Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 510

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

CERN

A Very Cool Master Thesis

by

Paolo Rondot

Supervised by

Prof. Alonso Gustavo

April 2024–September 2024

D INFK

Acknowledgements

Abstract

Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 CERN

2.1.1 ATLAS Experiment

2.1.2 Liquid Argon Calorimeter

2.2 High Level Synthesis

2.2.1 Catapult CCOREs

When designing blocks, Catapult offers the possibility to set them as CCORE. This directives should be used for elementary blocks used many times throughout the design. CCOREs are synthesized and optimized as a single block once and then reused where needed. This can save synthesis run time and area on the FPGA.

Two types of CCORE are offered by Catapult: sequential or combinational. The former should be used when registers are required for sequential steps, like counters, while the latter should be used for combinational logic, such as routing. Throughout the thesis, the CCOREs played an important role, and setting them correctly was crucial for the optimization process.

Chapter 3

Latome Hardware Design

3.1 Specifications

3.1.1 The Liquid Argon Calorimeter Data Path

The LAr Calorimeter is made of 180,000 cells which are then summed up to form 34,048 supercells. The 12-bit readings of 8 supercells are serialized into one packet and sent through one optic fiber. The 116 LATOME boards are each connected to 48 optic fibers, thus receiving data from $48 \times 8 = 384$ supercells. These boards are responsible for transforming the ADC levels into energy levels, reorganizing the data and summing some energies for LATOMEs covering specific regions of the detector and distributing the data to the Feature Extraction (FEX) system. The FEX system is responsible for processing the data and sending it to the Level 1 Trigger system.

The FEX system is made of 3 groups: the Global Feature Extractor (gFEX), the Jet Feature Extractor (jFEX) and the Electron Feature Extractor (eFEX). Within these groups, gFEX only contains one board which gets summed data from all LATOMEs, thus showing the lowest granularity but with one board covering the whole detector. Then the jFEX is made of 6 boards, each receiving data from 19 LATOMEs, and finally the eFEX, with the highest granularity, is made of 24 boards, each receiving data from 4 LATOMEs.

3.1.2 The LATOME Specifications

The part of the LATOME firmware responsible for the conversion of ADC levels into energy levels is referred to as *User Code* in the LAr group. This code is not owned by the LATOME HLS team. However, the team is responsible for the different data organization steps, also called mapping or remapping, and the summing of energies for LATOMEs in specific regions of the detector. Additionally, since the output energies are only encoded on 10 bits, it is necessary to compress the data from 12 to 10 bits via a Multi Linear Encoder (MLE).

Initially, the whole LATOME firmware was written in VHDL. The implementation of all the functionalities took over 8 years and still had timing violations.

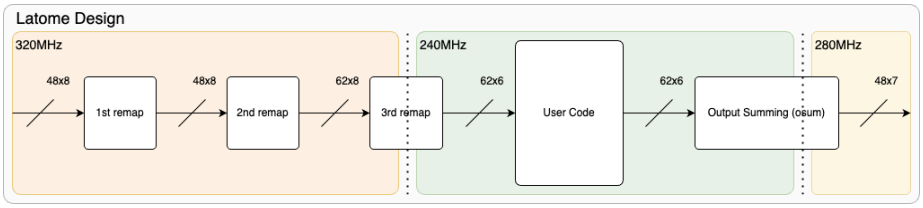


Figure 3.1: Original VHDL Design

Because in the next runs of the LHC, the amount of energy and collision is expected to increase, the LATOME design should be optimized to leave space for other functionalities, and sustain the increase in data.

3.2 Current LATOME Firmware

The original design is characterized by more clock domain crossings than the current one.

Because bunch crossings (BC) happen at a frequency of 40MHz, it is necessary to run the different blocks of the design at multiples of this frequency. The figure**??** shows three different frequencies being used. At the first remap stage, since the frames are made of 8 words, it is necessary to run the blocks at 320MHz. Then the frames become 6 words long, corresponding to a frequency of 240MHz. Finally, the FEX systems expect 7 32 bits words, thus the frequency is 280MHz.

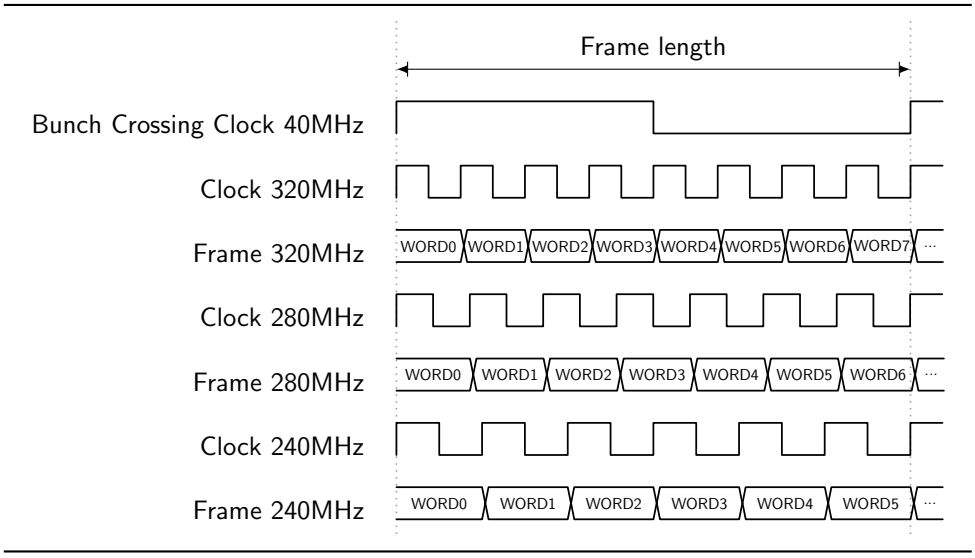
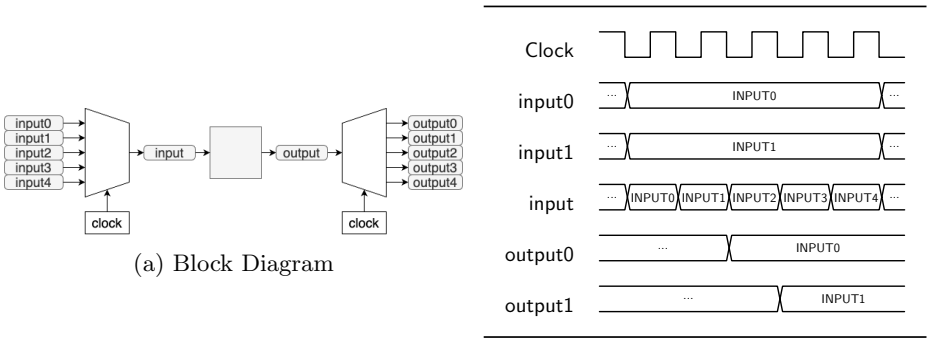


