



UNIVERSITÀ DEGLI STUDI DI MILANO

Facoltà di Scienze e Tecnologie
Corso di Laurea Magistrale in Fisica

DATA COMPRESSION FOR READ-OUT CIRCUITS
IN THE ATLAS UPGRADE

Supervisors: Valentino Liberali
Attilio Andreazza
External supervisor: Walter Snoeys

Master Thesis of:
Lorenzo VIGORELLI
Registration Number:
882731

Contents

1	Monolithic pixel sensors for ATLAS upgrade	5
1.1	Muon spectrometer	6
1.2	Calorimeter	6
1.3	Tracker	7
1.3.1	Pixel tracker	7
1.4	MALTA detector	9
1.4.1	Modified process	10
1.4.2	Asynchronous column drain readout	10
2	Trigger memory	15
2.1	Tasks and constraints	15
2.2	Meaning and content of the data	16
2.3	Estimate of the memory size	18
3	Compression algorithm	26
3.1	Redundancies in the data	26
3.2	The compressed format	27
3.3	Hamming encoding	30
3.4	Memory size estimation with compression	32
3.5	Conclusion	37
4	Hardware high level description	38
4.1	Hardware Description Languages	38
4.2	Blocks description	39
4.2.1	Writing block	40
4.2.2	Memory cells array	43
4.2.3	Reading block	45
4.2.4	Trigger controller	47
4.2.5	Token ring	48
	Bibliography	50
A	Code	53

Introduction

In particle detectors used for high energy physics experiments, a trigger memory is often required as a temporary storage for the acquired data. In this memory the data are stored for a short period of time, until an external trigger signal informs the detector on what are the data to send out. In this case in particular the reason why a trigger memory is required is the great number of data produced inside the detector: since it is not possible to send out all the data produced it becomes necessary to select only some of them using external parameters based on independent measurements. The elaboration of such parameters requires some time, during which data are stored in the trigger memory waiting for the trigger signal is produced.

This thesis work focused on the development of a trigger memory for the MALTA chip, a pixel sensor that is being developed for the inner tracker of ATLAS Phase II upgrade. The same memory could also be used for MonoPix, a detector that is being developed in parallel to MALTA with the same technology and more in general its principles could be applied to any pixel matrix that cannot retain its data during the trigger latency. The main purpose was to elaborate a compression scheme that could meet area occupancy and data storage requirements in a high data rate environment. Therefore, the proposed compression algorithm was studied specifically to exploit the frequent occurrence of subsequent events by concatenating them, thus reducing the information required for data indexing.

Several compression formats have been contrived and their efficiency has been evaluated according to the specific environment. As the implementation of this compression scheme is intended for a trigger memory, the fundamental parameters are the latency of the trigger signal and the rate of events. Thus, for the proposed schemes, several simulations have been carried out on a subset of these parameters corresponding to the range of environments expected for this memory. Then a choice of the scheme was made, in order to meet the constraints in terms of area occupancy and maximum writing frequency even in the tougher environment.

Finally, for the chosen scheme, a compression algorithm was created and a full compression, write, read and retrieval algorithm was implemented and coded in hardware description language (using both VHDL and System Verilog). The last part of the work consisted of several iterative steps of

synthesis of the code, simulation of the corresponding circuit and code modification, until finally the the desired behaviour was achieved with a circuit capable of working at the desired frequency of 320 MHz.

Chapter 1

Monolithic pixel sensors for ATLAS upgrade

ATLAS (A Toroidal LHC ApparatuS) is one of the four main experiments at the LHC (Large Hadron Collider). It physically consists of a set of position detectors for particle tracks reconstruction, electromagnetic and hadronic calorimeters to measure the energy release and muon chambers [5].

In 2024-2026 the Large Hadron Collider will be upgraded to the High Luminosity-LHC (HL-LHC). HL-LHC will operate at a five times higher luminosity ($5 - 7 \times 10^{34} \text{ cm}^{-2}\text{s}^{-1}$). As a consequence, the detectors will have to handle higher particle densities leading to higher hit rates and consequent data rate in the detector components and higher radiation levels. The number of events per collision will increase from around 25 to up to 140-200 (depending on the luminosity) events per collision [8].

In particular the ATLAS tracker will have to be upgraded completely to cope with the increased particle rate and its inner part will consist of a 5-layer Pixel detector. My thesis work is located within this upgrade project, in particular in the context of a collaboration for the development of a CMOS pixel sensor in TowerJazz 180 nm technology. The collaboration includes two groups, one from Bonn and the other from CERN, that are investigating two possible realisations of a matrix with very similar frontend but a different readout architecture, one synchronous and more conservative, the other one asynchronous with some more risk. The synchronous solution is used in Bonn's detector (called MonoPix), while the asynchronous one has been implemented by CERN's group, whose detector is called MALTA (Monolithic pixel sensor from ALice To Atlas). Nonetheless, besides the analog front end, the matrices will also share most of the peripheral components, that are actually being designed in collaboration by both groups.

After a brief overview of ATLAS components a more detailed description of the tracker and the detector will be given.

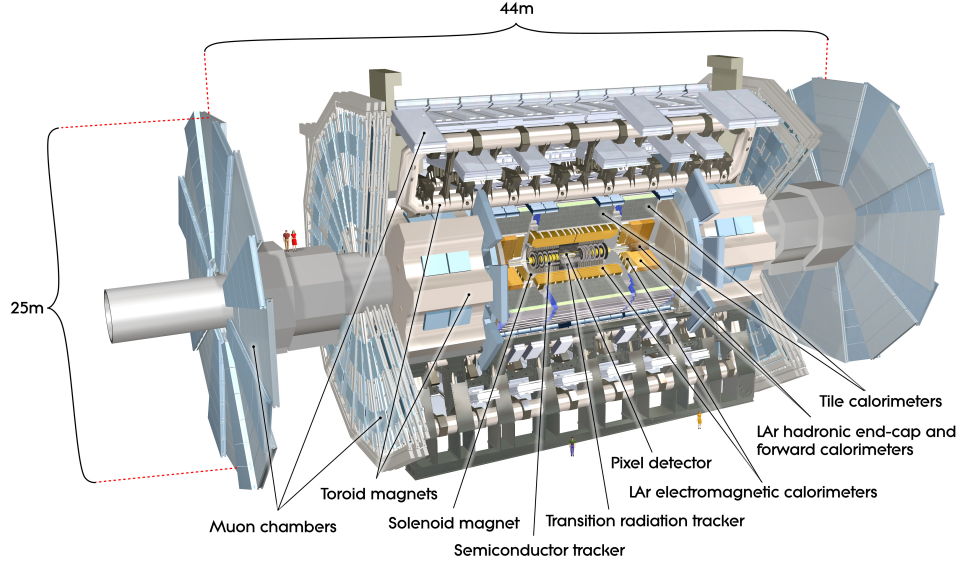


Figure 1.1: Cut-away view of the ATLAS detector

1.1 Muon spectrometer

The muon spectrometer is the external part of ATLAS experiment. It is based on magnetic deflection of muon tracks by means of superconductive toroidal magnets. The muon spectrometer purpose is to detect muons momentum and approximate position, as muons usually pass undetected through the other inner layers of detectors. In order to obtain the particles tracks the detector is composed of more than 5,000 individual muon chambers [5].

1.2 Calorimeter

The calorimeter sits inside the muon spectrometer and its purpose is to measure the energy lost by particles at their passage. ATLAS calorimetric system is made of two subsystems: an inner Liquid Argon (LAr) electromagnetic high granularity calorimeter for measuring the energy and position of electrons and photons and an outer Hadronic calorimeters for measuring the energy of hadrons [5].

Both systems are constituted of a cylindrical detector (barrel) around the interaction point and two endcaps, in order to better cover the full angle. In the electromagnetic calorimeter both detectors work with liquid Argon and have a variable positional granularity. The Hadronic calorimetric system (placed directly outside the electromagnetic calorimeter) is instead constituted of a Tile calorimeter as the barrel detector and liquid Argon

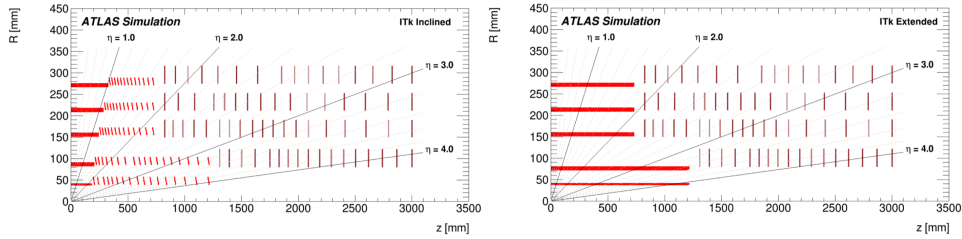


Figure 1.2: Schematic layouts of the inclined inner and outer barrel layer (left) and the extended inner barrel layers (right)

hadronic detectors at the encaps.

Calorimeters must also provide good containment for electromagnetic and hadronic showers, and must also limit punch-through into the muon system. Hence, calorimeter depth too is an important design consideration.

1.3 Tracker

The innermost set of detectors is the tracker, responsible for the reconstruction of the particle tracks by finely detecting the crossing position of particles in its detecting layers. It consists of a set of cylindrical barrels around the interaction point and a set of end caps to ‘close’ the barrels at both ends and detect particles also at larger angles.

The currently existing inner detector will be entirely replaced during the LHC Phase II shutdown by an all-silicon tracking system called the ITk (Inner Tracker). It will consist of a 5-layer Pixel detector as the inner part and a 4-layer Strip detector as the outer part. All components of the new Inner Tracker are currently under development [8].

1.3.1 Pixel tracker

For the 5 layers of pixels two main layouts have been proposed: one with traditional barrels and endcaps and the other implementing also some tilted detectors, the two layouts are shown in Figure 1.2. The coordinate η in the figure is called pseudorapidity and is commonly used in particle physics to describe the angle θ between a particle and the beam axis, it is defined as:

$$\eta \equiv -\ln \left[\tan \left(\frac{\theta}{2} \right) \right] \quad (1.1)$$

It is important to note that the smaller is the angle the larger is the pseudorapidity. Since a common problem in particle physics experiments is to record events also in the region close to the beam, then the problem becomes that of having a large η coverage. These solutions have been conceived for

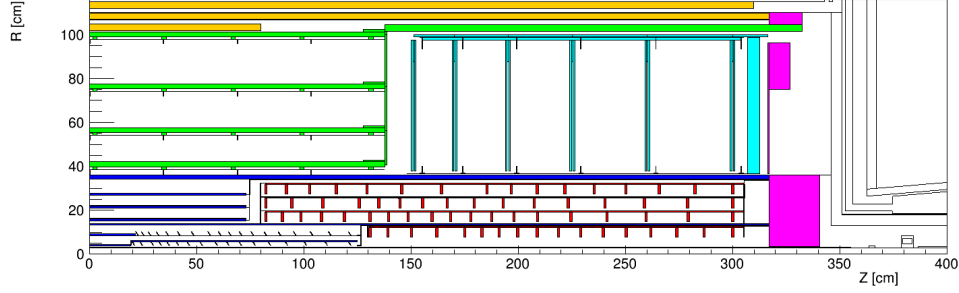


Figure 1.3: Schematic layout of the tracker including both pixels and strips detectors. The five barrel pixels layers are shown in blue and the tilted detectors of the first two layers are visible

this reason and actually are both capable of covering large pseudorapidity angles.

To achieve this, in the more traditional solution the two inner barrels are simply extended as close as possible to the end caps. In fact, due to the layout of the end caps, it is possible to extend the inner layers more than the others, as shown on the right of Figure 1.2. In the other layout it is also considered the option of tilting the detectors at the ends of the barrels, as shown on the left of the figure, so that the particles cross them almost orthogonally, thus leading to a slightly more complex design.

From a performance point of view the layout with inclined detectors has some advantages [4]. It allows in fact for better track reconstruction at high pseudorapidity angles (η in the figure), as particles at large η will easily cross more than one detector in this configuration, giving thus some redundancy to the tracks reconstruction. Furthermore, particles at large η will also cross less material in this case, as the detectors will be traversed perpendicularly, and strictly connected to this there is also the fact that the total required silicon surface would be lower.

Among these extreme solutions it is also possible to conceive intermediate hybrid layouts. This is actually the case of the layout considered in this thesis work, on which all the proposed simulations are based. In particular, the hit rates per chip that I have received and used are relative to the layout shown in Figure 1.3. Then, since the environment is the same even for different layouts, the proposed results maintain their validity, at least qualitatively, also for other solutions.

In this particular layout only the first two layers have both tilted and non-tilted detectors, while the other three layers don't use any tilted detector. In the next chapter we will refer only to this layout, since it is the solution considered for this work, thus the layers will be identified with an enumeration from 0 to 4 with the 0th and 1st layers that will have the attribute “tilted” or “flat”.

1.4 MALTA detector

At present the pixel detectors of ATLAS tracking system are hybrid detectors. Nonetheless, with the great technological improvements achieved in the last years in the field of monolithic CMOS pixel detectors, it has become possible to conceive tracking systems for high energy physics experiments fully based on monolithic sensors, with a great reduction of costs and an increase of the total detecting surface.

