a pipe and feed the outcome of a function as input to the next one.

[1] For further referece refer to the project manual [1]

```
----------------------------------------------------------------
-- NovaSparks Confidential
--
-- (c) Copyright NovaSparks. 2010, 2019
-- The source code for this program is not published or otherwise
-- divested of its trade secrets.
----------------------------------------------------------------
-- This module is generated by script gen_renormalizer.py
----------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.commands_pkg.all;

entity renormalizer is
    generic
    (
        FAMILY_NAME : string;
        IN_OP_WIDTH : positive := 7;
        IN_CMD_WIDTH : positive := 1
    );
    port
    (
        in_ready      : out std_logic;
        in_valid      : in  std_logic;
        in_cmd_valid  : in  std_logic;
        in_opcode     : in  std_logic_vector(IN_OP_WIDTH-1 downto 0);
        in_cmd        : in  std_logic_vector(IN_CMD_WIDTH-1 downto 0);

        out_ready     : in  std_logic;
        out_valid     : out std_logic;
        out_cmd_valid : out std_logic;
        out_opcode    : out std_logic_vector(OP_WIDTH-1 downto 0);
        out_cmd       : out std_logic_vector(CMD_WIDTH-1 downto 0)
    );
end entity;

architecture rtl of renormalizer is
begin
-- Simply foreward the ready and valid commands no latency loss here
in_ready        <= out_ready;
out_valid       <= in_valid;
out_cmd_valid   <= in_cmd_valid;
out_opcode      <= in_opcode;
{% set out_fields = out_info.fields %}
{% set uoid = out_fields['unique_ns_order_id'] %}
{% for market_name, market_info in markets_dict.items() %}
{% set in_fields = market_info.fields %}
{% set mkt_loop = loop %}
{{market_name}}_renormalizer : if FAMILY_NAME = "{{market_name}}"
↪  generate
```

```
52  process(in_opcode, in_cmd)
53  variable in_opcode_v    : natural range 0 to 99;
54  variable out_cmd_v      : std_logic_vector(CMD_WIDTH-1 downto 0);
55  begin
56  in_opcode_v := to_integer(unsigned(in_opcode));
57  -- Adding default zero value for the output
58  out_cmd_v := (others => '0');
59  case in_opcode_v is
60  {% for command, command_info in market_info['commands'].items() %}
61  {% if command_info['width'] != 0 %}
62  when {{"{:2d}".format(command_info['opcode'])}} =>
63  {% for field in command_info.fields %}
64  {% set in_field = in_fields[field] %}
65  {% set out_field = out_fields[field] %}
66  {% if in_field.width == out_field.width %}
67  out_cmd_v({{out_field.offset + out_field.width - 1}} downto
    ↪  {{out_field.offset}}) := in_cmd({{in_field.offset + in_field.width
    ↪  - 1}} downto {{in_field.offset}});
68  {% else %}
69  {% if in_field.is_signed %}
70  out_cmd_v({{out_field.offset + out_field.width - 1}} downto
    ↪  {{out_field.offset}}) :=
    ↪  std_logic_vector(resize(signed(in_cmd({{in_field.offset +
    ↪  in_field.width - 1}} downto {{in_field.offset}})),
    ↪  {{out_field.width}}));
71  {% else %}
72  out_cmd_v({{out_field.offset + in_field.width - 1)}} downto
    ↪  {{out_field.offset}}) := in_cmd({{in_field.offset + in_field.width
    ↪  - 1}} downto {{in_field.offset}});
73  {% endif %}
74  {% endif %}
75  {% if field == 'ns_order_id' %}
76  out_cmd_v({{uoid.offset + out_field.width - 1 + (mkt_loop.index |
    ↪  log(2) | ceil | int ))}} downto {{uoid.offset}}) :=
    ↪  "{{"{:b}".format(mkt_loop.index0)}}" & in_cmd({{in_field.offset +
    ↪  in_field.width - 1}} downto {{in_field.offset}});
77  {% endif %}
78  {% endfor %}
79  {% endif %}
80  {% endfor %}
81  when others => out_cmd_v := (others => '0');
82  end case;
83  out_cmd <= out_cmd_v;
84  end process;
85  end generate;
86  {% endfor %}
87  end architecture;
```

Appendix D

commands.json

The following json is an extarct from the **nasdaq_bats_commands.json** file used to generate the VHDL command package for the studied family.


```
{
    "commands":{
        "del_order": {
          "fields": [
            "buy_nsell",
            "do_not_publish_flag",
            "exponent",
            "instr_id",
            "instr_seqnum_valid",
            "latency",
            "latency_enable",
            "market_seqnum",
            "market_seqnum_valid",
            "market_time",
            "market_time_valid",
            "ns_order_id",
            "ns_order_number",
            "price",
            "quantity",
            "unique_ns_order_id",
            "unit"
          ],
          "opcode": 8,
          "width": 350
        }
    },
    "fields": {
        "price": {
          "is_signed": true,
          "mod4": false,
          "occurrence": 22,
```

```
            "offset": 171,
            "width": 33
        }
    }
}
```

Bibliography

[1] *Jinja2 Documentation*, 9 2017. Release 2.9.6.

[2] E. S. Shin, V. J. Mooney, and G. F. Riley, "Round-robin arbiter design and generation," in *15th International Symposium on System Synthesis, 2002.*, pp. 243–248, Oct 2002.

[3] S. Rudregowda, "Implementation of bus arbiter using round robin scheme," *IJIRSET*, vol. 3297, 07 2014.

[4] Altera-Intel FPGAS, *Avalon Interface Specification*, 1 2020. Rev. 17.

[5] Xilinx, *Large FPGA Methodology Guide - Including Stacked Silicon Interconnect (SSI) Technology*, 10 2012. Rev. 14.3.

[6] S. T. Trade, "The history of high frequency trading from 1602 to the present day," 2017.

[7] Wikipedia, "High-frequency trading," 2017.