Figure 3.2: Different clocks used in the design

These clock domain crossings made the original implementation very complex...



(a) Block Diagram

(b) Timing

Figure 3.3: Space Division Multiplexing

3.3 Leveraging Time vs Space Division Multiplexing

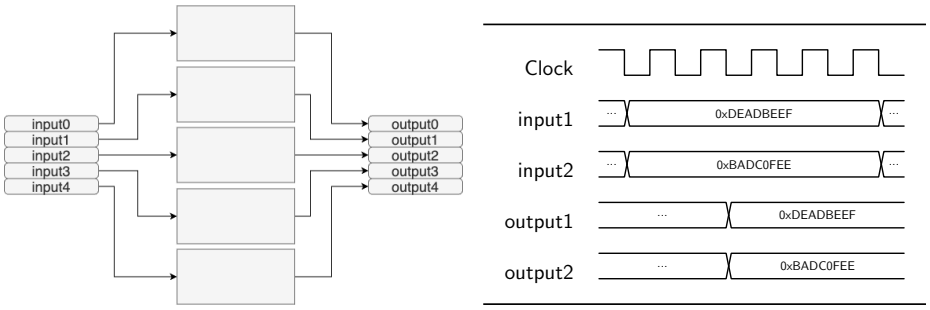
When a chunk of data is available at a given time for processing, it can all be processed at the same time, or pieces after pieces. Processing simultaneously all the data offers the advantage of a reduced latency, but requires creating copies of the processing blocks for each piece. A sequential processing gives the possibility of having just one processing block treating the data step by step at a price of latency.

A simple parallel with human body is our hands. When writing a text, because of our small number of hands we are condemned to write texts one letter after another. But if we imagined having 1000 hands, and enough space to hold as many pen without interfering with one another, we could imagine writing whole paragraphs at once.

3.3.1 Time Division Multiplexing

The time division multiplexing approach consists in sharing the resources, or processing blocks, by routing each piece of data depending on time t. Choosing this implementation can be intuitive as computers usually process data using frames of 32 to 64 bits. It has the advantage of reusing design blocks hence possibly reducing area usage, but performs poorly in case of data dependencies. If the word from time $t = 0$ of the frame needs to be available at the same time as the one of $t = 4$, the logic becomes very complex thus canceling the positive area impact unlocked by resource sharing.

This sequential processing approach requires counters used by multiplexers to latch the input data at the right moments. This approach is represented in figure **??**.



(a) Block Diagram (b) Timing

Figure 3.4: Space Division Multiplexing

3.3.2 Space Division Multiplexing

Another way of treating the data is to make it all available at the same time, as parallel inputs of the processing block. This approach is more natural as it is closer to what humans experience; we see, hear, feel, taste at the same time, not one after the other. It is represented in figure **??**. Space Division Multiplexing generally offers small latencies, as every block runs in parallel and reduces the use of sequential logic which would be required in TDM. In fact, using a parallel logic, some processing blocks can become purely combinational ones.

3.3.3 Tradeoffs

Whether one approach is better than the other is impossible to determine prior to having a deep understanding of the data processing blocks. The most common tradeoff is the one of latency at the price of area. But the Maximal Operating Frequency (F_{max}) can also be traded to improve other metrics.

Having access to all variables at the same time makes the code development easier.

3.4 New HLS Design

When developing the new firmware from the specifications in HLS, very interesting insights from Dr. Marcos Oliveira emphasizing benefits of space division multiplexing were taken into account. By using parallel full frames, the frequency of the design blocks is not locked anymore depending on the frame size. This allows to reduce the logic dedicated only to synchronization and time-division multiplexing which can become very complex when there are dependencies between the words of the frame.

The figure **??** shows that all the remapping and processing blocks except for the User Code are being processed using parallel data. This is possible thanks to the serial-to-parallel and parallel-to-serial data transformation blocks. This approach drastically simplifies the design and showed very promising results in terms of area reduction, reduced latency.

