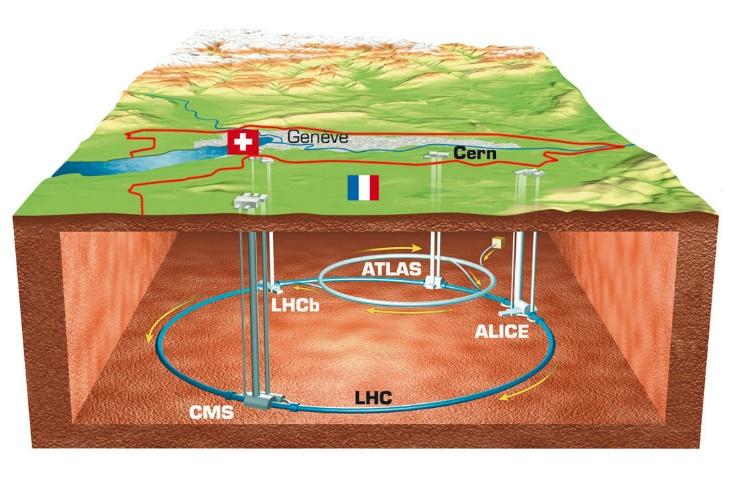
**Rapid Synchronization Recovery from Single Event Effects in the Large Hadron Collider**

**Introduction**

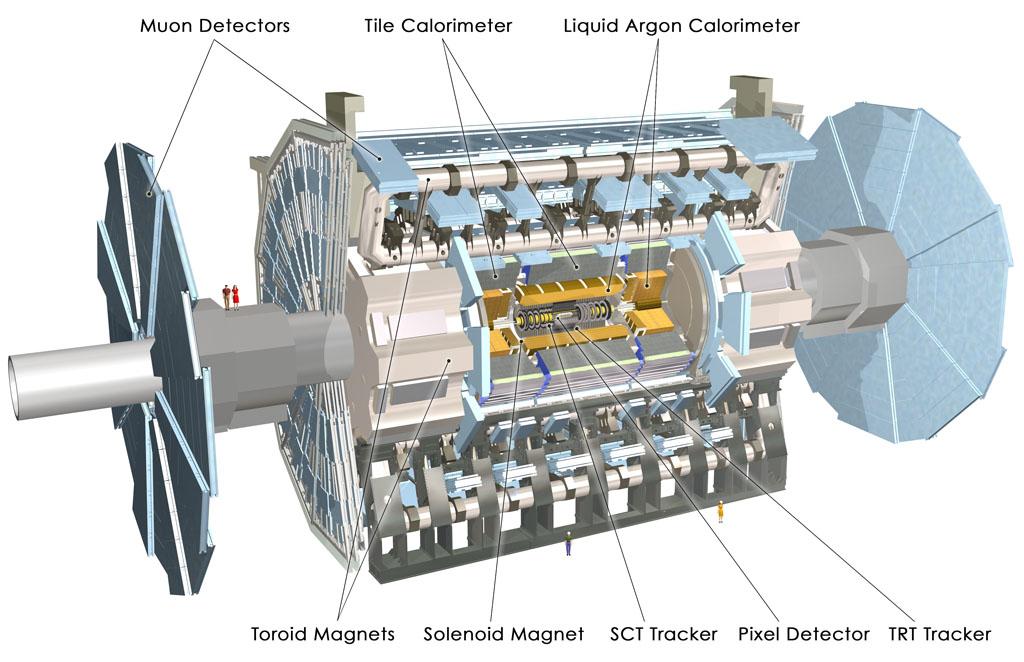
[Intro catchy paragraph]?

The Large Hadron Collider (LHC) is currently the largest and most advanced particle accelerator laboratory in the world. Sitting a hundred feet below the ground, the LHC spans the length of 27 km (16.5) miles in a ring at the European Organization of Nuclear Research (CERN)[citation]. This particle accelerator sits alongside the French and Swiss border, outside of the historic city of Geneva. Its main function is to accelerate atomic and subatomic particles close to the speed of light, collide these particles and then record the result of the collisions. While this seems like a crude way to study physics, this method reached a monumental breakthrough in 2012, with the verification of the existence of the Higgs Boson resulting in the completion of the standard model of physics[citation]. Since then, the LHC has been used to explore new physics, beyond the standard model it had been built to support.