Monolithic sensors have in fact already proven their precision as tracking and vertexing devices and are now ready to be implemented in high-rate and high-radiation applications such as LHC experiments. The main difference in these applications is the fact that a fully depleted active region is mandatory in order to have sufficient signal and fast timing, as in a fully depleted region the charge is collected by drift in an electric field (and not by diffusion as usually happens in CMOS sensors). It is not easy to achieve full depletion in a CMOS sensor and to keep the depleted zone in the substrate (without interfering with the overlying electronics), also considering that the circuit needs to survive in a high radiation environment. However, all these challenges can be addressed by a high-resistivity substrate material ($>1 \text{ k}\Omega\text{cm}$), a high bias voltage (typically tens of volt)[14] and deep buried doping wells for shielding the readout electronics. With these characteristics it is in fact possible to deplete a large volume, necessary to collect the generated charge from the whole pixel matrix without leaving blind areas.

Some of these solutions have already been implemented for the next upgrade of ALICE inner tracking system, that is in fact based on ALPIDE [11], a monolithic pixel sensor successfully implemented in TowerJazz 180 nm imaging technology. The peculiarity of this detector is that it adopts a small collection electrode for low power consumption instead of using as electrode the buried doping well underlying the electronics that, due to its large surface, is highly capacitive. An additional modification of the process, agreed with the foundry, allowed the implementation of an ATLAS-specific solution for the inner tracker in this same technology. This led to the design of MALTA (Monolithic pixel sensor from ALICE to ATLAS) and MonoPix [1] [2], whose target is the outer layer of the inner tracker.

MALTA contains a 512×512 matrix of pixels with a $36.4 \text{ }\mu\text{m}$ pitch (and thus approximate dimensions of $2 \text{ cm} \times 2 \text{ cm}$), its in-pixel discrimination is based on ALPIDE [10] and it has a time response shorter than 20 ns. In order to further reduce power consumption it has a fully asynchronous readout architecture, that in fact does not require a clock distribution in the whole matrix, but only in the periphery, as indeed synchronisation will be required in the final implementation for the experiment if the MALTA architecture is chosen.

From here on I will refer to MALTA in particular, as this work has been carried out specifically for this detector. Nonetheless the same concepts,

magnitudes and, partially, the implementation apply also to MonoPix detector, as the purpose and the technology are the same.

1.4.1 Modified process

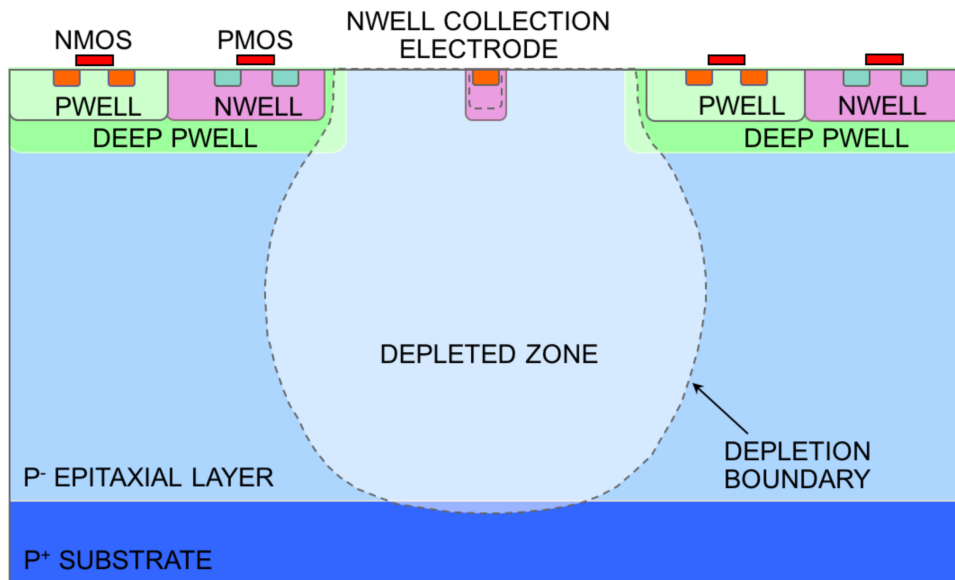
In ALPIDE a small collection electrode is used, surrounded by the pixel electronics. This solution allows for a less capacitive electrode, with the advantage of minimising analog power consumption with given analog performance [13]. However this has the disadvantage of limiting the expansion of the depleted region, as in fact the depleted region will contact the substrate before extending in the layer underlying the electronics (see Figure 1.4a). In general this would not be a problem, since the charge could still be collected by diffusion from the non-depleted region, but, after severe non-ionising radiation damage, the collection of charges by diffusion would be strongly limited by the generation of traps in the Silicon lattice. What in fact happens is that chips produced with standard TowerJazz technology are capable of withstanding non-ionising energy loss up to 10^{14} n_{eq}/cm² [6], still below the required dose of 10^{15} n_{eq}/cm² [1].

Nonetheless it was possible to develop a modified process together with TowerJazz foundry in order to achieve full depletion of the epitaxial layer [12]. The modification consists of a low dose n-type implantation underlying all the electronics and the electrodes as shown in Figure 1.4b. With this modification the sensor junction becomes planar and is extended over the full pixel width, the additional doping implantation, in fact, shifts the junction from the electrode to the plan between the n implant and the p epitaxial layer. Since depletion starts at the junction, it immediately extends over the full width of the pixel. This would actually cancel the advantage of using a small capacitance electrode, since the capacitance in this case would be given by the full junction. Nonetheless, by applying sufficient reverse substrate bias, the depletion will extend to the n-well collection electrode, thus exploiting its small size to achieve a small capacitance. Since the modification does not interfere significantly with the circuit layout, it is possible to produce the same design in both processes with the addition of only one mask, allowing a direct comparison between the two.

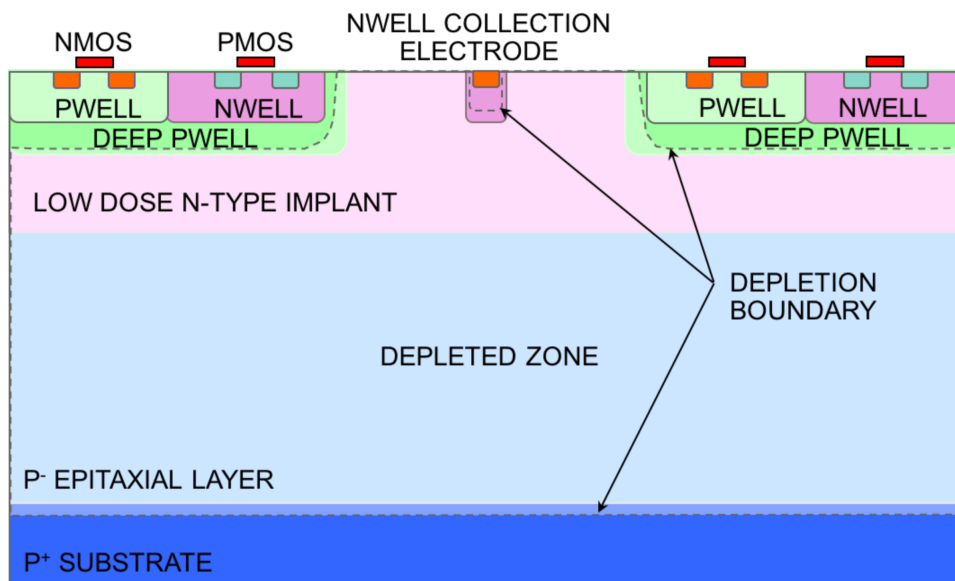
1.4.2 Asynchronous column drain readout

The readout architecture of MALTA detector has been studied specifically for low power consumption. For this purpose it has been designed to work completely asynchronously, thus avoiding the clock distribution with the relative power consumption.

The adopted solution is an asynchronous version of the column drain architecture [1]. It is in fact based on columns and all the data are sent out to the periphery in one particular direction along the columns. It relies



(a) Standard process



(b) Modified process

Figure 1.4: Comparison between standard and modified TowerJazz process. Reproduced from [12].

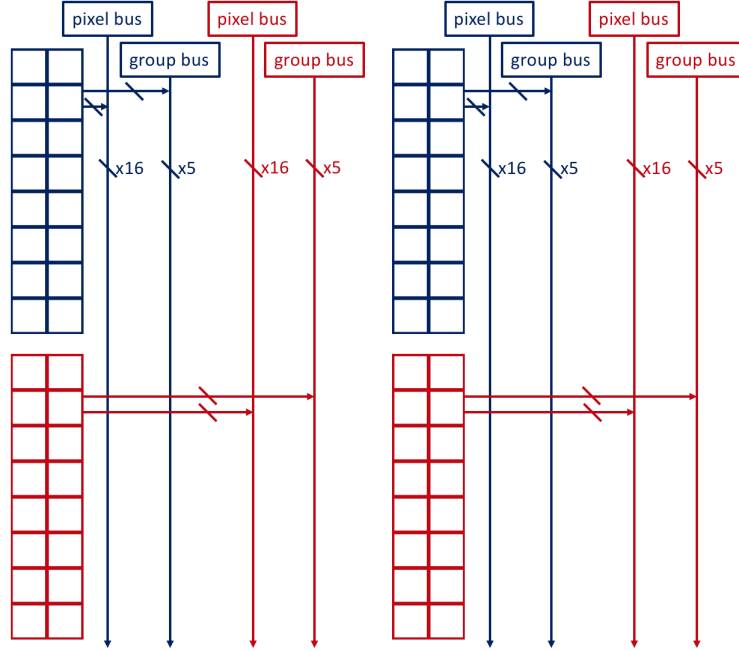


Figure 1.5: MALTA column drain readout architecture. Sketch of a portion of two double columns

on the assumption that the rate of hits in the matrix is relatively low, as lots of communication lines are shared among the pixels, but there is no synchronisation between them.

More specifically, this readout architecture divides the matrix in groups of two columns, usually referred to as double columns, that are repeated along all the matrix. It will thus be sufficient to explain the readout principle of one of these double columns.

The double column is divided vertically in 64 groups, each one containing 16 pixels. In the current implementation the columns are in fact an array of 512 pixels, but for a future implementation more groups will be used, as the pixel size will be reduced and their number increased. Each group is made of 16 bits from the two columns, it is thus a 2×8 matrix of pixels and all the 64 groups are simply piled up, one on top of the other, as shown in Figure 1.5. The groups are then divided in even and odd groups, that in the figure are marked in blue and red. They use two separate communication buses, one is shared among the even groups and the other among odd groups. Both work in the same way, with the only difference of the groups they belong to. Each data bus is made of two sub buses, a 5 bits bus (that may become 6 if more pixels within one column are needed) to transmit a static code for the group number, and another 16 bits bus to transmit the fired pixels within a group, clearly without any encoding of the position. The pixel information is

not encoded because a single crossing particle can easily activate more than one neighbouring pixel, hence more than one pixel should be able to send a signal at the same time and in fact with 16 bits it is possible to send the full pattern of activated pixels (that will be then encoded in the periphery of the chip). This is also the reason why the groups have been divided in even and odd, if in fact a particle crosses the detector between two groups then some pixels of both groups would probably be activated, but this is not a problem since even and odd groups do not share the transmission buses.

The maximum latency in transmitting the signal from the pixels to the periphery is around 5 ns for the farther pixels, as in this case the signal has to be transmitted along the full chip. The signal from the pixels is a pulse with a programmable width (0.5, 1 or 2 ns). The signal is sent once a charge threshold is surpassed, the time that it takes to surpass this threshold depends on the total generated charge. The peculiar fact in this sensor is that the collected charge is actually evaluated from this time information (that we will refer to as “time of arrival” or ToA), while more often a time over threshold (ToT) information is used to represent the charge information. The advantage of measuring the time of arrival instead of the time over threshold is that the latter requires a large pulse width to allow for sufficient resolution on the analog value, typically much longer than a single bunch crossing period. This creates mixing of information between different bunch crossings which can be handled, but at the cost of larger memories and a much increased circuit and data transmission complexity. In particular the measurements in [1] confirm that it is possible to acquire all the hits from the matrix within the 25 ns time window given by the separation between events in LHC, with the reasonable assumption that all the particles cross the detector at the same time. There are in fact several orders of magnitude separating the speed and times of the electronics and the speed and times involved in particles scattering.

It is important to consider that this scheme also introduces an inconvenience, as in fact the time of arrival at the bottom of the chip will depend not only on the collected charge but also on the distance of the involved pixel to the periphery. However this is not really a problem, as the charge information can be reconstructed using the position and time of arrival information together, the reconstruction could also be made directly in the periphery of the detector.

To acquire the data and associate them with a precise time information a fast synchronisation memory will be placed at the bottom of the columns. It has already been designed and simulated and it is ready to be implemented in the next test chip. Its purpose is to acquire the informations arriving asynchronously from the columns, associate them with a precise time information and store them in a short term memory. The short term memory is constituted of 4 rows, that (according to simulations based on expected data and on the memory circuit) should be enough to smear out statistical

fluctuations around the average number of hits. Once a signal arrives from the pixels and group buses (and flagged by an additional reference bit) it is directly stored in the memory with the relative time information. The memory controller then moves to the next row of the memory, in order to be able to store the next signal. The time is evaluated by a counter as the time between the beginning of the event and the acquisition of the signal. The counter works at a frequency of 640 MHz, that is 16 times the 40 MHz frequency of events, thus it will give a time granularity of around 1.56 ns.