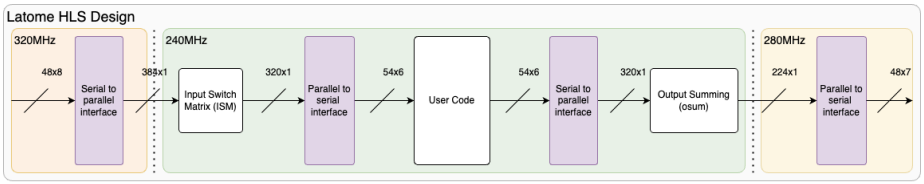


Figure 3.5: Current HLS Design

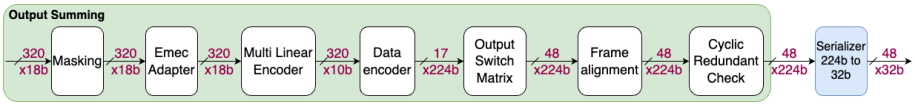


Figure 3.6: Output Summing Block Diagram

The Output Summing was completely redesigned and broken down in different blocks which are represented in figure **??**. It is made of the following blocks:

- Masking: responsible for disabling some inputs depending on the detector region that a specific LATOME treats
- Emec Adapter: this block performs additional summing for the LATOME responsible for the EMEC region of the detector
- Multi Linear Encoder (MLE): Since the ADC readings are 12 bits long but the FEX systems requires 10 bits long energy levels, this block encodes the ADC readings on 10 bits
- Data Encoder: Once the encoded energy levels are ready, this block creates 17 frames of 224 bits representing the whole data
- Output Switch Matrix (OSM): This switch matrix routes the 17 224 bits frames to the different FEX systems that the LATOME is connected too
- Frame Alignment: Each couple of BC, the LATOME does not send any data but instead it sends an alignment frame built by this block
- CRC9: This final block appends to the last 9 bits of each frame a CRC9 value which it computes

Chapter 4

Implementation

4.1 Different Spaces of Design Exploration

The optimization of the LATOME firmware can be done by leveraging different division multiplexing approaches, HLS directives or design implementations. These are three different spaces of design exploration that were explored to improve the firmware. This chapter will present the different explorations that were conducted in the context of this thesis in a first section. Next, the second section will focus on the results obtained after testing these different options.

4.2 Methodology

4.2.1 Time Division Multiplexing

The v5 of the LATOME firmware only uses the TDM approach. The data frames are broken down in different words which are processed sequentially. On the other hand, the v6 firmware leverages almost only the SDM approach with parallel interfaces for the ISM and OSUM blocks, using deserializer and serializer.

As the base code for this thesis is the v6 HLS firmware, the TDM design space can be explored by moving up or down the serializer currently placed at the output of the OSUM. The figure **??** shows the different possible positions for the serializer.

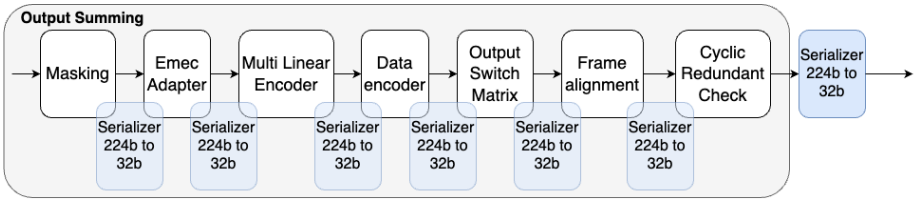


Figure 4.1: Current and potential positions of the serializer

The serializer at the output of the OSUM breaks down frames of 224 bits into 7 words of 32 bits. The main intuition behind this design exploration is that using smaller data words could reduce the size of some design blocks. Meanwhile, the latency of the design remains unchanged, as the serializer is only moving up in the design.

During this thesis, the serializer was moved up to two main different positions. The first one was before the Output Switch Matrix, and the second one was before the Data Encoder. In both cases, the OSM, Frame Alignment and CRC9 blocks were modified to handle the new data format.

An important aspect to check is the data dependencies, since these could have a critical impact when using TDM. The OSM block is a routing block which does not change any data in the frames. The Frame Alignment only overwrites the data that it gets as input if the current BC index is an alignment one, hence it does not show data dependencies. Finally the Cyclic Redundant Check can be both computed in a parallel or serial fashion equivalently and only depends on the CRC temporary result from time $t = -1$. Having no data dependencies, the TDM approach can be used without any issue.

The new design of these blocks required little code changes but introduced new sequential logic, like counters, to keep track of the current word being processed. Additionally, the operating frequency becomes $7 \times 40 = 280$MHz, as the serializer must process 7 words of 32 bits in 7 clock cycles and the OSUM must maintain a throughput of 7 clock cycles to never miss new data from the detector.

Output Switch Matrix

The different LATOME boards must route the data differently to the FEX systems depending on the detector region they are connected to. To tackle this problem, either each LATOME has a different version of the firmware, or some code must be added to allow an online configuration procedure defining the routing. The second possibility was chosen in the LAr group, and the solution proposed by Marcos for the v6 firmware was to add an Output Switch Matrix.

The OSM uses multiplexers to route the 17 inputs to the 48 outputs and takes its selection bits from registers configured online via an ipbus. The HLS implementation is very simple, involving a 2D array with 48 rows made of a subgroup of the 17 possible inputs.

Without looking at the different LATOME routings, it appears that, there should be 48 multiplexers of input size $224 \times 17 = 3808$ bits. However when considering the data paths and optimizing for them, the biggest multiplexers turn out to have at most 4 input frames, meaning that each row from the HLS 2D array has only 4 entries. In other words, the Output Switch Matrix is a sparse matrix. By moving the serializer before the OSM, the biggest multiplexers have an input size of 4×32 instead of 4×224.

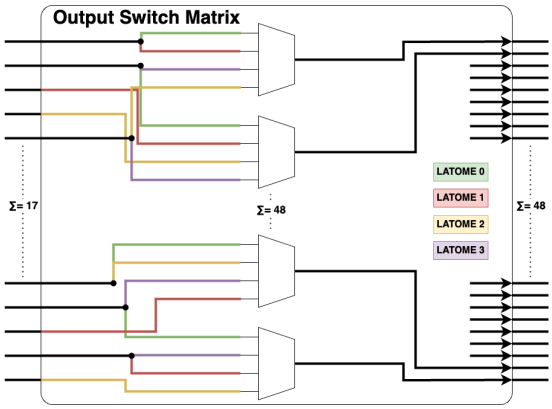


Figure 4.2: Output Switch Matrix Simplified Diagram

Frame Encoder

The frame encoder block is responsible for creating the 17 frames of 224 bits representing the whole data. When using parallel data, the block is very simple and only needs to concatenate and order the different data words. Serializing before the Frame Encoder, makes the building of 224 bits frames unnecessary.