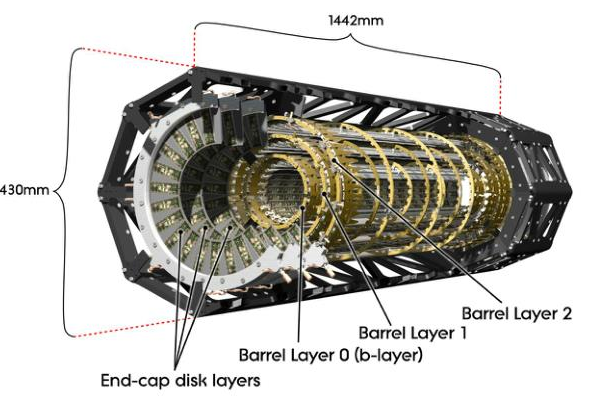
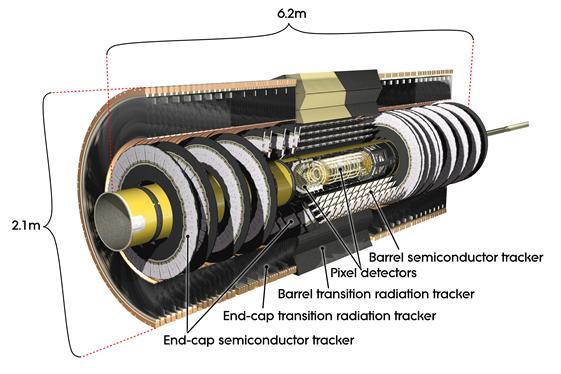
*Figure 1: Aerial* *view of where the LHC sits on the border of France and Switzerland.*

The LHC, like other particle accelerators, utilizes powerful magnetic and electric fields and charged ions to reach high energies prior to collision. Hydrogen atoms are first sent through an electric field to be stripped of atoms, leaving behind positively charged protons. Being positively charged, protons will accelerate in electric fields and arc in magnetic fields, both of which are utilized to accelerate the protons in a circle. The proton steps through a series of acceleration stages through smaller rings with electrical and magnetic fields where it’s joined with other protons into bunches of  protons. The bunches, once at the correct energy level or speed, are split and injected into the LHC, with half going in each direction around the ring. The split bunches, called beams converge, and collide in one of four sites: ALICE, CMS, LHCb, and ATLAS. Each collision of energy generates mass which is recorded by sensor instrumentation in each of the collision sites, filtered, and sent through data acquisition modules to be stored and read. Despite being a complex and precise process, the LHC is capable of producing 600 million of these collisions every second. These collisions generate as much as 60 terabytes of information every second, placing heavy demands on the sensing and filtering instruments within the collision sites [citation].



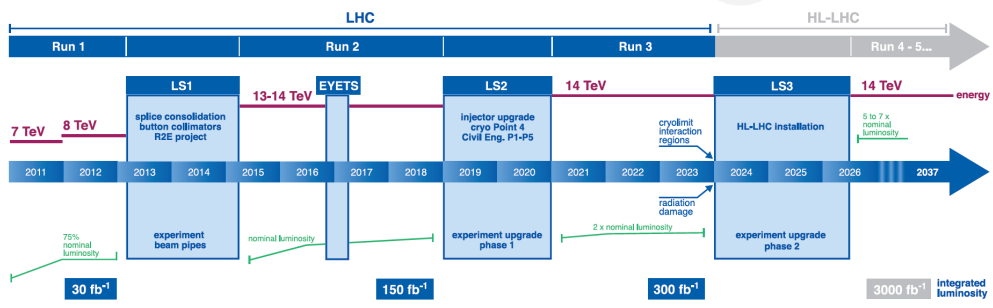
*Figure 2: ATLAS Pixel Detector*

One of the collision sites at the LHC is the ATLAS (A Toroidal LHC ApparatuS). It is the largest general-purpose detector ever made for any collider and sits at 46 meters long, 28 meters in diameter, and weighs nearly 7000 tons[citation]. Figure 2 shows the detector to scale against people to demonstrate the size. The detector is built out of multiple layers or shells of detectors designed to record the trajectory, momentum, and energy of the individual particles seen during collisions. The four major components are the Muan Spectrometer, Magnet System, Calorimeters, and the Inner Detector. The Inner Detector is the innermost layer and is further broken down into its own layers with its own innermost layer being the pixel detector.



*Figure 3: The Inner Tracker (left) and within it is the Pixel Detector (right).*

The Pixel Detector contains a silicon chip which is referred to as the Front-End chip whose basic operation is to convert charge from passing particles into a digital value. It also records the length of time this value was above a threshold value, referred to as the Time over Threshold (ToT). These values are bundled into data and passed through to a data acquisition (DAQ) module, in ATLAS’s case this is the Yet Another Rapid Readout (YARR). Unlike the Front-End chip, the DAQ modules are stowed away further from the collision point, and further process the data before forwarding it to software databases for analytics.



*Figure 4: LHC Shutdown and Luminosity timeline.*

Since its opening in 2010 and following its discovery of the Higgs Boson, the LHC has undergone shutdowns in 2013 and 2018. Another long shutdown (LS) is due in 2024 which aims to increase luminosity, the number of collisions per bunch, or the particle flux through detectors, by an order of magnitude, from 300  to 3000. This new High Luminosity LHC (HL-LHC) will demand upgrades to pixel detectors within the ATLAS. Development for the inner tracker has been ongoing over the previous years, with the development of new silicon technology to support and tolerate the increases in occupancy, bandwidth, and radiation damage resulting from the increase in luminosity.