Chapter 2

Trigger memory

In both MALTA and MonoPIX pixel sensors, the expected hit rate is so high that it wouldn't be possible to transmit all the data produced, as neither the number of lines nor the data bandwidth per line can be increased sufficiently. Therefore data corresponding to particular events have to be selected based on a fraction of the data from the ATLAS detector. Thus it is quite important to order the data and associate them to the correct event. In this case the event, called bunch crossing, corresponds to the intersection of two bunches of protons crossing the centre of the experiment, the resulting collisions will generate all the data inside the detectors that constitute the instrument. The selection is made on bunch crossings because each one of them is independent to the others and thus they can be treated as separate events.

The event selection is based on calorimeter and muon informations, whose measure, elaboration, evaluation and transmission require a time of the order of tens of microseconds. As this time corresponds to hundreds of bunch crossings, a memory is needed in the detectors to store all the data produced while waiting for the event selection. The expected average frequency of the first level of trigger signal is 4 MHz, effectively reducing the amount of data to be transmitted by a factor 10. Data is further selected and reduced using several other trigger levels.

2.1 Tasks and constraints

The memory we refer to is commonly called a trigger memory. The name is due to the fact that all the data are written in the memory, but an external trigger signal is used to select only a fraction of them for readout. In this case the trigger signal has a fixed delay of $12.5\ \mu\text{s}$ (500 bunch crossings) from when the event actually occurred. It is important to notice that in this case the latency of the trigger signal is fixed, this means that, if an event will have to be readout, its corresponding trigger signal will arrive

exactly 500 bunch crossings after the event occurred. Given the purpose of the memory it is possible to highlight its main characteristics and the tasks that the memory and the control circuitry have to accomplish:

Write First of all the control circuit has to write all the acquired data continuously in the memory and it has to be fast enough to account also for statistical fluctuations in the amount of data produced by the detector at each event. The data have to be written in a way that allows for distinction between separate events and grouping of data within the same event. Here it applies (in a different way) the same problem of bandwidth mentioned above, the memory will in fact have a maximum writing frequency that depends on its physical characteristics. This is a non negligible bottleneck for the data and in fact it will limit the maximum acceptable data rate (and it will make it necessary to design more parallel memories in the same detector).

Store The memory has to store data for a period of time at least equal to the latency of the trigger signal. This (together with the data rate and format) gives the size of the memory. The memory shape itself will depend on the format in which data are stored.

Read and trigger The reading circuitry has the task to find and readout all the triggered events. This means that data should be stored in a way that makes it easier to retrieve the data based on the event they belong to. The trigger controller instead has to match the trigger signal with the correct event, occurred exactly $12.5 \mu\text{s}$ before.

2.2 Meaning and content of the data

Having faced the problem from a general perspective it is now appropriate to consider the data involved in this particular case.

The inner tracker purpose is to reconstruct the tracks of the particles, in order to succeed it needs to know where and when the particles have passed. This is why the detector is a pixel detector and the most important informations it acquires are the position and the time. At the moment the detector is made of a matrix of 512×512 pixels, but a reduction of the size of the pixel (and hence a further increase in the number of pixels) is also being considered.

For each event the hits are very sparse, since the particles will cross only a small fraction of the total number of pixels. For this reason there is no need to store a full image of the pixel matrix and it will be enough to store the position of the pixels where a particle crossing has occurred. In general an information (that in this case is the position) can be encoded with a well defined minimal number of bits, this number is the lower integer greater than the base 2 logarithm of the total number of possible data arrangements

considered for storing. In this case the possible data considered for storing will of course be the possible positions inside the pixel's matrix.

$$n_{bit} = \lceil \log_2(N) \rceil, \quad (2.1)$$

Considering the current matrix size of 512×512 pixels, 18 bits are needed to store the full position of a certain hit. We have in fact 2^9 columns and 2^9 rows, it is thus possible to store the column address (or row address) with a digital number of 9 bits, without any repetition or redundancy

As shown in the previous chapter the actual distribution of signals from the matrix is quite complex, so it is necessary to arrange the data properly and the position stored is not a simple representation of the position address based on columns and rows, but depends on how data are grouped and it will be reconstructed offline. The only important fact to consider is that the position must be reconstructed univocally, moreover if the reconstruction is two-way univocal then the amount of information to store is the same, even if the shape of the data is different. What in fact matters is the number of pixels, if it is a power of two the position can be encoded in the most efficient way, this is true for the columns, the rows, but also for other forms of grouping. More in general we could state that what matters is the amount of information, that is conserved across different encodings if they are all maximising the efficiency.

Indeed in this detector within half of a double column there are 32 groups and the number of pixels per group is 16 (as described in section 1.4.2). This means that the group can be encoded with 5 bits and the pixel position inside the group can be encoded with 4 bits. This makes 9 bits in total, exactly the same to encode the pixels of a full column, if we consider that two halves of double columns cover exactly two columns then it is clear that the same number of bits is used even if the grouping is different.

In practice the group information that arrives at the bottom of the matrix is already encoded. The pixel position inside the group is instead not yet encoded. This is because a hit inside the group may easily activate more than one pixel and also because it is not possible [or convenient/easy] to encode the pixel position inside the matrix, as a consequence it has to be encoded in the periphery, once the signal arrives from the matrix. As more than one pixel signal may arrive at the same time, in general it wouldn't be possible to directly encode the 16 bits in one code of 4 bits, it is thus necessary to translate the 16 bits in a sufficient number of separate codes, one per each fired pixel.

The other information to store with the position is the digital value corresponding to the collected charge. As shown in the previous chapter, in section 1.4.2, the value of the collected charge corresponds to the time at which the pixel information arrives at the bottom of the matrix with respect to the beginning of the event. For this reason there is a fast clock

with a counter that evaluates the time at which the data arrives (the “time of arrival” or ToA), this number is then stored as the digitalisation of the collected charge together with the corresponding pixel position. We can then see that data arriving at the same time actually have the same digital charge value to store, hence it will be possible to decode the 16 bits of pixel’s position within the group by just duplicating (or triplicating or in general copying) the other data and encoding each of the fired pixels separately. Due to the maximum frequency of 640 MHz at which is possible to acquire these data the time of arrival is encoded in 4 bits. The sampling frequency is in fact exactly 16 times the 40 MHz frequency of the events.

The charge information is actually needed only to have a finer positional information. The required final resolution is indeed higher than that currently allowed by the pixels’ spacing, thus the charge information turns out to be helpful in overcoming this deficiency by allowing for an interpolation to be performed. More specifically it could be possible to better reconstruct the positional information by relying on the fact that a hit will likely activate more than one pixel and then interpolating the charge and position of the activated neighbouring pixels.

Some improvements of the pixel matrix are currently being investigated, they also include a reduction of the pixel size that could make the charge information unnecessary. But, even if this improvement were implemented, it would not change substantially the handling of data inside the detector. In fact the position reconstruction would in any case be performed outside the detector, thus, while inside the detector, the charge information will be treated simply as an appendix of the pixel position of a hit. Then the two informations will always be considered together as a whole datum relative to the hit and this would happen both with or without the charge information.

2.3 Estimate of the memory size

An estimation of the memory size is reported in this section. It is based on the simple idea of storing each hit with an identifying tag that assigns the hit to the corresponding event.

The estimation is carried out by considering the memory cell size, the data format and, of course, the total amount of data that have to be stored, that is obtained from the hit[s] rate and the trigger latency. With the data is included also the identifier used to univocally assign each hit to the corresponding event.

The identifier simply consists in a progressive number, the number 0 is assigned to the first event after setup, thereafter a constantly increasing number is progressively assigned to each new event. This same number is then assigned to all the data belonging to that event. We will refer to this number as bunch crossing identifier or, briefly, BCID.

Table 2.1: Fluence across layers

Layer	Fluence [cm ⁻² BC ⁻¹]	Fluence [chip ⁻¹ BC ⁻¹]	Hit rate [chip ⁻¹ BC ⁻¹]	Hit rate [chip ⁻¹ MHz]
0 [non-tilted]	18.06	68.6	102.9	4,117.7
0 [tilted]	20.70	78.7	118.0	4,719.6
1 [non-tilted]	4.60	17.5	26.2	1,048.8
1 [tilted]	9.28	35.3	52.9	2,115.8
2	2.06	7.8	11.7	469.7
3	1.22	4.6	7.0	278.2
4	0.80	3.0	4.6	182.4

This number serves for discriminating between all the data acquired in the last 500 bunch crossings, it is in fact possible to discard all the older data because the trigger signal will always arrive with a constant latency of 12.5 μ s that exactly corresponds to 500 bunch crossings. For doing so at least 500 distinct identifiers are needed, that means all numbers between 0 and 500. As binary numbers are used, the least number of bits needed is 9, this will in fact permit to encode 512 different identifiers. Of course, as data have to be acquired continuously, the progressive counting will have to begin again from 0 once the highest number is reached, however this would not create any ambiguity because there will still be at least 500 different identifiers for the last 500 events.

The data for each hit therefore consists of the identifier, the pixel position and the charge information (or time of arrival). As the former consists of 9 bits, the second of 18 bits and the latter is encoded in 4 bits, then the total number of bits per each hit is 31.

For the calculations I relied on some hit rates estimation that were provided as results of simulations, based on a proposed layout for the detectors and on the expected charged particles flow. The results are listed in Table 2.1. The layer refers to the position inside the inner tracker, the corresponding number is progressing from the inner layers to the outers. The labels ‘tilted’ and ‘non-tilted’ correspond to the orientation of the detectors in the barrel, ‘non-tilted’ refers to the detectors in the middle of the barrel, aligned with the barrel walls, while ‘tilted’ refers to the detectors closer to the edges of the barrel that are rotated in order to be orthogonal to the particles coming from the interaction point.

The target for both MALTA and MonoPIX is the 4th layer, nevertheless I evaluated the memory size also for the higher rates reached in the inner layers. The goal was to check if it is possible to extend the detector’s target and make it usable in a wider range of situations, to better understand the

limitations of the approach.

However, it is important to note that writing serially all these data is not possible. In fact the data in the 4th column (where the hit rate is expressed in MHz rather than bunch crossings) hint that very high frequencies would be required for writing. Those frequencies are far from being achievable, a 1 GHz frequency is already very hard to reach and it's possible only for small components and also clock distribution is not easy. Then, the process of writing in particular is not a fast one, the maximum frequency reached for a very small memory in this same project is 640 MHz. Thus, this bandwidth limitation can only be addressed by segmenting the matrix and operating several memories in parallel.

For each hit rate the average amount of data in bits is then simply given by the number of bits per hit times the hit[s] rate times the latency and both latency time and rate can be expressed in terms of microseconds or equally in terms of number of bunch crossings.

$$N_{bits} = \frac{\# \text{of hits}}{\text{BC}} \cdot (\# \text{of BC}) \cdot \frac{\# \text{of bits}}{\text{hit}} \quad (2.2)$$

From this this simple estimation of the average is possible to obtain the data reported in Table 2.2. In the list also an estimation of the area is given, it is obtained by multiplying the number of bits with the area of the elementary RAM cell available in TowerJazz 180 nm, that is $10.52 \mu\text{m}^2$. It is important to notice that in this estimation the area used by circuitry and memory periphery is not considered, as it is just an estimation based on data. Also an estimation of the memory physical depth is given, this will in fact give a more useful hint on the space used by the memory in the periphery at the bottom of the chip. The width is in fact fixed and the actual parameter to minimise is the depth, indeed in order to fit the standard chip frame the memory depth has to be kept largely below 1 mm. In general the smaller is the memory the better it is, because, since also other components are currently being designed, the constraints are still not completely defined, and a smaller memory will also give lower power consumption.

The depth is simply obtained by dividing the memory area by the full chip width, evaluated as the number of pixels in a row (512) times the $36.4 \mu\text{m}$ pixel pitch (that is the distance at which the pixels are repeated).

For properly evaluating the memory size I couldn't just use the average value, because the hit rate is actually variable across the events. The hit rate is in fact an average rate at which particles cross the detector, but what is needed is the proper distribution of data. Thus, to perform an accurate analysis and obtain the data distribution I had to use Poisson statistics.

Indeed, as the data produced in a collision are fully random, there will always be the chance for the memory to be filled, regardless of its size. This is why a proper statistical study turns out to be useful, it will in fact give proper awareness of data loss. In fact, since it's not possible to comprehend

Table 2.2: Average data produced

Layer	Average hits #	Average data [bits]	Average area [mm ²]	Average depth [μm]
0 [non-tilted]	51,450	1,594,950	16.77	899.607
0 [tilted]	59,000	1,829,000	19.23	1031.619
1 [non-tilted]	13,100	406,100	4.27	229.054
1 [tilted]	26,450	819,950	8.62	462.480
2	5,850	181,350	1.91	102.288
3	3,500	108,500	1.14	61.198
4	2,300	71,300	0.75	40.216

all the possible situations, some data loss is accepted and the goal is to reach a target percentage efficiency. Thus, this study aims at finding the memory size for which this efficiency is achieved.