The Frame Encoder takes as input the 320 MLE results, the control bits and the BCID and assembles them. To create the 32 bit words, a counter is used to keep track of the current word being processed. A "switch" statement is used to handle the different cases.

Impact on the Following Blocks

The seven sequential cycles needed to process the serial words making up one frame are created using a simple for loop without catapult unroll directives. The frame alignment and CRC9 blocks use a counter to keep track of the current word being processed. In the case of the CRC9, since the code is appended on the 9 last bits of the frames, the block must first compute 6 32-bit word followed by one 23-bit word. The overhead of having two specialized blocks is considered to be very small as each of them only contain a limited number of XOR gates.

The frame alignment block reads the current BC index and if it matches the alignment index, it overwrites the data with the new data, otherwise it simply forwards it. The introduction of a simple counter indicating which of the 7 words is currently being processed together with a "switch" statement taking care of the different cases is enough to get a TDM version of this block. Additionally, this block sends a 4 bit control character indicating for each byte if it is a data byte or a control byte. Before the serialization, the block sent the complete 28-bit control word, now it sends 4 bits at a time.

4.2.2 Leveraging HLS Directives

The second space of design exploration is the HLS directives, specifically, the CCORE related ones. Each processing block of the Output Summing is declared as a CCORE one. This makes Catapult optimize the design inside of the CCORE once to reuse it for all the instances. The two options offered by Catapult (sequential or combinational) were tested for each block.

Additionally, Catapult gives the possibility to set the design goal for the optimization to be either latency or area. Setting this goal to either one will have an important impact on the final result.

Overall, there are six different blocks with each two options for the CCORE and two options for the optimization goal. This gives a total of 128 different possible combinations. The table **??** shows the different possible combinations. The CRC9 block is not included as it is, by essence, a sequential block, and cannot be set as combinational.

	Option	Value
CCORE	Masking	Sequential / Combinational
	Emec Adapter	Sequential / Combinational
	MLE	Sequential / Combinational
	Data Encoder	Sequential / Combinational
	OSM	Sequential / Combinational
	Frame Alignment	Sequential / Combinational
Optimization Goal		Latency / Area

Table 4.1: Possible values for each HLS option

In the OSUM, many "for" loops are used to create the different instances of the blocks. A full unroll means that the for loop creates as many parallel instances as there are iterations. In some cases, a partial unroll can be used to create a fixed number of instances which are reused to process all the data. For example a for loop of 320 iterations can have an "unroll 4" directive to create 4 instances that will process 80 iterations each. This can lead to very high area reduction at the price of latency. The choice of full or partial unroll can be done manually, or left to Catapult to decide. For this design exploration, we have given Catapult the freedom to choose the best unroll factor, within the following constraints:

- The initiation interval (throughput) must be 7 clock cycles.
- The maximum latency is 26 clock cycles.

The first value is a direct consequence of the 280MHz operating frequency, while the second value is the latency of the v5 design, which should not be exceeded.

To create all the different combinations, a tcl script looping through each of them and launching the Catapult synthesis was used. The results were then analyzed to find the best combination of options. A follow-up Quartus compilation analysis was done to get more precise and meaningful results. Overall, the HLS synthesis run time for all the combination lasted around 9 hours, while the Quartus compilation for all the combinations lasted 3 days.

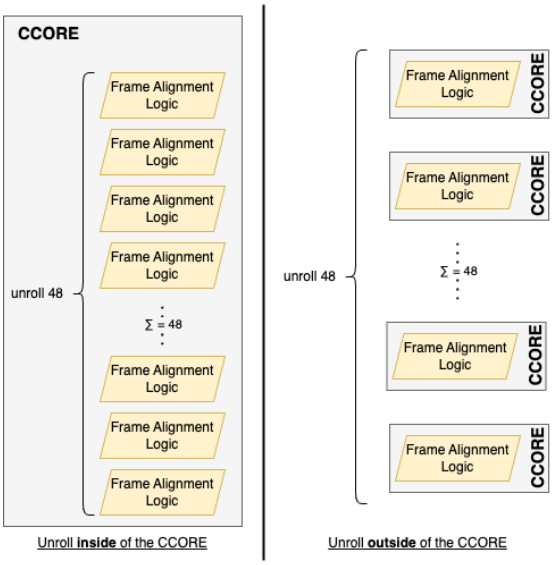


Figure 4.3: Unrolling inside vs outside the CCORE

4.2.3 Unrolling Inside or Outside the CCORE

When developing the HLS code for these design blocks, two options can be considered using CCOREs in for loops, as illustrated in figure **??**. The first one is to set the OSM, Frame Alignment or CRC9 as an elementary block processing only one frame at a time, and to create as many copies as there are frames to process. The second option is to make the design block process all the frames, and to set it as a CCORE. Using the HLS jargon, the first option refers to *unrolling* "outside" of the CCORE, while the second refers to *unrolling* "inside" of the CCORE.

4.3 Design Space Explorations Results

To get results from a specific optimization, one can compile only the block being studied, the block within its parents block, or the whole firmware. In the LATOME HLS team, four levels of compilation exist and can be considered in this thesis: the block alone, within the OSUM, with the ISM and finally the whole firmware. Improvements on one level does not necessarily translate to the next as Quartus may or may not be able to perform further optimizations. However, at each level, the results can be compared to the original design to get a sense of the impact of the optimization. Additionally, the smaller compilation times of the lower levels can be used to quickly test different options.