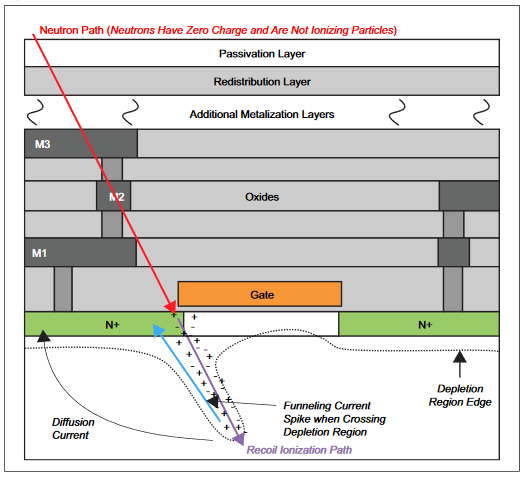
While the inner layer electronics are being updated to tolerate the increased radiation damage they’ll experience, significantly more electrical anomalies are anticipated to appear within the Front-End chip. Toleration of these anomalies as well as the mitigation of their consequences falls not only on the inner layer electronics but also upstream DAQs that not only filter and process the incoming data but will also need to detect and correct failures and corrupt data resulting from radiation damage to the electronics.

**SEEs**

*Introduction*

A natural consequence of data acquisition in high-energy environments is a significant increase in single event effects (SEEs). SEEs are not a new problem in electronics but have been recorded as anomalies in electric equipment in nuclear testing from as early as 1954. Additional instances of these apparently random errors and interruptions in otherwise perfectly operating ICs were observed in space electronics. The first paper describing this phenomenon wasn’t published until 1975.

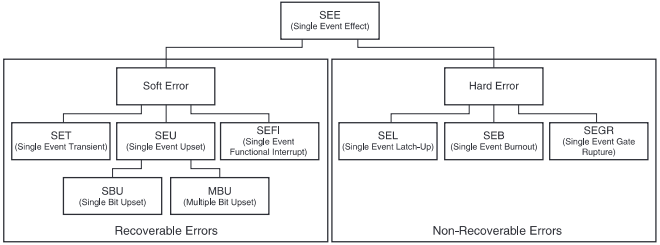
SEEs are a phenomenon resulting from interactions and collisions of high-energy particles with devices in integrated circuits. When a high energy particle passes through the silicon substrate of a device it generates charged particles along its path through a series of collisions. If the charges are generated at or near a transistor junction, the new charge can induce an upset in the transistor resulting in a change of state usually manifesting in a memory bit flip or a sudden spike in voltage or current. The particle itself can be charged, but usually, it is an uncharged particle, such as a neutron, which only begins generating this ionizing path after collision with a doped substrate.



*Figure 1: High-Energy Neutron generating an ionizing path after collision.*

These high-energy particles are present nearly everywhere, however, there are specific environments and sources where they become a genuine reliability concern. Galactic cosmic rays (GCR) from space are the most common source. GCR are made up of subatomic particles and light ions traveling at nearly light speed. While they might not necessarily directly strike ICs, they generate high energy neutrons through nuclear spallation resulting in air showers. If these neutrons retain energy, generally greater than 10 MeV, they can induce an SEE. The density or flux of these Neutrons is dependent both on altitude and longitude. For example, a New Yorker would experience twice the neutron flux as someone in Singapore, whereas airplane passengers would experience 600 times the neutron flux the New Yorker would. Given that altitude has the greatest influence on the probability of an SEE occurring, mitigation and recovery are usually considered in space and aviation applications. Although the devices in the LHC are contained and shielded from atmospheric effects, the collisions the LHC is emulating generate an equivalent environment.

Single event effects are an umbrella term for several possible errors that can occur, categorized as soft and hard errors. Hard errors are errors that cause lasting or permanent damage, and generally can’t be solved using logical techniques and will not be a focus of this paper. Soft errors are upsets to a device’s operation but are self-correcting in time or are correctable. These can generally be described as one of two events: upsets and transients.



*Figure 2: SEE Categorizations*

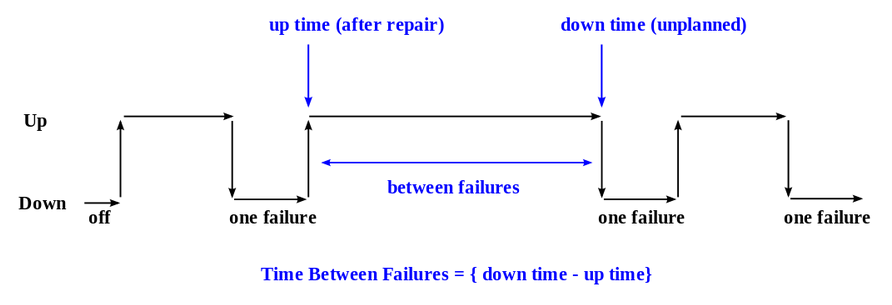
Single event upsets (SEUs) occur when high-energy particles impact transistors in memory cells, which result in an immediate bit flip. A single-bit flipped is referred to as an SBU (Single Bit Upset) and if multiple bits are affected it is referred to as an MBU (Multiple Bits Upset). If the bitflip occurs in a Datapath that otherwise doesn’t influence any system state registers, the flip propagates but ultimately is flushed out. However, if the bitflip modifies a state register, such as an FSM state, a control counter, or another register that doesn’t naturally flush its contents, the effects can be more severe up to requiring a system reset. Even worse, in FPGAs, is when the bitflip occurs within the configuration memory, resulting in LUT contents or routing to change, referred to as routing errors.