Given a rate and a period of time, the Poisson distribution is the probability distribution of the events occurring during that period of time. More in detail it is a discrete probability distribution that assigns to each integer number the probability for that number of events to occur, this with the assumption that events are uncorrelated, that is that the events happen with a constant rate and probability is independent to any previous event.

The Poisson distribution is given by:

$$P(k) = \exp(-\lambda) \cdot \frac{\lambda^k}{k!} \quad \text{where} \quad \lambda = r \cdot t \quad (2.3)$$

Where k is the number of events and λ is the expected number of occurrences per time unit (that is the average number of occurrences during the time step). The average, λ , is obtained as the rate r times the time step. The average number of events is already known, but the exact distribution gives also information about the deviation of the total amount of data from the mean value. The variance of a Poissonian distribution is simply λ , hence the standard deviation is its square root.

The required efficiency for this detector is 99.7%, thus to fulfil this requirement I added to the average amount of data a safety factor of 3 times the standard deviation, so that the probability of overflow will remain under 0.3%. Given this safety factor it is possible to calculate the real required memory size in bits and then translate it again in area and depth. Data are shown in Table 2.3 and a plot is given in Figure 2.1, where in blue is written the average and in red the safety factor.

The first thing to note is that the memory depth widely exceeds the available space for the periphery. And together with size there is the problem relative to bandwidth.

Table 2.3: Required memory size

Layer	Required number of hits #	Required storage [bits]	Required area [mm ²]	Required depth [μ m]
0 [non-tilted]	52,130	1,616,045	16.99	911.505
0 [tilted]	59,729	1,851,590	19.46	1044.361
1 [non-tilted]	13,443	416,744	4.38	235.058
1 [tilted]	26,938	835,075	8.78	471.011
2	6,079	188,463	1.98	106.300
3	3,677	114,002	1.20	64.301
4	2,444	75,760	0.80	42.731

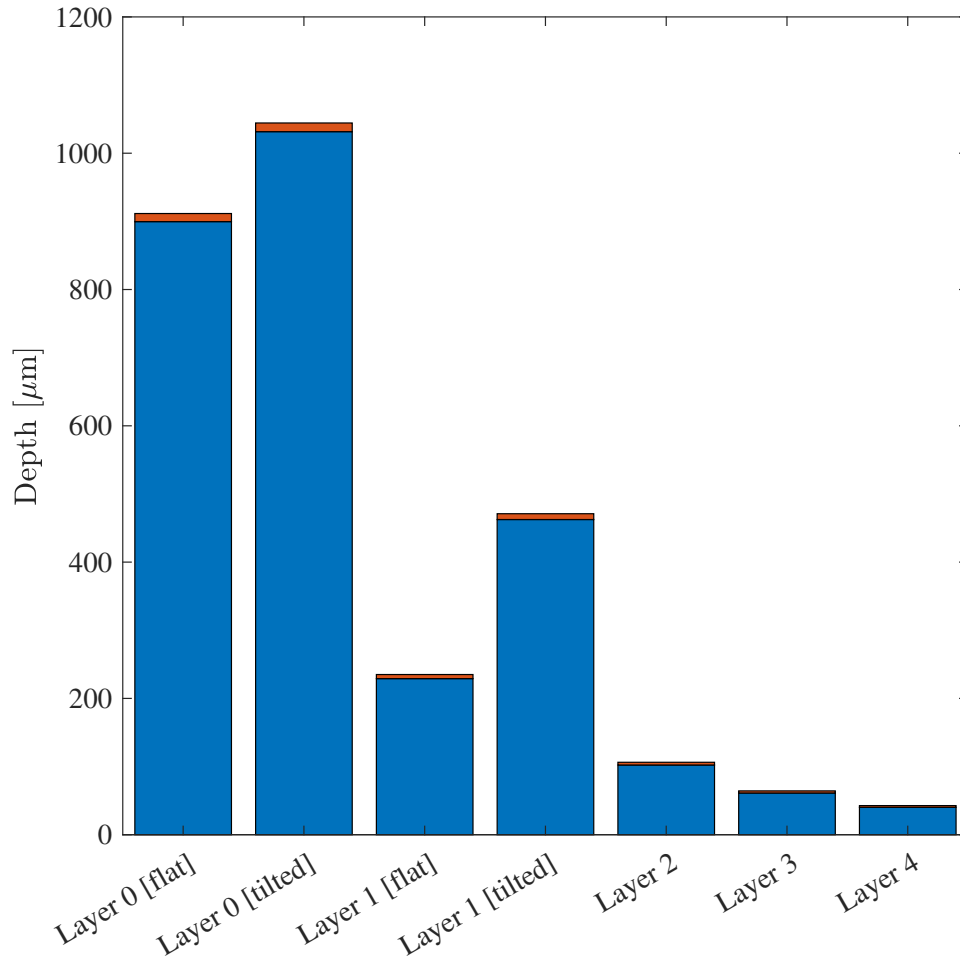


Figure 2.1: Memory depth for the different layers

Something that can be useful to address both these issues is dividing the memory in more submemories. In fact this will clearly reduce the data rate per memory (thus reducing the frequency required for writing) and will also allow to reduce the address size. If in fact each memory is assigned to a precise subset of pixels of the matrix, then it will have less pixels to address. The additional position information will be added later during readout, according to the submemory from which data is being sent out.

Clearly, by creating more memories rather than only one, the amount of control circuitry may increase, as lot of components will be repeated in each memory. The impact of control circuitry on size is not easy to foresee, as it will depend on the actual implementation of the readout and it will not change trivially with the chosen grouping. For this reason (and because the area used by RAM cells would in any case be the major part) the following discussion is only based on data and a more precise estimation is possible only after the implementation of the control circuitry.

Considering that the readout architecture is based on columns, the most natural way of dividing the matrix is in groups of columns. For the same reason discussed above in section 2.2, the best way of grouping columns is in numbers that are exact powers of 2. Thus I calculated the depth of the memory considering all the possible groupings of columns between one and 512, that is the total number of columns, so that it is possible to have a complete overview of the trend of the memory depth. I calculated the rate of each group assuming that it depends only on area, hence it is proportional to the number of pixels inside the group, the proportionality is obtained by the hit rate given for the full chip. The physical memory depth is evaluated as the area of the submemory divided by the width of the group, that is the width of one column times the number of columns grouped together. The width of one column is, like the width of one pixel, $36.4 \mu\text{m}$.

The results are shown in Figure 2.2 and in Table 2.4. The lines correspond to the average value, the error bars instead equal the safety factor of 3σ and thus give the final size of the memory. The plot clearly shows that the depth is reduced by grouping less columns, but also hints that this advantage is weakened by the increase of the safety factor with the decrease of the group size (as highlighted by the error bars). This happens because (as stated above) the standard deviation is the square root of the expected number of occurrences, that is the hit rate times the latency. For this reason (and agreeing with common sense) the safety factor is relatively larger for smaller hit rates.

Actually at some point the trend is even inverted and the increase of the safety factor prevails over the decrease of the average amount of data, thus setting a minimum to the memory depth. The position of the minimum depth relatively to the group size vary with the layer, as for the outer layers the hit rate is lower and thus the contribution of the safety factor is more relevant.

Table 2.4: Required memory size

Layer	Required number of hits #	Required storage [bits]	Required area [mm ²]	Required depth [μm]
0 [non-tilted]	52,130	1,616,045	16.99	911.505
0 [tilted]	59,729	1,851,590	19.46	1044.361
1 [non-tilted]	13,443	416,744	4.38	235.058
1 [tilted]	26,938	835,075	8.78	471.011
2	6,079	188,463	1.98	106.300
3	3,677	114,002	1.20	64.301
4	2,444	75,760	0.80	42.731

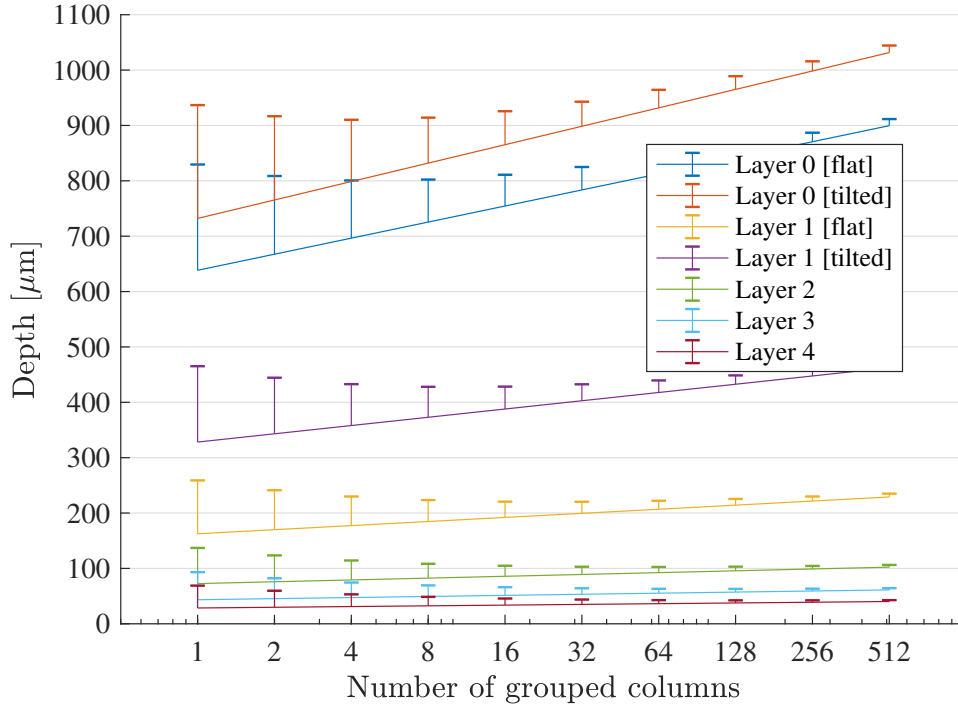


Figure 2.2: Memory depth for different layers and groups

The grouping will have to be the same for all the layers because the aim is to produce only one detector for all layers, hence the worst case has to be considered (that is the 0^{th} tilted layer). For this layer the trade off between address reduction and safety factor increase is found with groups of 4 columns, that require a total memory height of $910.3 \mu\text{m}$. According to this simple analysis it is possible even for the inner layers to design a memory that would fit in an acceptable area at the periphery. Nonetheless, a reduction of the memory size could allow for a better management of the periphery area and a reduction of power consumption.

Therefore, the main purpose of this Thesis work was to find a compression algorithm capable of reducing the amount of data to store while keeping power consumption low. The proposed solution is deeply discussed in the next chapter.

Chapter 3

Compression algorithm

In this chapter the mechanism adopted for data compression is explained.

The aim of the proposed solution is to reduce the impact on memory size given by the bunch crossing identifier, as it takes 9 bits per each hit, that is a relevant part of the whole memory. Thus, this compression algorithm aims at finding an alternative solution for ordering and retrieving the data in terms of bunch crossing.

Also other ideas were considered, but focusing on the identifier appeared to be more effective. The individual hit data are in fact already maximally compressed and it was also quite hard to find spatial correlations between them because they are too sparse in the matrix. The only actual correlation is given by the clusters of pixels fired by the same hit, but their average size (estimated by simulations) does not justify a compression mechanism based on clustering, clusters are in fact too small to make such a compression efficient.

3.1 Redundancies in the data

The observation that led to this compression mechanism is that the same bunch crossing identifier is usually stored several times, as in most of the bunch crossings there will be actually more than one hit.

More in general any compression algorithm is based on the idea of finding redundancies in the original data in order to avoid the storage (or transmission) of the same information several times. In this case the bunch crossing identifier is very often a redundant information, thus it is possible to store the same information by using less bits.

First it is better to highlight that the purpose of this mechanism is to reduce the memory size required for the inner layer, as that is the worst case. Hence it is intended to work efficiently in a high rate environment, but it will not necessarily be efficient in other cases. Indeed another important fact about compression is that it depends on statistical characteristics of data,

that correspond to the expected redundancy. In fact, as data are not known in advance, a compression algorithm will have to store with less bits the informations that are expected to occur more often. In other words (since the redundancy is actually a correlation between data) the algorithm will have to rely on an expected correlation across data, if then the correlation is different from the one expected the given compression mechanism will result to be inefficient.

For example, the extreme case of a perfectly white noise is impossible to compress without losing information because all the bits are completely random, thus there is no statistical information on which it would be possible to rely, this also means that there is no correlation between bits. In more theoretical terms the white noise already maximises Shannon's entropy. The same consideration applies to compression algorithm, in a maximally efficient algorithm the final compressed data will maximise Shannon's entropy.

Actually, even though this is the underling theory, we approached the problem from a more practical point of view, as there were several implementation constraints. The first is that the algorithm cannot be too complex, it has in fact to be implemented in hardware without using too much area and without consuming too much power. Then it is also important for this algorithm to allow for an easy retrieval of data according to the bunch crossing they belong to.