4.3.1 Time Division Multiplexing

Serializing Before the OSM

The expected results at the block level are an increase of the maximum operating frequency, as less signals must be synchronized, as well as an area reduction of a factor of 7.

Version	Area (ALMs)	F_{max} (MHz)
Original OSM	3621.5	310.17
Serialized OSM	476.8	524.38
Improvement	−86.8%	+69.1%

Table 4.2: Results From the OSM Optimization

The table **??** shows an actual area reduction of a factor of 7.6 and an increase of the maximum operating frequency of 69%. The area reduction is slightly higher than the expected factor of 7, which can be explained by the overhead that the signal synchronization logic had on the original design.

Serializing Before Frame Encoder

4.3.2 HLS Directives

Once the results were obtained, they were extracted in an Excel file to be analyzed. A python script created the different figures presented in this section.

The first figure, figure **??**, shows the overall results for all the combinations, each represented by a point. The color of the points indicate the latency for this solution. Finally, the blue horizontal line shows the operating frequency of the OSUM block, 280MHz. This figure shows that the solutions with the highest latencies are the ones with the worst F_{max}. This is explained by the fact that when Catapult implements resource sharing, with partial unrolling, the latency increases, but the resource sharing logic has a negative impact on the F_{max}.

In figure **??**, the points are "+" when the goal for the solution is latency and "o" when it is area. Each line connects two solutions with all the same options except for the optimization goal. The line colorization depends on the latency. For instance, the points on the left of the figure all have lines going from bottom left to top right, with a color gradient from red to yellow. Additionally, the points at the bottom left are all area optimized solutions, while the ones at the top right are latency optimized solutions. This indicates that using the latency optimization goal will lead to a higher F_{max}, lower latency at a small area cost, since the lines have a very big slope. Two clusters can be clearly distinguished, the one on the left showing a clear tendency and the one on the right showing a more scattered distribution.

Figures **??**, **??**, **??**, show no clear tendency when varying the CCORE type for the Masking, EMEC Adapter and Data Encoder blocks. These blocks are more secondary ones for the optimizations and are very dependent on the more important ones. Hence they should be used for adjustments once the main blocks are optimized.

The figures **??**, **??**, **??** show the results when varying the CCORE type for the MLE, OSM and Frame Select blocks. The MLE block shows a clear tendency: all the points on the right are combinational ones, while the ones on the left are sequential, the lines have a very small slope, and points are far from each other. This means that setting the MLE CCORE as sequential instead of combinational has a small impact on the F_{max}, but reduces the area drastically. The OSM has almost no impact on the area, since the lines are vertical, but has one on the F_{max}, which is very dependant on the other blocks' settings. Finally, most of the lines for the Frame Select block are going from the bottom right where the combinational points are to the sequential ones on the top left, indicating that the sequential setting has a positive impact both on the F_{max} and the area.

4.3.3 Unrolling inside or outside the CCORE

4.3.4 Output Switch Matrix Optimizations

4.3.5 Description

The different boards must route the data differently to the FEX systems depending on the detector region they are connected to. To tackle this problem, either each LATOME has a different version of the firmware, or some code must be added to allow an online configuration procedure defining the routing. The second possibility was chosen in the LAr group, and the solution proposed by Marcos for the V6 firmware was to add an Output Switch Matrix.

The OSM uses multiplexers to route the 17 inputs to the 48 outputs and takes its selection bits from registers configured online via an ipbus. The HLS implementation is very simple, mainly involving a 2D array with 48 rows made of a subgroup of the 17 possible inputs.

Without looking at the different LATOME routings, it appears that, there should be 48 multiplexers of input size $224 \times 17 = 3808$. However when considering the data paths and optimizing for them, the biggest multiplexers have at most 4 input frames, meaning that each row from the HLS 2D array have 4 entries. Figure **??** shows the parallel to serial block creating 32 bits words for the FEX systems. Moving this block to the left would have no impact on the latency. However, by moving this block before the Output Switch Matrix, the biggest multiplexers would have an input size of 4×32 instead of 4×224.

An important aspect to monitor is the data dependencies, since these could have a critical impact if using TDM. The OSM block is a routing block which does not change any data in the frames. The Frame Alignment only overwrites the data that it gets as input if the current BC index is an alignment one, hence it does not show data dependencies. Finally the Cyclic Redundant Check can be both computed in a parallel or serial fashion equivalently and only depends on the CRC temporary result from time $t = -1$.