Single event transients (SETs) result when high-energy particles impact a combinatorial path of an IC and result in voltage or current spikes on a wire. There are two instances where this results in an error. The first is when the transient occurs on any data or signal line leading to a clocking element. If the pulse width of the spike is wide enough and happens at the right time, the glitch can propagate through the circuit. The second is when the glitch appears on critical system signals such as a clock or a reset. Unless the signal trees for these are built to resist errors like this, SETs can result in extra transitions and unintended system resets. While transients don’t directly influence memory cells the way SEEs do, their influence on data as well as on critical system wires can result in similar effects to SEUs or worse. When the consequences of either of the above noticeably freeze the regular function of the system, they are further categorized as SEFIs (Single Event Functional Interrupts).

*Characterization*

If it can be measured, it can be managed. Characterizing SEEs happens at the physical level and the functional level. An SEE is inherently a physical phenomenon and for that reason to recreate and emulate it, understanding and replicating the physics behind what causes the SEEs is important to understand its effects and the efficacy of mitigation methods. TODO: FINISH THIS PARGRAPH

At higher levels of abstraction, we’re more in measurements of interruption and data loss, which aren’t unique to SEEs. These might include real-time values like MTBFs (mean time between failures), MDT (mean downtime), or FIT (failures in time), the last of which is representative of the number of failures in device hours. At a more granular level, we’d also be interested in the mean amount of data lost per interrupt or the % amount of corrupt or lost data to valid data.

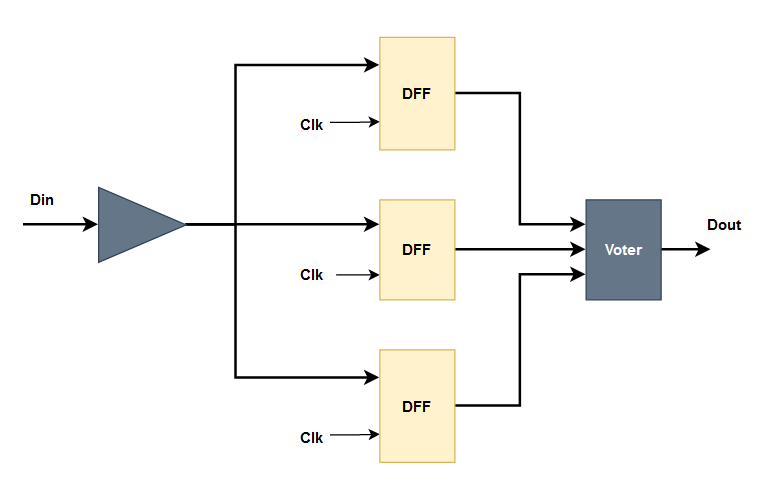


*Figure 3: TBF (time between failures)*

All that considered, SEEs aren’t unstoppable, and when they are they can often be corrected. Solutions, mitigation, and detection and recovery schemes exist to fit a variety of failures but come with their tradeoffs. In mitigation or prevention systems the difference in FIT rates and MTBFs is a standard measure of efficacy. In recovery, the detection time of a failure as well as the following recovery time are the primary metrics. Both however also consider the costs of their implementation, including device performance and implications because of these added complexities, which often result in device resource, time to market, and monetary costs.

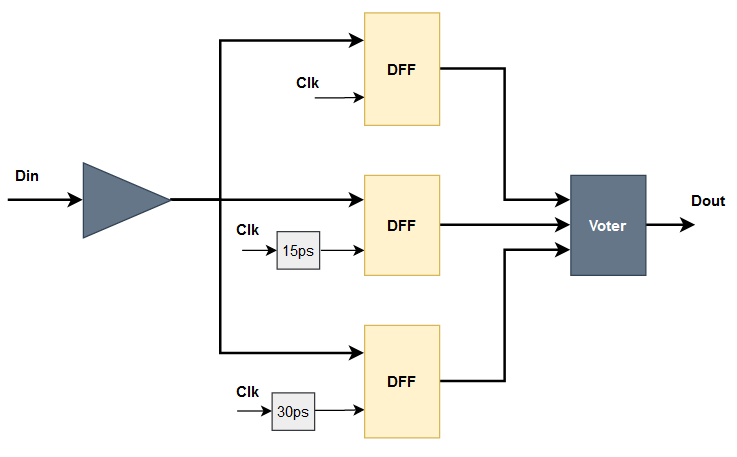
*Mitigation, Detection, and Correction: TMR*

The simplest and most implemented mitigation method implemented is TMR (triple modular redundancy). TMR is a fault tolerance methodology that duplicates system units in anticipation of one failing. By having three duplicate copies of a component, usually a DFF or other memory storage, and a voter circuit, which just forwards whichever value appears twice or more on its input lines (see Figure 4). If any single DFF encounters a bit flip or other upset, the remaining two will hold a majority and will outweigh the single corrupt DFF. The corrupted DFF can eventually expect to be rewritten or refreshed to the correct value and will no longer be in contention with the other two.