3.2 The compressed format

As stated before, the compression algorithm applies to the bunch crossing identifier. The single hit information is in fact already maximally compressed and there is no way to find a correlation between hits, as their position and charge is completely random and they are too sparse in the matrix (in fact, even considering the worst case, there are, on average, only 118 hits per bunch crossing out of 512×512 pixels).

There are instead correlations between the bunch crossing identifiers of the hits, it is in fact known that:

- The number assigned to each bunch crossing is constantly increasing.
- The same number may often label more than one hit.

Thus, if data are properly ordered, the bunch crossing identifier of each hit is always greater or equal to that of the previously stored hit. If then the hit rate is high enough the bunch crossing identifier of each hit will most of the times be the same as the previous hit or the next.

The solution adopted for compressing is thus to avoid the storage of the full bunch crossing identifier and tag the data with three possible labels:

The first label refers to the case in which the corresponding hit belongs to the next bunch crossing, right after the previously stored hit.

The second label tags all the hits, after the first, belonging to the same bunch crossing. It is thus added to all the hits that belong to the same bunch crossing of the previous hit.

The last label is needed in order to take into account the chance of a bunch crossing in which no hit occurs. In this case in place of the data is stored the bunch crossing identifier corresponding to the next bunch crossing in which some hits occurred.

Thus, as there are three possible labels (or headers), only two additional bits are needed to label each hit, instead of the 9 bits that would be required to store the full bunch crossing identifier.

There is clearly one possible state that is wasted in this case, as with 2 bits it is possible to encode 4 states. The remaining code could be used in the case when there is a single bunch crossing without hits. In particular it could be used as header of the next set of data, in this way it could be possible to distinguish it from the header used for consecutive bunch crossing and at the same time there would be no need to store the bunch crossing identifier in between. This could allow for even better compression, in particular if we consider that a good number of the empty bunch crossings are isolated. This solution has not yet been implemented in the final circuit, but could be easily integrated as an improvement in the present structure of the code. For this reason I will show in the next section the memory estimation both with and without the use of the additional code.

The idea for the header encoding is shown in Figure 3.1. In yellow is highlighted the redundancy given by the repetition of the bunch crossing identifier, in red is instead highlighted the redundancy given by the constantly increasing and consecutive bunch crossings. It is in fact possible to think of this latter redundancy as the repetition of most of the bits of the identifying number across separate events.

Actually this mechanism relies on concatenation of data and the information about the bunch crossing relies on all the previously stored labels. This consideration addresses two important issues. The first is that at some point a synchronisation is needed, because while writing the data the tags will refer to particular bunch crossings that will have to be reconstructed during readout. For this reason the third label turns out to be useful not only in the case of empty bunch crossings, but also for setup and synchronisation between writing and readout. The other important issue is strictly correlated with the first and it is the fact that this system is very fragile, in particular considering the fact that it has to work in a very tough radiation environment because of radiations. There is in fact a non negligible probability to alter the data stored in the memory, this phenomenon is usually

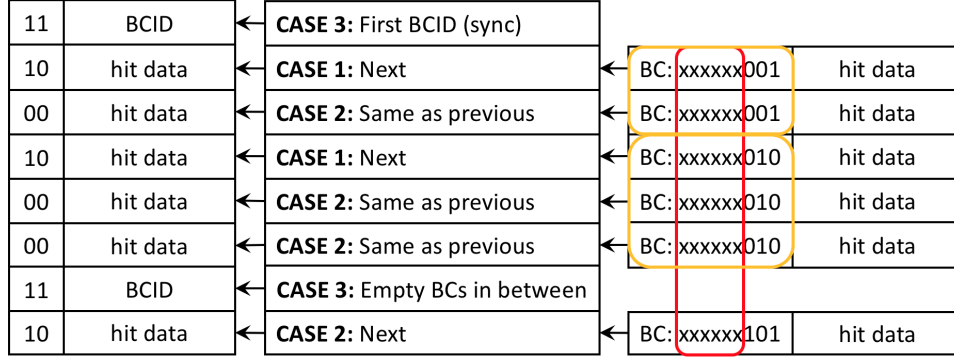


Figure 3.1: Memory depth for different layers and groups

called Single Event Upset (SEU) and refers to the eventuality of a bit flip in a memory cell due to the generation of holes and electrons by a passing charge. In case of bit flip (SEU) in one of the header bits or in a bunch crossing identifier, the synchronisation would be completely lost. It could only be restored once the next full bunch crossing identifier is read out, but all the data stored between the wrong identifier (or header) and the next correct identifier would be lost. Even worse, while not synchronised the memory would assign the wrong data to a certain triggered bunch crossing and then there would be no way to distinguish between correct and wrong data.

It is possible to find details of SEU cross section for TowerJazz 180nm RAM cells in [7]. The reported cross sections have been evaluated with proton beams of energy ranging from 32.2 MeV to 320 MeV, and they are all under $12 \cdot 10^{-14} \text{ cm}^2 \text{ bit}^{-1}$. It is possible to calculate the bit flip probability by the product of particle fluence, trigger latency and SEU cross section. The result is the probability for a single bit to flip during the storage in the trigger memory. For the highest fluence of $20.7 \text{ cm}^{-2} \text{ BC}^{-1}$ a probability lower than $1.3 \cdot 10^{-9}$ is obtained, that, despite being small, addresses the need for a more robust solution. This is because the event of an error will desynchronise the readout, thus generating lots of errors from a single one.

The fact that the compressed data format is less reliable is an intrinsic problem. Also reliability depends in fact on redundancy, as in case of error the redundant information may be used to reconstruct the original information. Thus the reduction of redundancy is strictly connected with the weakness of this scheme.

Anyway it is also true that not all redundancy is helpful in increasing robustness with respect to SEU. Actually a good deal of theoretical works and studies had been carried out on this topic, as reliability in information technology was a frequent issue in early the times of elaborators and some extra robustness is still necessary in lots of critical applications. These

studies aimed at adding some targeted redundancy for error correction and led to the elaboration of the so called perfect codes, that are the codes that minimise the ratio between redundancy bits and total number of bits.

Therefore, as a solution to strengthen the data storage, Hamming error correcting code was adopted. A brief description of Hamming code and its implementation in this particular case is given in the next section.

3.3 Hamming encoding

Hamming codes are a set of parity check error correcting codes, originally conceived by Richard Hamming (see ref. [9] for further details). They are all based on the same idea and change only for the size of the encoded word and the number of required parity bits.

Hamming codes in particular allow for single error correction within a word of bits, this means that, if an error occurs in the word, it is then possible to reconstruct which one is the wrong bit and correct it. With the addition of an overall parity bit it is also possible to detect double errors (that would otherwise pass unnoticed as a single error that would be corrected in the wrong way). The power of Hamming codes lies in this ability to correct the errors and to join more bits together in a word in such a way that the longer is the word the more efficient is the code.

In Hamming encoding the parity bits are used to identify the bit that was affected by an error. For this purpose each parity bit is defined as the parity of a different pattern of the data bits. The patterns are arranged in such a way that, after the parity check, in the event of an error the combination of wrong parity bits can give a unique reconstruction of which bit is in error, this includes the parity bits themselves. Indeed if only one parity bit is incorrect then is the parity bit itself to be in error, otherwise, if a data bit is subject to an error, there will be two or more parity bits that will not pass the parity check.

With n parity bits it is possible to have n different patterns and thus to univocally identify an error in $2^n - 1$ bits. This happens because with n bits it is possible to encode 2^n possible situations, that, together with all the possible errors, comprehend also the eventuality of having no errors. As it is possible to identify an error in a word of $2^n - 1$ bits and this same word includes also the parity bits, then there will be $2^n - 1 - n$ total data bits. This is why there exist maximally efficient codes that are the ones based on words of $2^n - 1$ bits, the other codes are called truncated codes, as they are obtained by the previous by just eliminating the appropriate number of data bits.

In order to introduce single error correcting double error detecting codes (SECDED codes) it is first better to introduce the notion of Hamming distance.

The Hamming distance evaluates the strength of an error correcting code in terms of its ability to detect and/or correct single or multiple errors. It is defined as the minimal number of different bits between allowed codes, in other words if the Hamming distance of a certain code (being it a Hamming code or not) is d , then all the allowed codes will have at least d different bits. This notion turns out to be very important because error detection is based on the fact that only some words are allowed and error correction is instead based on distance as it will assign eventual forbidden codes to the closet allowed code, by assuming that the number of errors was the minimum possible. If the distance is odd then all the codes (allowed or not) will have only one closest allowed code, thus error correction will always be possible. If instead the distance is even there will be forbidden codes equally distant to at least two allowed codes, for these codes error correction will be impossible and there will be only detection. More in general error correction can be left out in favour of detection of a greater number of errors.

Hamming codes have a minimum distance of 3 and thus will allow single error correction only, or double error detection if correction is not attempted. This is because with a minimum distance of 3 there will be at least two forbidden codes between all the correct codes. Then, if a forbidden code is received, it will be assigned to the closest correct code. This will result in a correction if only one error have occurred, but if two or more errors have occurred it means that the received code has gone too far from the original and will therefore be closer to another code which will be the (wrong) result of the correction. If error correction is not attempted, then an error will be detected if any wrong code is received, thus also double error will be detected.

By increasing the distance to 4 it is possible to correct single errors and detect double errors (or leave correction and detect up to 3 errors). Single Error Correcting Double Error Detecting codes (usually referred to as SECDED codes) with respect to simple Hamming codes have the great advantage of detecting double errors, and what makes them even more advantageous is the fact that SECDED codes can be achieved from a simple Hamming code at the price of adding just one more bit. The added bit is just an overall parity bit, thus in case of double error it will be evaluated as correct (both if the bit itself is involved or not), but from the other parity bits this situation will be evaluated as a single error. Thus the double error detection will be given by the intersection of a single error detected by the classic Hamming code with a correct overall parity bit. In case of single error instead the correction will be performed, but with the difference that the overall parity bit will be incorrect (unless it is the overall parity bit itself to be incorrect, and in this case no correction will be performed) and this will give the difference with the situation in which there are two errors.

In the considered memory Hamming encoding has to be implemented for the header and the bunch crossing identifier. The encoded bunch crossing

identifier can easily fit in the space used for hits data. A length of 10 bits was in fact used to identify the bunch crossing in order to keep a safety factor and avoid overflows. The corresponding Hamming code is thus 15 bits long, in order to allow both single error correction and double error detection. This means that it can actually fit in the 17 bits required for storing each hit with a grouping of 16 columns, and that could also fit the 16 or 15 bits required for grouping respectively 8 or 4 columns. The header instead will need 6 bits to be encoded with SECDED, Hamming codes are indeed very inefficient for small groups of bits.

The memory size estimation with the compression mechanism (with both encoding or not) are reported in the next section.

3.4 Memory size estimation with compression

In order to evaluate the memory size we have chosen in this case to simulate the data generated in the chip. The number of data words that will contain a bunch crossing identifier will in fact depend not only on the number of bunch crossings with no hits, but also on how often and how many of them occur consecutively. Thus, the most flexible way of obtaining the memory size was that of simulating data and extract the memory size from the simulated data with the compression algorithm.

The following plots and tables report the memory size evaluation for different situations. In particular we will show the simulations obtained with and without Hamming encoding, and what could change by using also the fourth state for the header.

The first plot, in Figure 3.2, refers to the memory estimation without Hamming encoding for the header, the relative data from 1 to 64 columns are shown in Table 3.1. This estimation does not correspond to the final implementation of the memory, but is given to better appreciate the performance of the compression algorithm. In this case the header consists of only two bits, thus it is possible to achieve a good level of compression, as it substitutes the 9 bits used for the full bunch crossing identifier. We find in fact that the depth required for the memory would be in this case $598.32 \mu\text{m}$ for a grouping of 16 columns.