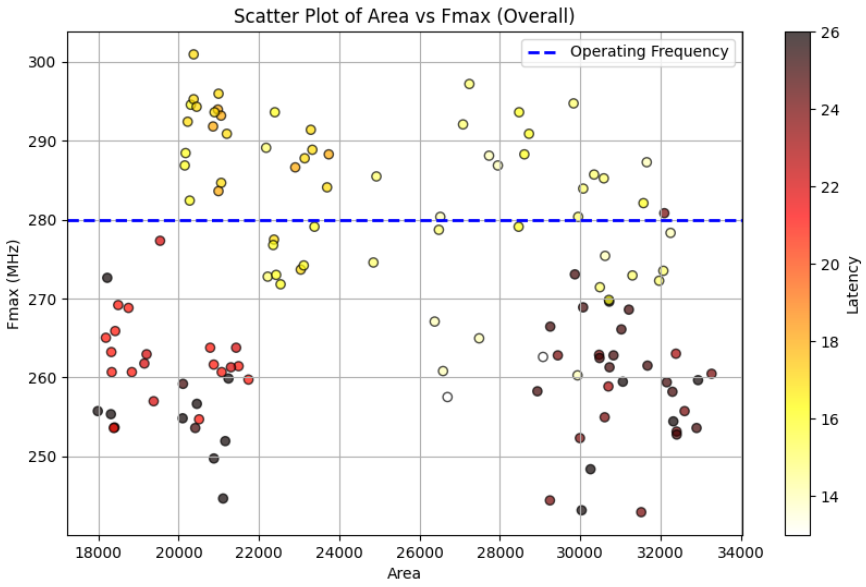


Figure 4.4: Overall Results

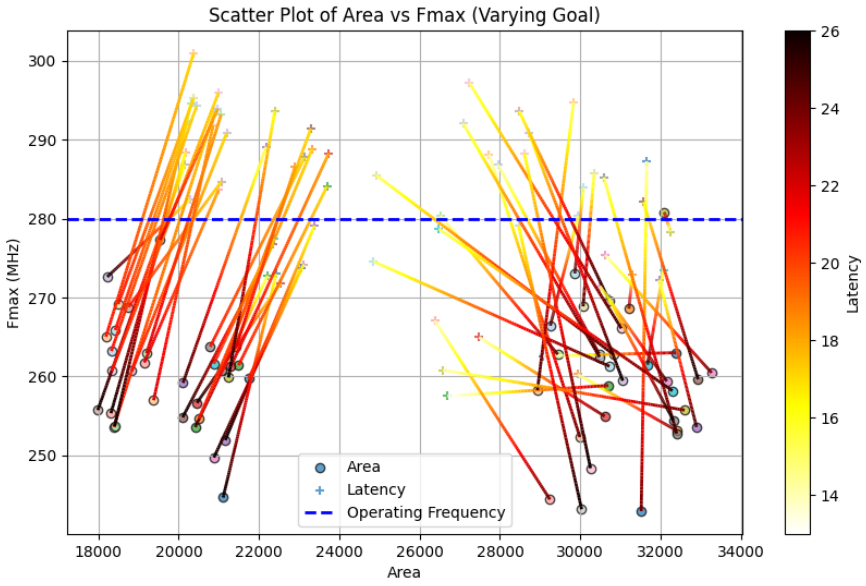


Figure 4.5: Results when varying the design goal

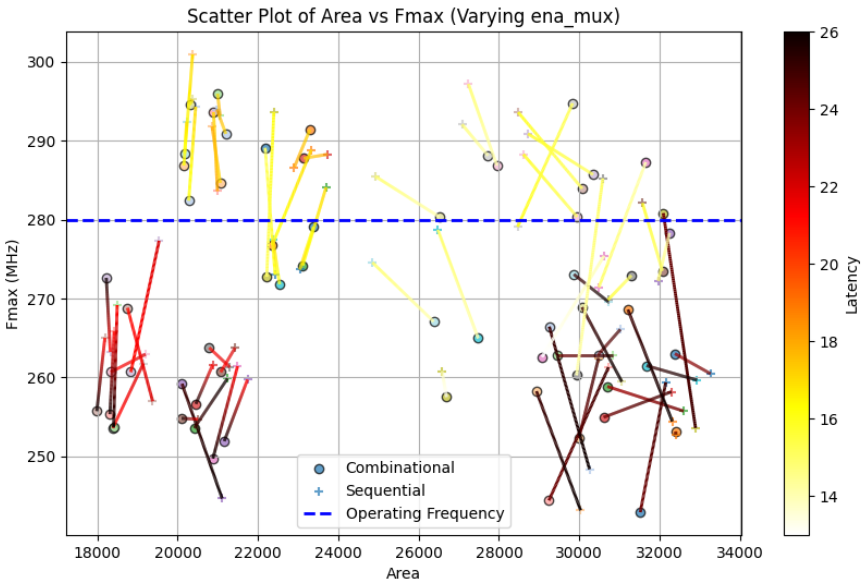


Figure 4.6: Results when varying the CCORE type for the Masking

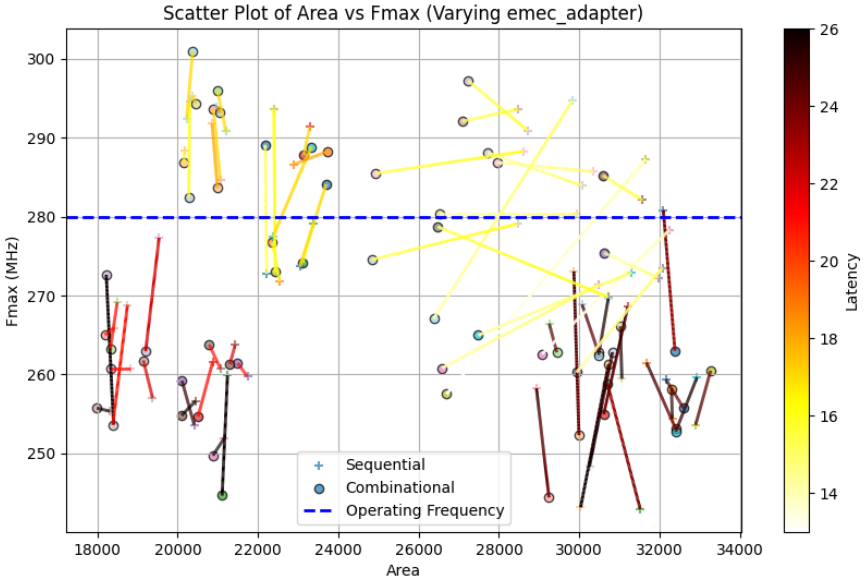


Figure 4.7: Results when varying the CCORE type for the EMEC Adapter

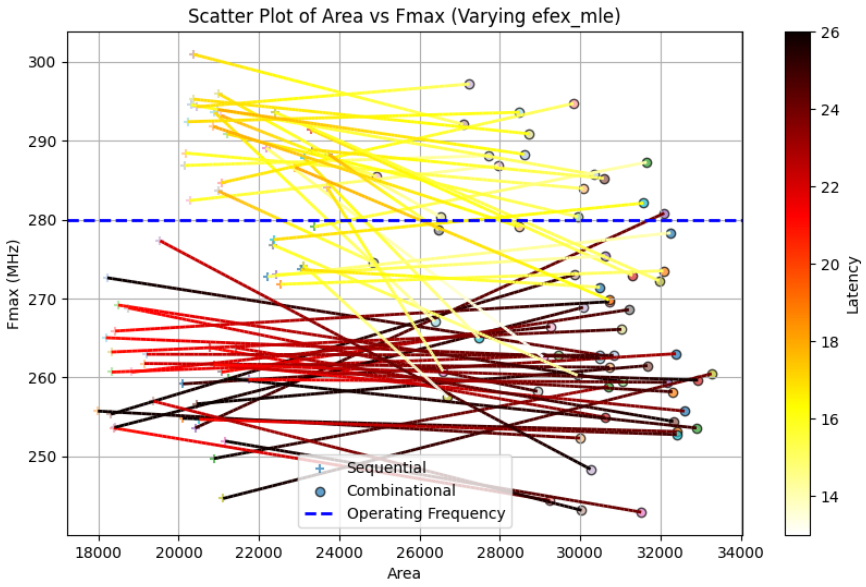


Figure 4.8: Results when varying the CCORE type for the MLE

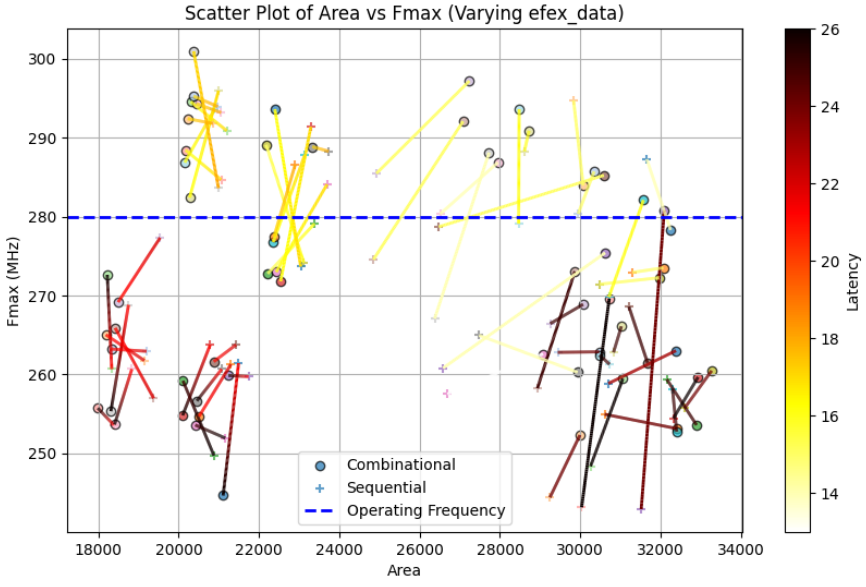


Figure 4.9: Results when varying the CCORE type for the Data Encoder

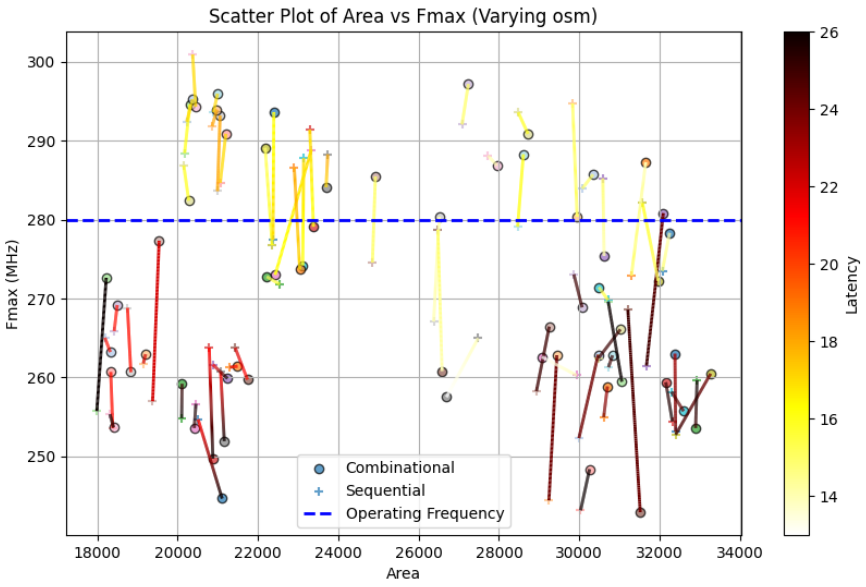


Figure 4.10: Results when varying the CCORE type for the OSM

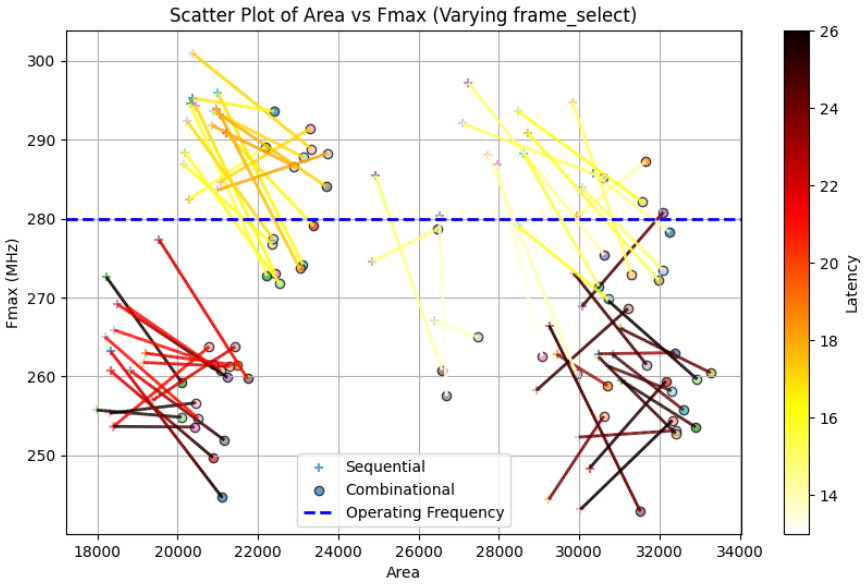


Figure 4.11: Results when varying the CCORE type for the Frame Select

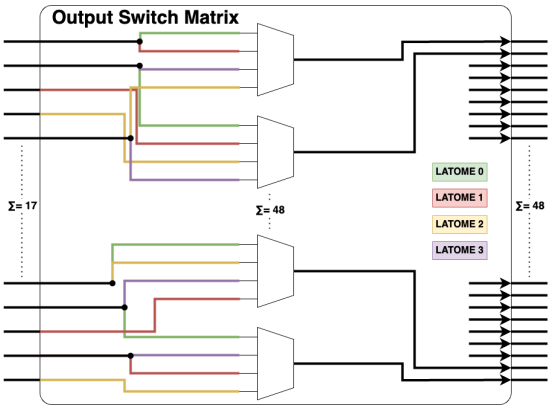


Figure 4.12: Output Switch Matrix Simplified Diagram

4.3.6 Serializing Before the OSM

4.3.7 Improvements

4.3.8 Serializing the Next Blocks

As the figure **??** shows, the Frame Alignment and CRC9 blocks are two blocks that should be modified to handle the serial inputs.

Frame Alignment

When developing the HLS code for these design blocks, two options can be considered using CCOREs, as illustrated in figure **??