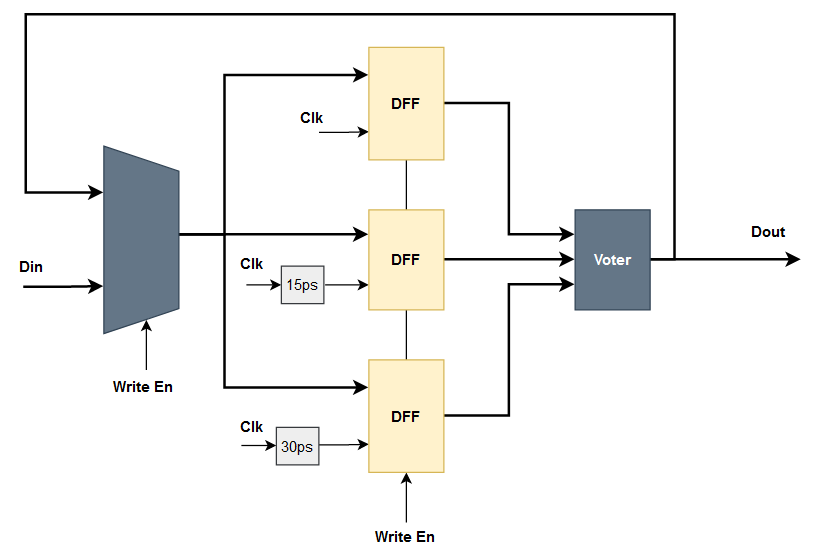
*Figure 4: Generic TMR (triple modular redundancy)*

This version of TMR resolves SEUs, but SETs remain a threat. Since all three registers receive the same input and latch onto it at the same time, an SET can simultaneously corrupt all three of them. A relatively straightforward solution to this is just to add buffers or delays onto the clock input line of the DFFs. By setting the difference between each clock’s arrival time to some time *t* (15ps in Figure 5), the TMR would be able to mitigate any SET corruptions with glitch lengths of less than *t*.



*Figure 5: TMR with delayed clocks to mitigate SET interrupts*

In circumstances when the DFFs are updated infrequently, the TMR described above can simply delay a failure rather than mitigate it. In critical memory, failures of any kind, delayed or not, are unacceptable and so correction within the TMR registers is sometimes necessary. By adding a feedback loop (Figure 6) from the voter output back into DFF input, the system can ensure not just mitigation but also correction, assuming only a single bit flips at a time.



*Figure 6: TMR with voter feedback for correction*

Further measures could be added to improve reliability in TMR. Spacing the TMR components out on a chip is important to keep two or all the DFFs from flipping through an MBU. Spaced components also mean that high fanout signals like clock and reset are separated through buffers, which would stop simultaneous upsets as well. Alternatively, the three components can be reduced to in with DMR (dual-module redundancy) so that while SEEs aren’t mitigated they are still detected. Regardless, the basic TMR and its variations all come with a cost in resource utilization on the device. In resource or power-hungry applications, this may be beyond the constraints or would require larger and more expensive devices.

*Mitigation, Detection, and Correction: ECC Memory and Scrubbing*

TMR is useful in ASICs as well as the configured logic of FPGAs, however, it becomes prohibitively expensive when applying it to the configuration memory of an FPGA. When it comes to large and system critical memories, duplicating and running each bit or word through additional logic to read it out becomes cumbersome. For these systems, ECC (error-correcting code) is utilized to provide upset detection and correction to memory blocks.

ECC memory is data storage that uses extra bits and logic to detect and correct memory corruption. This data is stored in memory the same way that data is often transmitted in communication, so that the sender, the device writing to memory, and the receiver, the same device reading from memory, can reliably expect the data to be read out correctly, or if not then to be alerted of an error and respond appropriately. ECCs come in many formats, with the most known utilizing Hamming code.

Hamming code works by allocating specific bits to be parity for different sections of the data word. TODO: FINISH THIS PARGRAPH/SECTION

While configuration memory ECC and TMR are the most common methods of mitigating SEEs, other methods have been utilized for various applications as well. Watchdog timers are broadly useful with SEEs and other functional disrupts. This timer is set up to force a system reset once it expires, but has a timer restart built-in that will restart the timer provided some checkpoint is regularly reached by the device logic. Watchdog timers add small amounts of complexity but tend to have a fixed high upper bound for error detection and an inelegant method of correction. Safe state machines are a passive recovery scheme that utilizes FSM transitions to safely guide the control state along a recovery path when illegal states or data inputs are read.

*Application to the LHC DAQ*

There are instances where none of the above are applicable, reliable, or perform to par with what a system requires. In these circumstances, application-specific detection, correction, and or mitigation systems must be developed from the ground up. Such is the case in DAQ systems of the LHC.

Unlike the RD53 pixel readout chip mixed in with the action of the LHC, the DAQs enjoy a safer, less chaotic environment hidden behind radiation insolation. However, they don’t go unaffected by SEEs. While SEEs aren’t expected to occur within the DAQ, it is expected to occasionally receive bad data from the readout chip which will be constantly bombarded by high energy particles. The pixel readout chip is directly connected to the DAQs and passes serial data across parallel lines with minimal overhead to maximize the data throughput to accommodate the increase in luminosity expected at the LHC.