Unfortunately it is not possible to actually achieve this level of compression, in fact due to the high level of radiations it is necessary to add Hamming encoding. This cancels a great part of the compression, as actually the header turns out to be bigger than half of the full bunch crossing stored in the non-compressed format. To be more precise the bunch crossing identifier requires 9 bits and the encoded header requires 6 bits, if we also consider the data bit we see that for the uncompressed format $17 + 9 = 26$ bits are required, for the compressed format instead $17 + 6 = 23$ bits are required. This already limits the maximal achievable compression, that will

Table 3.1: Depth required for different columns groupings without Hamming encoding

Layer	Columns grouping						
	1	2	4	8	16	32	64
0 [non-tilted]	966.05	849.20	719.27	623.76	598.32	611.12	632.43
0 [tilted]	1066.15	930.81	789.42	697.10	679.73	698.98	723.44
1 [non-tilted]	341.67	308.95	277.31	240.99	203.37	175.92	168.07
1 [tilted]	582.41	525.24	458.73	387.91	337.25	322.98	329.80
2	186.62	165.72	150.59	134.73	117.73	99.29	84.62
3	128.84	113.89	101.18	91.76	81.59	70.06	58.70
4	98.03	83.45	73.81	65.91	59.52	52.20	44.39

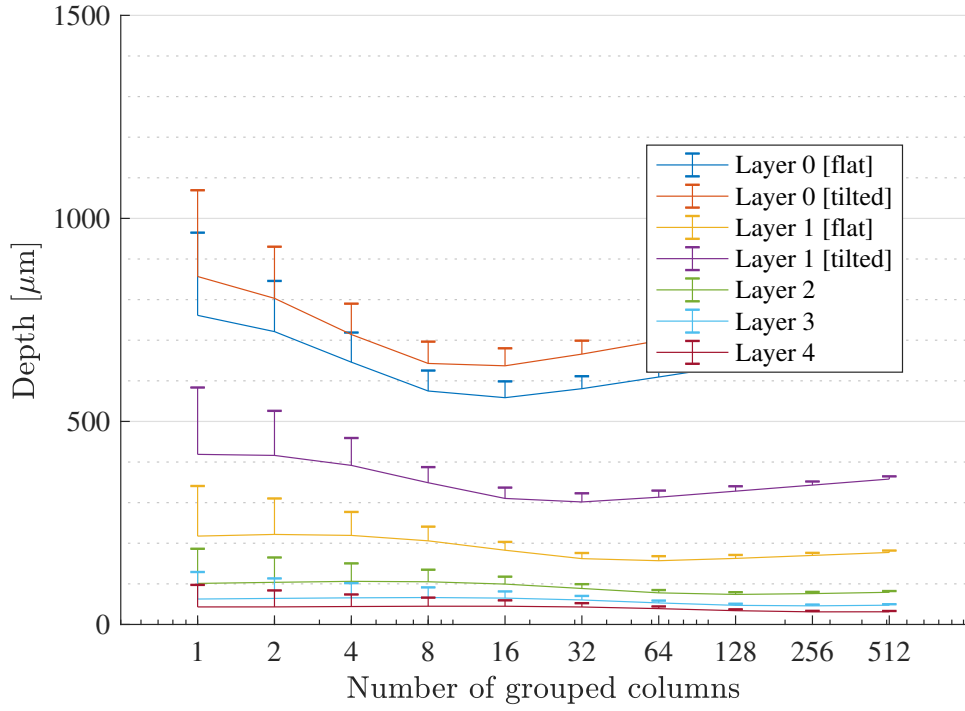


Figure 3.2: Memory depth without hamming encoding

Table 3.2: Depth required for different columns groupings with Hamming encoding

Layer	Columns grouping						
	1	2	4	8	16	32	64
0 [non-tilted]	1223.67	1061.49	888.51	762.38	724.28	733.34	752.89
0 [tilted]	1350.45	1163.51	975.17	852.01	822.83	838.78	861.23
1 [non-tilted]	432.78	386.19	342.56	294.55	246.18	211.11	200.09
1 [tilted]	737.72	656.56	566.67	474.12	408.26	387.58	392.62
2	236.39	207.16	186.02	164.67	142.51	119.14	100.73
3	163.19	142.36	124.98	112.15	98.77	84.08	69.89
4	124.18	104.32	91.18	80.55	72.05	62.64	52.84

Table 3.3: Depth required for different columns groupings exploiting the 4th state of the header

Layer	Columns grouping						
	1	2	4	8	16	32	64
0 [non-tilted]	1114.99	927.44	776.42	714.98	718.02	733.22	752.89
0 [tilted]	1220.46	1011.04	860.73	811.60	819.01	838.73	861.23
1 [non-tilted]	419.73	365.30	312.03	257.38	215.37	198.48	198.51
1 [tilted]	697.80	597.67	495.13	414.81	384.09	384.63	392.56
2	232.89	201.62	176.92	150.99	125.06	103.64	93.42
3	161.67	140.01	121.07	105.71	89.61	73.20	61.45
4	123.47	103.14	89.18	77.35	67.05	55.81	45.72

be a factor of $23/26 \approx 0.89$, that is the ratio between the bits used to store a full hit in the two different cases. Data relative to the scheme adopting Hamming encoding are shown in the plot in Figure 3.3 and in Table 3.2.

Nonetheless by using also the last available state for the header, as suggested in the previous section, it is still possible to achieve more compression. As we can see in the plot in Figure 3.4 (and in Table 3.3) the advantage is not great, but comes at no expense for the memory and can be easily implemented in the compression algorithm.

During this thesis work I also made some estimations on the power consumptions of the memory, based on the number of writing and reading operations. This is not to be considered a complete estimation of the memory power consumptions, as there are lots of factors involved and a physical simulation of the memory and the data should be performed, but it can give an idea of the trend, since the grater part of the consumptions is from

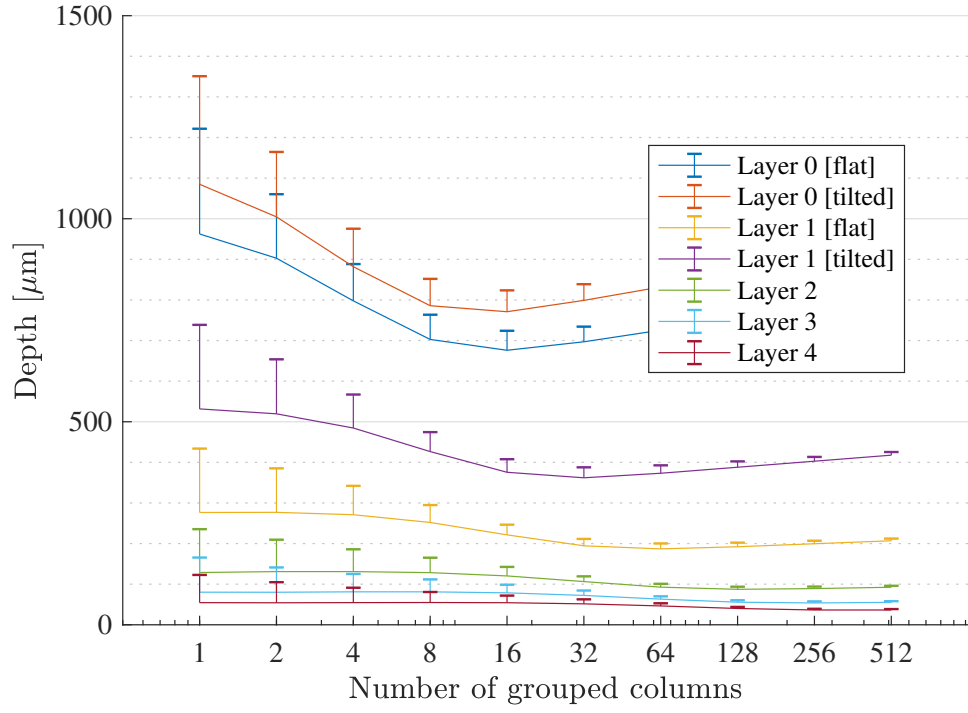


Figure 3.3: Memory with Hamming encoding

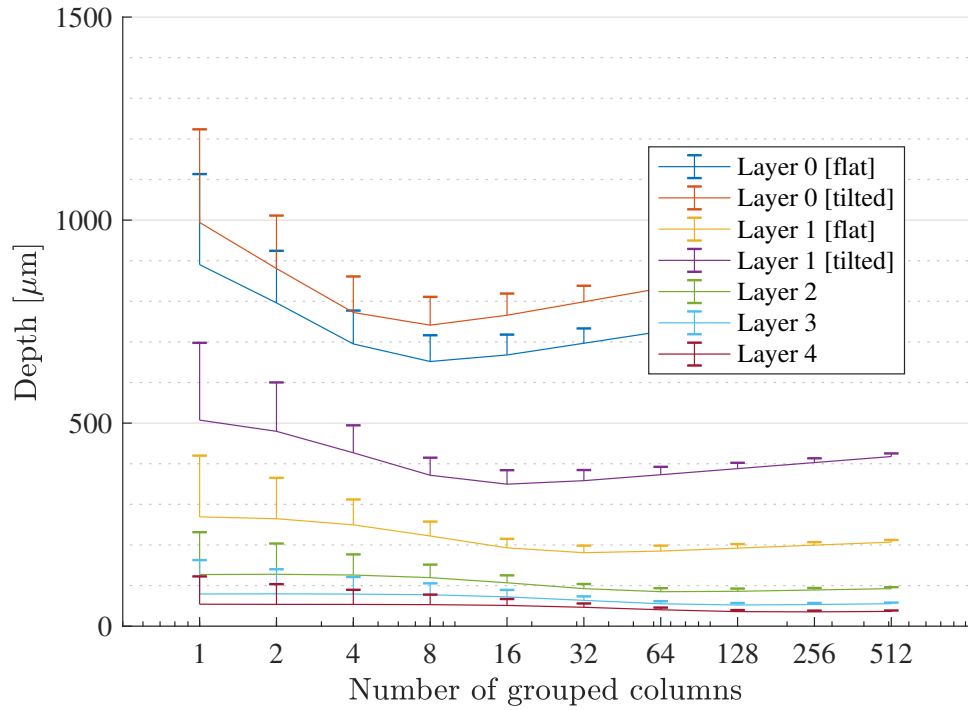


Figure 3.4: Memory depth exploiting the 4th header state

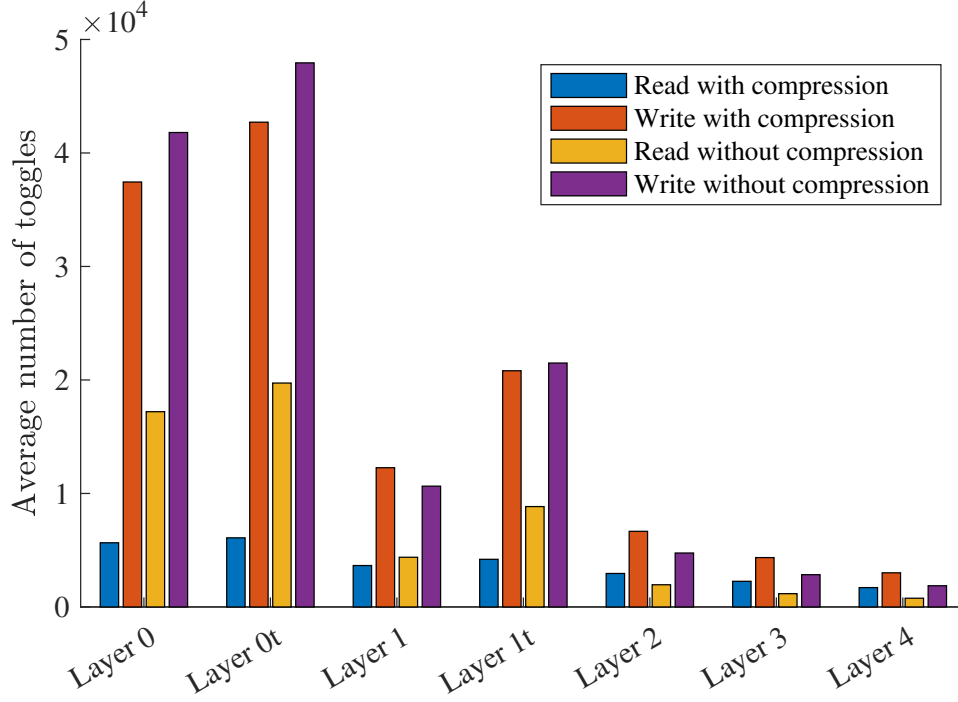


Figure 3.5: Average number of single bit toggles during the latency time

the read-write operations. Indeed, for writing and reading each data in the memory, the data buses have to be charged and discharged and, due to their high capacity, a lot of power is spent in this way. The data buses span in fact the whole memory and are thus much longer than most of the other metal tracks in the circuit.

In the plot in Figure 3.5 is shown the average number of bits read and written during a latency time. For the writing the reduction of operations simply corresponds to the reduction of data given by the compression algorithm, while for the reading the reduction is much more evident, since most of the times only the header is read, that is 9 bits in one case and 6 bits in the other. In this estimation also the data triggered for readout have been considered, assuming an average trigger ratio of 1/10.

To be fair it should also be considered the fact that a smaller memory wastes less power (because the buses to charge and discharge are shorter), thus the compressed version is comparatively even better than this. Nonetheless also this advantage would have a strong dependence on the physical memory implementation, thus it is not straightforward to add it to the comparison, we can only say that we expect it to be proportional to the decrease of memory size.

3.5 Conclusion

In principle, with this compression scheme and a grouping of 16 columns, it would be possible to achieve a memory depth of around $680\text{ }\mu\text{m}$. This means a reduction of the memory size of around 25% if compared with the $910\text{ }\mu\text{m}$ required for the non compressed scheme with a grouping of 4 columns (that is actually the best case for the non compressed format).

Nonetheless, the fragility of this compression scheme addresses the need for a protection against SEU, thus the actual achievable memory depth is around $820\text{ }\mu\text{m}$: 10% less than the non compressed format. Hence a compression is still achieved and actually the resulting scheme is even stronger against SEU than the original one. Indeed also in the non compressed scheme an error could create confusion in the scanning, as each bunch crossing would be searched relying on data sorting also in this case, otherwise it would be necessary to scan all the memory (or a substantial part of it) for every bunch crossing to retrieve inside.

The only way of retrieving data without scanning all the memory positions and without relying on data sorting would be that of using content addressable memories (CAMs), that are special memory cells that can directly compare an incoming signal with their content and return a signal in case of match. This would actually be a clean and efficient way of retrieving data, but CAM cells require much more space than RAM cells, as they implement some additional logic, thus they wouldn't be appropriate for this particular situation.