**. The first one is to set the OSM, Frame Alignment or CRC9 as an elementary block processing only one frame at a time, and to create as many copies as there are frames to process. The second option is to make the design block process all the frames, and to set it as a CCORE. Using the HLS jargon, the first option refers to *unrolling* "outside" of the CCORE, while the second refers to *unrolling* "inside" of the CCORE.

In the case of the Frame Alignment, the V6 firmware uses the first option. One frame alignment CCORE uses very little area, and because it uses only combinational logic, using the second option does not have any significant impact on the area. In table **??**, we see that with the first option yields a Frame Alignment of 21 ALMs. To compare it with the second option, we can multiply it by the number of instances that will be created: 48, for the 48 frames. This gives a total area of 1008, very similar to the 1087 of the second option. The F_{max} parameter, on the other hand, decreases by 98MHz, but this should be disregarded as the "unrolling inside" version of the CCORE contains the logic to synchronize the different instances.

To have an idea of the true impact that the two implementations could have on the full firmware, the table **??** also shows the area and F_{max} of the OSUM parent block. The low impact on the area is confirmed by our results, but the F_{max} gets a decrease of 3.6%, reducing the margin of the design.

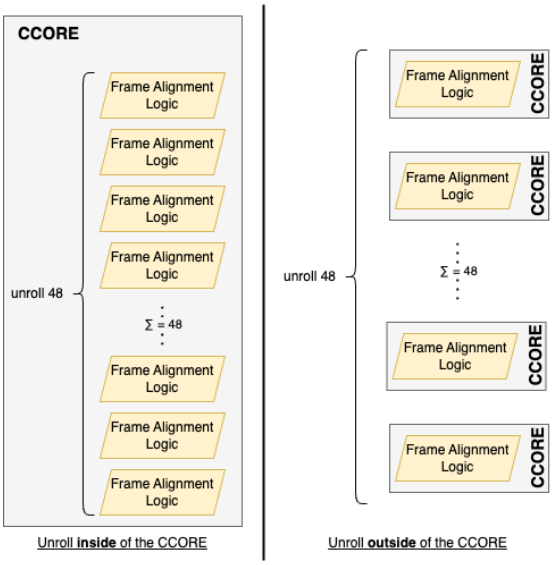


Figure 4.13: Unrolling inside vs outside the CCORE