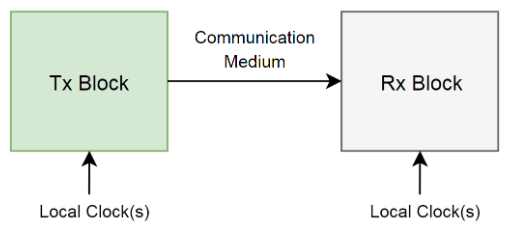
The DAQ reads bits and organizes them into distinct data blocks with a header to distinguish individual blocks. However, since these headers can only be in a particular format, the DAQ will realize that an issue has occurred once it has seen sufficient invalid headers. These invalid headers will be caused by SEEs on the readout chips end and will generally manifest in bit flips within the data block, bits added into the data block, and the counterpart bits or chunks removed from the data block. Since the data is coming in from a remote location, TMR is irrelevant, as are most other SEE mitigation techniques that check whether interrupts occurred in a particular system. ECC checking on the transmitted bits sounds attractive, but since the data is sent as a stream and bits can be added or removed ECC schemes would be unable to recover.

***YARR Tx/Rx:***

*Data Format and Protocol*

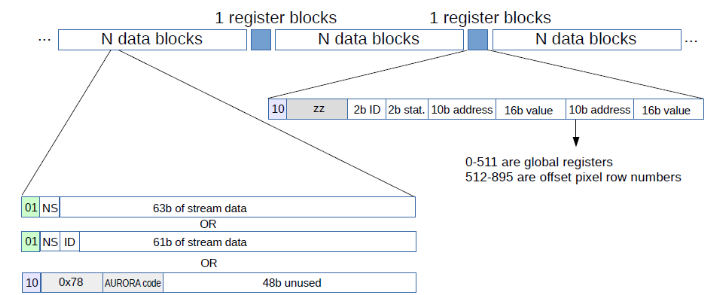
To understand the SEE detection and recovery system, an understanding of the logic and data surrounding the transmitting end of the RD53 readout chip and receiving end of a DAQ is necessary.

At the highest level, the connection between the YARR DAQ and RD53 pixel readout chip is made up of one or more serial data lines. The base case of this is a single channel or wire connection between the two blocks with no control or clock wires between the two, as shown in Figure X. Communication across links has inherently fault risk without even considering SEEs and additional complications may occur when there are differences in the arrival times or frequencies of clocks and no control signals are sent between receiver and transmitter to synchronize or perform handshaking. In the case of the YARR and RD53, the clock is shared and sent to both, mitigating most clock issues. However, problems like DC drift, excessively long run lengths, and block delimiting must still be addressed.



To resolve these issues, communication is done using a custom 64b/66b line encoding. 64b/66b encoding is a line code that prepends 2 header bits to a 64-bit payload, resulting in a limited overhead to maximize real data transmitted. The goals of the encoding are to support clock recovery, data stream alignment, DC balance, as well as transition density, and run length. Since the clocks used by both the DAQ and pixel readout chip are identical, that feature isn’t strictly necessary but had been required during early development. DC balancing of transmitted data resolves DC drift, which occurs when a receiving circuit accumulates charge and can cause issues in level detection thresholds. DC balancing ensures that a roughly equal number of 1s and 0s are transmitted so that there is no bias in either voltage direction. Transition density refers to the ratio of 1 to 0 or 0 to 1 transition in a stream against the total number of bits transferred in a bounded stream run. For example, if an 8-bit stream of bits came out as 00110100, the transition density would be 0.5 due to there being 4 transitions over 8 bits transferred. You might also note that those 8 bits are not DC balances. Related also is the maximum run length, which represents the most consecutive 1s in a row or 0s in a row. While 66b/64b encoding does not impose strict limits on transition density and run length, beyond the header which forces a transition, it does utilize logic to impose statistical bounds for these properties. The technical details of this will be developed in later sections.

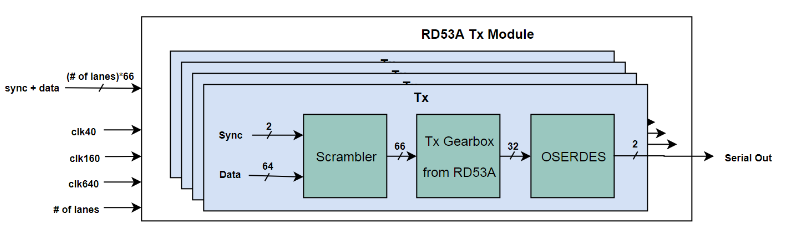
Another higher-level note of relevance is the format of the data to be sent to the DAQ. There are two types of data leaving the pixel readout which are hit data and register/service data. Hit data represents actual data collected as a result of the LHC collisions being recorded whereas register data are made up of register reads and writes issued to the pixel readout chip. If there are no hits during the transmission period, the data bits will be set to a reserved IDLE block. Register blocks are generally meant to be in response to a command issued to the readout chip, however, in the absence of such requests the blocks are filled with automatic reads of internal registers There are always N data blocks before an register block. N here is a parameterizable value and defaults to 50. N should be kept sufficiently high to maximize real data throughput. The allocation of data blocks and register blocks is strict meaning there are always N data blocks followed by a register block and this order is followed regardless of how much, if any, data or register requests there are.