By exploiting also the fourth state of the header some more compression was achieved and actually all the known redundancy was removed, as already described in section 3.2.

The other advantage of this scheme is power consumptions, as a smaller memory wastes less power and less data require less write and read operations. Furthermore, for retrieving data in the compressed format, most of the times only the header will have to be readout, that is only 6 bits instead of 9. In fact around 70% less reading operations are required. While for the writing there is a percentage gain of 10% in terms of numbers of operations, due to the reduced amount of data.

Chapter 4

Hardware high level description

In this chapter a general description of the compressing, writing and readout circuit is given. To give a more clear explanation the hierarchy of the circuit will be preserved in the text, and each block and sub-block will be described in terms of its behaviour and implementation.

4.1 Hardware Description Languages

For the circuit implementation an hardware description language was used. Hardware description languages (HDL) are used to handle the design complexity of electronic systems. The power of hardware description languages lies in fact the possibility to create a structure with a hierarchy and thus to divide complex problems in subcomponents (called entities or components, depending on the particular language) that can be realised and simulated independently. All the components can in turn be dividend in subcomponents and so on, giving a final hierarchy. In this way it is possible to divide and face the complexity at different layers going in more details at the bottom and keep an eye on the whole system at the top. Each component after simulation and validation can be considered as reliable and can be used in the higher levels. Different kinds of descriptions are possible, some of them are intended only for simulation and others are instead intended for physical implementation.

Hardware description languages come in the form of text code. To some extent they may resemble programming languages, as they have a well defined syntax and describe precisely the desired behaviour (and take a lot of time in debugging). Nonetheless they are inherently very different, programming languages describe in fact a procedure while hardware description languages describe hardware. Thus they describe components, links and simple operations, and all of this is not executed, but is to be thought as

existing at the same time. The main operational difference is then that the statements are not consecutive, but concurrent and this gives a the great difference on how the code should be conceived and read.

Despite this, HDLs have some constructs that are also found in programming languages, but they should be intended in a different way. These constructs are `if` statements, `for` loops and other constructs that, to some extent, may be reduced to these first two. As for the `if` statements they are actually multiplexers, thus the conditions will give a choice for the outputs and if the outputs are not included in all possible branches of the `if` statement then a latch (memory) will be inferred (as some of the outputs will not necessarily change with the inputs). The `for` instead usually models a physical repetition of some component, more precisely the statements inside the loop will be considered as concurrent.

In this particular case the main components were the writing of data (that includes compression), readout of data and trigger control. They have been implemented in different HDLs, the reading and trigger control have been implemented in VHDL, as I begun and closed their implementation alone and this was the HDL I knew better. The writing instead was initially conceived together with people at CERN who suggested me to learn and use System Verilog.

4.2 Blocks description

The description follows the code hierarchy, hence a description of the top level is given first and then the components and subcomponents will be explained precisely.

The top level is interfaced with the fast synchronisation memories belonging to each column and receives data from them as an input, this interface is handled by the writing block. The top level also receives the trigger signal as an input, the trigger signal is assumed to be a single pulse that arrives in a window of 25 ns exactly 500 bunch crossings ($12.5 \mu\text{s}$) after the event it refers to, this will be managed by the trigger controller. On the output the full block will send only the triggered data that are retrieved by the readout block.

There will be no need to append the bunch crossing identifier to the triggered data, it is in fact only an internal information needed for indexing and retrieving the data, and will thus be used in writing, reading and in the trigger control that has to append to the trigger signal the correct bunch crossing identifier. This is also the reason why it was possible to conceive this information more flexibly, as actually it is only needed for internal indexing.

As for the layout, the writing block will have to be close to the fast trigger memories, hence it will be placed above the memory cells. The reading circuit instead will have to be placed close to the detector periphery, thus it

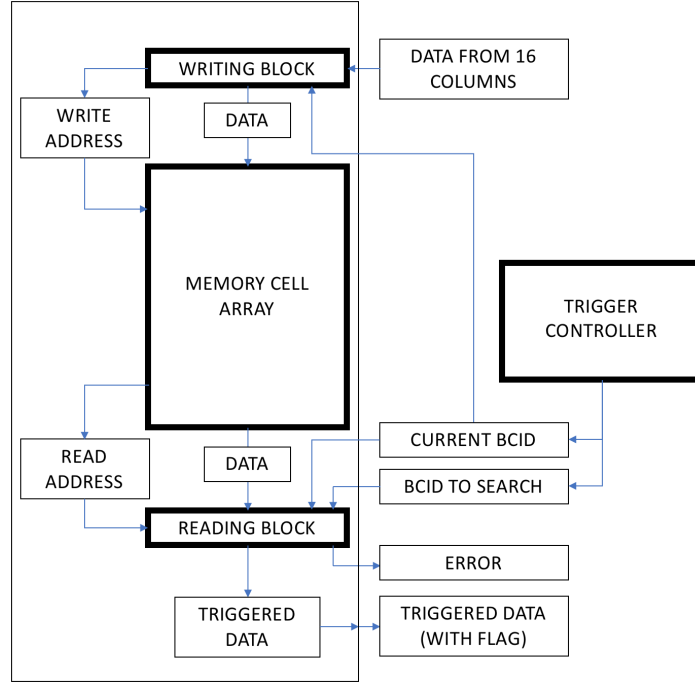


Figure 4.1: Memory blocks. Not all the signals are displayed, but only the most important

will be placed at the bottom of the whole memory block. Hence, the reading and writing blocks can be considered as independent, even though they will need to share some informations such as the bunch crossing identifier provided by the triggering circuit and the address they are using, in order to avoid conflicts.

More detailed descriptions of the components are given in the following sections.

4.2.1 Writing block

The writing block has to read data from the 16 memories by sending a read signal to each memory subsequently. Data have to be ordered in terms of bunch crossing number, compressed with the proposed scheme, the header and stored bunch crossing identifiers have to be encoded for error correction and detection using Hamming codes and finally data will have to be written in the trigger memory.

Data merging

The writing block has the hard task to merge the data coming from 16 different columns and stored temporary in the fast synchronisation memories.

The task is made even harder by the requirement that all data have to be collected and stored in order of bunch crossing. This is not straightforward because data are previously stored and thus the order in which they are received depends on how they are readout from the 16 memories and not necessarily on the order in which they have been written, in fact it will depend on the order in which the 16 memories will be read.

The synchronisation memory helps on this by providing a flag signal only when there is a stored data corresponding to the oldest bunch crossing currently being read. In order to do this the synchronisation memory has to write a portion (3 bits) of the bunch crossing for each hit data (as it would happen in the non compressing scheme) and the writing block has to provide to the synchronisation memory a portion of the bunch crossing that is currently being written in the long term memory. In the synchronisation memory a comparison is performed and thus a flag is risen only when the oldest data stored in the memory corresponds to the bunch crossing identifier required by the writing block.

More precisely the writing block will provide the same bunch crossing identifier to all the synchronisation memories in parallel, each of them will then accordingly provide a flag or not. These flags are then latched in the input of the writing block and remain latched until the first data from each of them is readout. To readout as fast as possible a priority encoder is used, it is in fact able to produce the address of the first memory with data (given a certain priority order). A read signal will be sent to the memory by just decoding the obtained address, in this way the priority encoder will both generate the address to be stored and set a priority to read faster. While the data is being read out the corresponding latched flag is cleared, so that the priority encoder can generate the next address. Another priority encoder with opposite priority is used to finish the readout faster, in fact when the two priority encoders give the same output it is clearly the last column that is being readout, thus it will be possible to accept and latch the next array of flags, instead of wasting a clock cycle looking for data when they have all been readout.

This will go on until no more flags are provided, once this happens a new bunch crossing identifier has to be sent. The number of bunch crossing bits to use in the fast synchronisation memory depends on the average time required for reading out all the data corresponding to a certain bunch crossing. The few bunch crossing bits will in fact overflow every few bunch crossings (8 in this case, as 3 bits are being used) thus a certain bunch crossing has to be read before the next bunch crossing with identical identifier is written, otherwise there will be no way to distinguish between them. The choice of using 3 bits is based on this and should avoid the problem of overflow.

State machine for header preparation

The idea explained in the previous section will allow for reading out data only for the desired bunch crossing. By relying on this it is also possible to generate more easily the bunch crossing encoding.

For doing this a state machine was generated. Each state corresponds to a particular situation, without going into the details of all the states the situations can be categorised as follows:

Waiting In this state the machine simply waits for a flag signal from one of the memories. There are actually more waiting states, because the behaviour of the circuit has to be different according to the bunch crossing that the machine is waiting for. If the machine is waiting for data belonging to the same bunch crossing of the previously stored data or to the next following bunch crossing then the header to append to the data is different. In order to exploit the 4th state of the header it would be necessary to add another state to consider also this case. There is also the case in which the state machine has been waiting for more bunch crossings and thus, once a flag is received, the first thing it has to do is to write the bunch crossing identifier with the corresponding header and then move to a state in which the writing of the data begins.

Choose synchronisation memory and readout data In this state the machine enables the readout of the memory chosen by the priority encoder and prepares the corresponding data for writing. If only one hit is stored in a word, then the machine will stay in this same state and prepare to move to the next memory (by clearing the current flag), otherwise it will go in the state described below.

Read single data word with multiple hits The machine moves to this state when there are multiple hits stored in the same memory, this happens because the pixel position inside the group is stored in 16 bits, that is exactly the number of pixels in a group. Thus, when more than one pixel is fired, there will be more than one bit set to 1, therefore it is not possible to simply encode the 16 bits into 4. The better way to proceed is just encoding all fired pixels one after the other and generate a single separate word for each one of them, this is what this state is intended to do. The strategy applied is the same used in the data flag, that is to latch the 16 bits, encode the position with a priority encoder and then clear the position that has already been read. In order to save clock cycles also here it has been implemented the trick of using two opposite priority encoders to recognise in advance the last pixel to read. After this state the machine may go in waiting state or to the previous state in order to choose another memory, depending on the particular situation.

In order to anticipate all the possible situations there are actually more states than these. This has already been explained for the wait state and applies also to the other situations. In particular all the writing states are repeated for the different possible headers that have to be written. Also the state that reads single data with more than one hit may occur while reading from the last column or not and this will actually be encoded in two different states, as the behaviour will be slightly different (even if the main structure is actually the same).

Hamming encoding (BC and header)

The bunch crossing identifier and the header have to be encoded with Hamming encoding.

As far as the header is concerned the solution is trivial as there are only 3 (or 4) states to encode, thus they will simply be written as already encoded. In fact the header shouldn't be considered as a number, but as a tag, thus in each situation when the tag has to be appended to the data it can be done with the already encoded tag. And in fact it is possible to consider all the possible tags, as they encode few known situations that actually depend also on the state of the state machine.

The same does not apply to the bunch crossing identifier, whose meaning is actually the number it encodes and in fact it includes much more possibilities. I have chosen to encode it in 10 bits in order to have an extra safety factor, in this way it is in fact possible to encode 1024 possible bunch crossing identifiers, reducing the risk of mixing numbers.

Therefore, the bunch crossing identifier has to be actually encoded by producing the required parity bits. The parity is simply evaluated with *XOR* gates, it corresponds in fact to the modulo 2 sum between all the involved bits. Thus each parity bit is calculated as the *XOR* between all the bits included in the corresponding pattern. By doing this the different patterns are given by the sets of bits summed together. The overall parity bit is then evaluated as the *XOR* of all the bits, both data and parity bits.

Unfortunately Hamming encoding is not scalable, as the relative number of parity bits and the operations to perform vary non linearly with the total number of bits. Thus a fixed component has been created to perform Hamming encoding on the bunch crossing identifier and no variables have been defined for it.

4.2.2 Memory cells array

The memory cells array is divided in two submemories, one for the header and the other for the data and bunch crossing identifiers. The former in fact will always have to be read fully, in the latter instead only the bunch crossing identifiers and the data corresponding to a triggered event should

be read. In fact, by avoiding the readout of all the other data it is possible to save a considerable amount of energy, as the memory internal buses are highly capacitive (due to their dimensions) and in each reading operation they are charged by the RAM cells.

The memory cell array could be provided by TowerJazz with the advantage of saving a lot of design time, but with some constraints on the word width and general dimensions. Therefore, also the option of making a full custom RAM has been considered, as this could also allow for even better power efficiency (for example by dividing each submemory in even smaller components). Anyway, this does not make a big difference as far as the control circuitry is concerned, because both solutions will have to be driven by an address and will be split at least in a memory for the header and one for the data in order to save power.

The RAM cells on which the array will be based are in both cases the standard cells from TowerJazz, whose dimensions are $3.38 \times 3.11 \mu\text{m}^2$. The cells are based on a simple latch made of two inverters, their control circuitry makes them dual port memory cells, that in fact allow for independent read and write in the array. They have two inputs and two outputs for writing and reading the data and its opposite and two other inputs to enable the reading and the writing. The cells are in fact designed to work in an array where the write and read lines are shared. Thus, the data are written and read by enabling the writing or reading of the desired cell (or set of cells) in the array. In this way the data to read will be asserted on the reading lines and the data asserted on the writing bus will be stored in the cells enabled for writing.