unroll outside		
Block	Area (ALMs)	F_{max} (MHz)
Frame Alignment	21 ($21 \times 48 = 1008$)	371.4
OSUM	30449	274.5
unroll inside		
Block	Area (ALMs)	F_{max} (MHz)
Frame Alignment	1087 ($1087/48 = 22.6$)	272.6
OSUM	30512	264.6

Table 4.3: Frame Alignment design exploration using the V6 firmware

In fact, using the second option can be useful in case some sequential logic could be shared among all the CCOREs. The serialized Frame Alignment is a good example where unrolling outside has a positive impact, as it can be seen in the table **??**. In this specific case, Catapult is able to optimize the design inside of the CCORE by sharing some logic between the different parallel instances. As a result, unrolling inside of the CCORE will create redundant logic and the total area of the CCORE instances is almost the double of the "unrolling inside" version. Additionally, this negative impact gets amplified on the OSUM block, with an area increase of 15% and new timing violations with the "unroll outside" option.

CRC9

4.4 Optimizing the HLS Directives

unroll outside		
Block	Area (ALMs)	F_{max} (MHz)
Frame Alignment	$\underline{36}$ ($36 \times 48 = 1728$)	388.8
OSUM	18669	$274.7 < 280$
unroll inside		
Block	Area (ALMs)	F_{max} (MHz)
Frame Alignment	$\underline{940}$ ($940/48 = 19.6$)	345.1
OSUM	15875	291.1

Table 4.4: Frame Alignment design exploration using the new firmware

Chapter 5

Evaluation

Chapter 6

Conclusion