As mentioned before, the blocks sent across the serial connections are 66 bits wide, with 64 bits of data and 2 prefix header bits. The header bits are strictly either 01, a hit data block, or 10, a register data block. This guarantees at least a single transition per block and sets the upper bound for run-length to 65 bits. The NS is the new stream bit and appears in non-IDLE hit data blocks. ID represents the least significant bits of the chip ID and is relevant when multiple pixel readouts are wired to a single DAQ. In the circumstance that no data is being read, an IDLE block is sent in place of a data block that has a “10” header and is followed by 0x78 in hex and an aurora code. Register blocks are slightly more involved. The 8 zz bits designate what the register data consists of, whether this was an automatic read or a manually requested one, as well as the register address value of the register and the data within it. One important note is that the data transferred across the ports of the readout and DAQ are scrambled so that the data within it appears jumbled and meaningless. This scrambling is not for security purposes but rather to improve the transition density, run length, and DC balancing of data transmitted according to the principles of the 64b/66b protocol discussed earlier. Details surrounding the implementation of the protocol follow next.

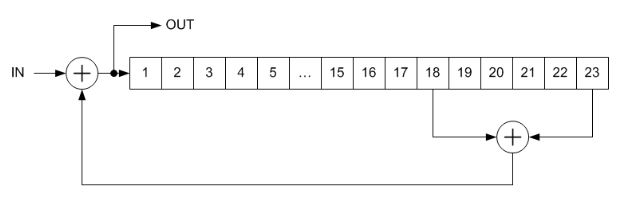
***Pixel Readout Tx Logic***

The pixel readout chip can have multiple simultaneous data output streams, however, each stream utilizes identical logic to package and transmit the data. Each port is made up of 3 major blocks which transform the data from a parallel framed 66-bit block in the format given above, to a scrambled block serialized at high speeds across the communication link: the scrambler, the output gearbox, and the OSERDES.



*Tx Scrambler:*

The 66 bits framed data blocks discussed previously paid no attention to analog issues associated with data transfer and haven’t made an especially convincing contribution to stream alignment issues. The scrambler is the tx solution to all those problems simultaneously. The purpose of the scrambler is to take a block of parallel bits and produce another block but with roughly an even number of 1s and 0s and as short of a run length and as high of a transition density as possible. The scrambled data can be later unscrambled, provided the transmitter and receiver both agree on scrambling parameters so that the inverse formula for scrambling can be applied.



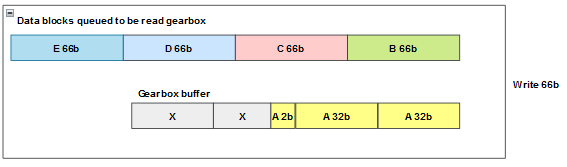
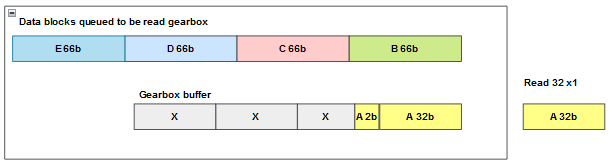
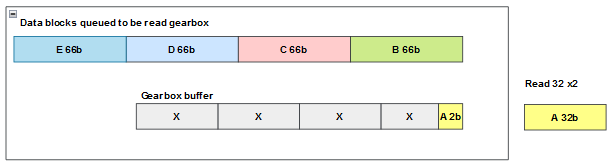
Scramblers are described with a polynomial which tells you the tap values to use for feedback. Figure X shows an example scrambler with the function . While the scrambler looks like a serial operation, it can, with sufficient resources and time, evaluate a full 64-bit block simultaneously. One thing to note is that the operation has memory so that the generation of bits for the output depends on the previously generated bits and feeding the scrambler the same block continuously, for example, the IDLE will not produce the same output cycle by cycle.

The scrambler used by the RD53 pixel readout chip uses a polynomial as per Xilinx documentation for its own IP 64b/66b protocol. This scrambler is multiplicative self-synchronizing meaning that this transmission scrambler and its receiver counterpart can be initiated at different times or with different state but still achieve synchronization. The RD53 scrambler is the first unit to receive the 66-bit data after is framed, and it separates the header, which does not get scrambled, from the data, which does. This is so that regardless of how the data is scrambled, the receiver can always anticipate seeing a 01 or 10 transition to delimit individual blocks. The header and scrambled data are then reassembled prior to moving to the gearbox.

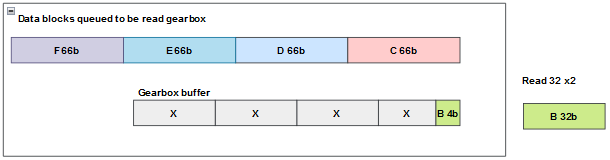
Before moving to the gearbox, it may seem like scrambling the data to introduce more transitions will work counter to the delimitation for stream alignment. With more transitions, how can the receiver be certain which transition is indicative of the header and which is simply well-scrambled data. However, the scrambler will not duplicate its output endlessly regardless of the input and the position of transitions within the scrambled data will change continuously, whereas the headers have a transition in the same bit position every 66 bits of the stream.