Memory overflow considerations

An important issue to consider is what happens in case of overflow. The memory is in fact dimensioned to keep most of the data given a certain desired efficiency, as stated in section 2.3, thus eventual exceeding data will have to be discarded somehow. What actually happens in case of overflow is that the address where the memory is writing reaches the address where the memory is reading, this must be avoided as it will cause uncontrolled deletion of data with a consequent desynchronisation of the readout.

The problem of overflow can be addressed in different ways and the possible solutions will concern both reading and writing circuitry. At the moment a solution has not yet been implemented in the code, as the reading and writing have been treated separately, but some options have been considered.

Actually the problem reduces to the choice between discarding data that have already be written or prevent the writing of new data (and thus discard in advance some data by not writing them). The probability to discard useful data is the same in both cases, as in fact it not possible to know in

advance if the data will be triggered or not. From an implementation point of view, instead, there are some advantages in preventing the writing. First, in this way there will be more space for the next coming data, the data stored have in fact already occupied space in the memory and deleting them would mean wasting the space used, while preventing the writing would leave more space in the memory and thus reduce the probability of other overflows. Strictly connected to this first consideration there is also a matter of power consumption, as in fact some energy has already been used for storing the data and some energy would be required to store the new data. Thus, by choosing to prevent the writing this energy can be saved. Another advantage is the simplicity, as in this scheme the writing circuit would just have to wait for the reading circuit to move forward in the memory. In the other option instead, the reading could not just jump to another position in the memory, as in this way the synchronisation would be lost, but would have to read and move faster to a next position. The writing block should hence be able to add a bunch crossing identifier with an additional tag to identify what data have been lost, but this would be a simple addition once the overflow has been detected. In addition to this there is the fact that the synchronisation memory too can provide some dampening for statistical fluctuations. Nonetheless, this potential improvement should be exploited within certain limits, as in fact also the synchronisation memory could run into overflow and the 3 bits of bunch crossing identifier stored there could overflow too. In general a simulation of the whole data flow should be performed in order to have a better understanding on which solution would be best for this situation and what would be the trade-off for efficiency.

4.2.3 Reading block

The reading block has the task of reading data from the memory and send out only the triggered data. To do this it receives from the trigger controller the bunch crossing identifier to look for, that may be the identifier of a triggered data or not. It is in fact important to move the read address even when there are no triggered events, as after $500 \mu s$ the data must be discarded to free up space for other data and also because in this way the triggered data will be found more rapidly as soon as the trigger signal arrives. Thus the trigger controller sends an additional flag when the identifier corresponds to a triggered event whose data must be sent out.

The reading block controls the memory cell array by sending the address to read and thus receiving the corresponding data. Actually the header and hit data will come from independent memory blocks (as described in the previous section), but the corresponding data share the same address, thus, besides the address, only an additional flag is required to control the enabling of the data memory.

On the output data bus instead, only the triggered data and the relative

flag will be displayed. The other outputs are the flags used to inform the trigger control when the reading has finished and a certain event has been found in the memory.

Hamming decoding

The first operation that has to be performed on the header and the stored bunch crossing identifiers is the Hamming decoding, that is actually the correction of single errors and detection of double errors. When a single error is corrected the circuit keeps on working as usual with the corrected data, if instead a double error is detected, then the circuit goes in a dedicated state used for searching the next available bunch crossing identifier, in order to regain the synchronisation and the correct indexing of data. Since the length of the header and the bunch crossing identifier is different two separate decoders have been implemented. An important difference between the two is the fact that the header should always be corrected, while the data coming from the data memory should be corrected only when corresponding to a bunch crossing identifier. At the moment the correction is always performed and the original data or the corrected data is chosen in the state machine according to the specific situation.

For the error correction all the parity bits have to be reevaluated exactly with the same scheme used for the encoding. Then an exclusive OR (XOR) operation will be performed between each received parity bit and each reevaluated parity bit, the result is a set of bits called syndrome. If the received parity bits are all equal to the reevaluated parity bits then the syndrome is an all zero vector. In this case there is no error, or at least no detected error, and the data will be treated as correct. If instead the bits are different the syndrome will work as an address to locate the bit in error according to the patterns in which the parity bits are evaluated. To allow for double error detection also the overall parity bit is reevaluated and it is used as described in section 3.3.

State machine for header decoding

To implement the bunch crossing identifier reconstruction and the control of all the readout a state machine was used. There are indeed several situations to consider, depending on the data read from the memory, on the bunch crossing identifier to search (whose data may also need to be readout depending on the read signal) and on the eventuality of an error. While at the same time it is also necessary to control the bunch crossing identifier number, the address to read from the memory and the enabling of the data memory. The data memory enabler in particular is not straightforward to control, there are in fact two cases in which it is needed. The first is when there are some data to readout and this simply depends on the external read

signal, the other case instead is when the header signals the presence of a bunch crossing identifier, but this is not known until the header has been readout. For this reason this operation has been split in two subsequent clock cycles in order to avoid a reduction of the clock frequency, to do this two separate states have been implemented.

All the states used are listed below.

setting is a simple idle state used during setup.

to_next is the state used to move from one bunch crossing to the next. It implements all the checks and read options to identify a new header.

wait is used when the bunch crossing identifier sent from outside is found. In this case the readout block simply holds the address of the memory and waits for a read command or for a new bunch crossing to search. The bunch crossing may actually have data or not, in the first case the address is held in the position of the first hit data, in the latter case instead it is held in the position where the bunch crossing identifier is stored, indicating the absence of data. Since the behaviour in the two different cases should be different, two different states have been implemented. Actually the main difference is what happens when a read signal is received, as in one case some data will have to be readout, while in the other there will be no data to send out.

pre_read - read are the states required for reading out data. The first is used for to prepare the memory for readout by enabling the data memory. The machine goes in this state if the read signal from the trigger controller is high and the bunch crossing identifier matches the one to readout. In this case the hit data belonging to that bunch crossing are simply sent to the output one after the other at a frequency of 320 MHz.

to_bc This state actually corresponds to the error state. It is in fact used to increase the reading address until the next stored bunch crossing identifier is found.

resync is the state in which the machine goes after the error state once a bunch crossing identifier is found. Its purpose is that of updating the bunch crossing counter with the number read from the memory.

4.2.4 Trigger controller

The trigger controller provides to the writing and reading blocks all the informations required for indexing the data.

A 40 MHz counter produces the bunch crossing identifier, which is then sent to the writing block and to the synchronisation memory and it is synchronised with the physical events using a 40 MHz clock provided from outside.

With a simple difference the delayed bunch crossing identifier is evaluated, that is the identifier corresponding to events occurred $12.5 \mu\text{s}$ before (that is exactly 500 bunch crossings before) and it is used to search the data in the memory. The delayed bunch crossing identifier is thus 500 units smaller than the original one and the two are both increased synchronously with the 40 MHz clock provided from outside, in this way when a trigger signal arrives it can be precisely associated with the bunch crossing identifier to read. Indeed, at the present time, we don't know when the trigger signal will arrive within the 25 ns window of time, this is why the two identifiers must be perfectly synchronous.

The delayed bunch crossing identifier is sent as is to the readout controller, unless there is a trigger signal. In this case the trigger controller sends to readout controller a read command (consisting of a single bit) and the bunch crossing identifier to be read and doesn't change it until a flag is received from the readout block that states that the readout of that event is finished.

In this block the state machine responsible of the circuit behaviour is much simpler than in the other cases and it is relative only to the readout, as the signal to send to the writing block is simply the value of the counter. In the machine there are only two states, one for searching and the other to manage the readout. The machine switches from search to read if a trigger signal is received, and goes back to search state when the readout has finished and the corresponding signal is received from the readout block.

4.2.5 Token ring

The readout has been implemented in order to be capable of working in a token ring scheme together with the trigger controller. In the token ring more readout controllers (belonging to different groups of columns) can be chained together and a signal called token is passed from one component of the chain to the next. In this case the token signal is actually the read command coming from the trigger controller. The read command is sent from the trigger controller to the first readout block, once it finishes the read it sends the "read finished" signal in the "read command" input of next block, that thus interprets it as a read command. The same happens in all the other components and it is achieved by simply connecting each "read finished" output to the "read command" input of the next block. The last component of the chain sends the token back to the trigger controller, that can thus go back from "read" to "search" state.

In this way the data are sent out of each block in series and they can

thus share a unique output bus. Another advantage is that a unique trigger controller is needed for more groups of columns. In general it is possible to put any number of blocks in the chain, hence it is also possible to find the best trade-off considering the average hit rate and trigger rate and the fact that by increasing the number of groups also the time required for readout is increased.

Conclusion

You need to add overall conclusions. - first that even for the inner layers memory size can be acceptable. - ...

You also looked at power consumption. It would be good to add this, as your coding helps to reduce the number of bits to be scanned (one can also further reduce the power consumption by splitting up the memory further, but this was not yet considered).

Bibliography

- [1] I. Berdalovic, R. Bates, C. Buttar, R. Cardella, N. Egidos Plaja, T. Hemperek, B. Hiti, J.W. van Hoorne, T. Kugathasan, I. Mandic, D. Maneuski, C.A. Marin Tobon, K. Moustakas, L. Musa, H. Pernegger, P. Riedler, C. Riegel, D. Schaefer, E.J. Schioppa, A. Sharma, W. Snoeys, C. Solans Sanchez, T. Wang, and N. Vermes. Monolithic pixel development in TowerJazz 180 nm CMOS for the outer pixel layers in the ATLAS experiment. *Journal of Instrumentation*, 13(01):C01023, 2018.
- [2] Roberto Cardella. Monolithic pixel detector for the ATLAS ITk upgrade. http://www.ub.uio.no/english/courses-events/events/ureal/2017/2017-06-09_postersphdday/cardella.html, 2017. [Online; accessed 26-June-2018].
- [3] Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006.
- [4] ATLAS Collaboration. Technical Design Report for the ATLAS Inner Tracker Strip Detector. Technical Report CERN-LHCC-2017-005. ATLAS-TDR-025, CERN, Geneva, Apr 2017.
- [5] The ATLAS Collaboration. The ATLAS experiment at the CERN Large Hadron Collider. *Journal of Instrumentation*, 3(08):S08003, 2008.
- [6] D. Dannheim, L. Musa, H. Pernegger, and P. Riedler. WG1 – Silicon Detectors. In *R&D silicon working group meeting*, March 2018.
- [7] B. Abelev et al and The ALICE Collaboration. Technical design report for the upgrade of the ALICE inner tracking system. *Journal of Physics G: Nuclear and Particle Physics*, 41(8):087002, 2014.
- [8] T. Flick. The phase II ATLAS Pixel upgrade: the Inner Tracker (ITk). *Journal of Instrumentation*, 12(01):C01098, 2017.
- [9] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, apr 1950.

- [10] D. Kim, G. Aglieri Rinella, C. Cavicchioli, N. Chanlek, A. Collu, Y. Degerli, A. Dorokhov, C. Flouzat, D. Gajanana, C. Gao, F. Guilloux, H. Hillemanns, S. Hristozkov, A. Junique, M. Keil, M. Kofarago, T. Kugathasan, Y. Kwon, A. Lattuca, M. Mager, K.M. Sielewicz, C.A. Marin Tobon, D. Marras, P. Martinengo, G. Mazza, H. Mugnier, L. Musa, T.H. Pham, C. Puggioni, F. Reidt, P. Riedler, J. Rousset, S. Siddhanta, W. Snoeys, M. Song, G. Usai, J.W. Van Hoorne, and P. Yang. Front end optimization for the monolithic active pixel sensor of the ALICE Inner Tracking System upgrade. *Journal of Instrumentation*, 11(02):C02042, 2016.
- [11] M. Mager. ALPIDE, the Monolithic Active Pixel Sensor for the ALICE ITS upgrade. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 824:434 – 438, 2016. Frontier Detectors for Frontier Physics: Proceedings of the 13th Pisa Meeting on Advanced Detectors.
- [12] W. Snoeys, G. Aglieri Rinella, H. Hillemanns, T. Kugathasan, M. Mager, L. Musa, P. Riedler, F. Reidt, J. Van Hoorne, A. Fenigstein, and T. Leitner. A process modification for CMOS monolithic active pixel sensors for enhanced depletion, timing performance and radiation tolerance. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 871:90 – 96, 2017.
- [13] T. Wang, M. Barbero, I. Berdalovic, C. Bepin, S. Bhat, P. Breugnon, I. Caicedo, R. Cardella, Z. Chen, Y. Degerli, N. Egidos, S. Godiot, F. Guilloux, T. Hemperek, T. Hirono, H. Krüger, T. Kugathasan, F. Hügging, C.A. Marin Tobon, K. Moustakas, P. Pangaud, P. Schwemling, H. Pernegger, D-L. Pohl, A. Rozanov, P. Rymaszewski, W. Snoeys, and N. Wermes. Depleted fully monolithic CMOS pixel detectors using a column based readout architecture for the ATLAS Inner Tracker upgrade. *Journal of Instrumentation*, 13(03):C03039, 2018.
- [14] N. Wermes. From hybrid to CMOS pixels ... a possibility for LHC's pixel future? *Journal of Instrumentation*, 10(12):C12023, 2015.
- [15] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

Appendix A

Code