*Tx Gearbox:*

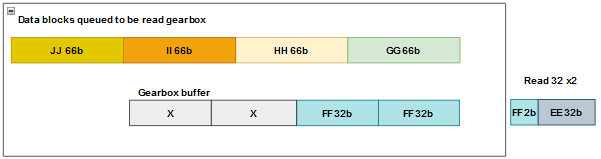
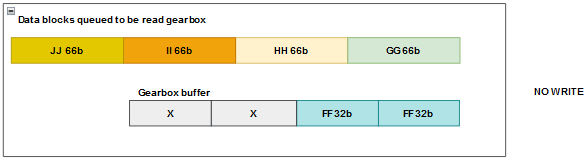
Between the scrambler and the OSERDES serializer is the gearbox. The gearbox receives the aggregate 2 bits of unscrambled header and 64 bits of scrambled data and packages them in 32-bit chunks so that they’re palatable to the serializer. The data is read out of the gearbox at twice the rate as is written in so that in every input clock cycle, 66 bits are pushed into the gearbox and 64 bits are pulled out. Clearly, this isn’t unsustainable, so every 32 blocks the flow control within the gearbox has the data driver pause for a single block so that the output can “catch up”. The function of the gearbox is illustrated below, which depicts the data that is coming in its 66-bit blocks, the 128-bit buffer gearbox split into 32-bit chunks, and the most recent read-out chunk. The gearbox is one of the more confusing elements in the output chain and is the primary target for where resynchronization occurs when synchronization is lost, and so considerable detail is included. The illustration depicts the second 32 bit read from the gearbox as happening separately from the 66-bit write for clarity, but they really happen at the same time:

1. 
2. 
3. 

*A Write and a Read x1 happens between 3 and 4*

1. 

*Multiple Writes and Reads happen between 4 and 5 so that FF is the 32 66-bit block to have been written in.*

1. 
2. 

The illustration above is an abstract depiction of the key takeaways from the gearbox’s operation. Data comes in 66-bit blocks labeled by letter and is loaded in the buffer as shown in (1). Each 32-bit chunk of data loaded in is kept separately to emphasize which bits are expected to be read out. Before the next 66-bit block is written in, two 32-bit chunks are read out of the buffer as seen in (2) and (3). It is important to note that after this three-step process is complete, there are now two residual bits still left in the buffer. When the first three steps repeat the result is shown in (4) where there are now four bits residual in the buffer. The continuous repetition of these steps inevitably leads the number of residual bits to increase by two until only 64 open bits remain following the second 32-bit read, shown in (5). Once this occurs, flow control within the gearbox freezes the data driver upstream, as shown in (6) to allow the following two reads to clear out the residual bits and start over from (1) again. This process requires a stall every 33rd write because two residual bits added each write cycle will result in 64 residual bits and only another 64 bits of open space, where the gearbox needs at least 66 to accept a write.

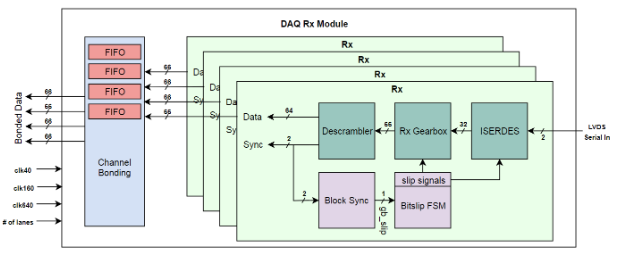
*Tx Serdes (Serializer):*

The final stop in the transmitter output sequence is the output serializer. Unlike the previous blocks, the Serdes is relatively straightforward. Its express purpose is to take parallel 32-bit chunks, split them into 8-bit quarter chunks and write them out to the communication medium. The serializer in the RD53 pixel readout chip is a combination of simple logic as well as a Xilinx IP primitive, OSERDES2. The table below shows the configuration settings:

|  |  |
| --- | --- |
| **Property** | **Setting** |
| Bandwidth | 1.28 Gbps |
| Interface Template | Custom |
| Data Bus Direction | Output |
| Data Rate | DDR (Dual Data Rate) |
| Serialization Factor | 8 |
| External Data Width | 1 |
| I/O Signaling | Differential (LVDS) |
| clk\_in (freq) | 640 MHz |
| clk\_div\_in (freq) | 160 MHz |

*DAQ Input Logic*

Just like the pixel readout, the DAQs input can have multiple identical channels, which follow an almost identical but reverse order of operations as the tx. The major difference between the rx and tx is the addition of desynchronization detection and correction elements for the block and block-set levels. At the block level are the block sync and bit slip FSMs which evaluates whether a block is valid and sets tolerance levels for how many invalid headers can be seen before desynchronization is assumed and synchronization search begins. At the block-set level is the channel bonding FIFO array, which ensures that all blocks are arriving simultaneously, and no blocks are expressly lost. Channel bonding is only necessary when more than a single channel is enabled.

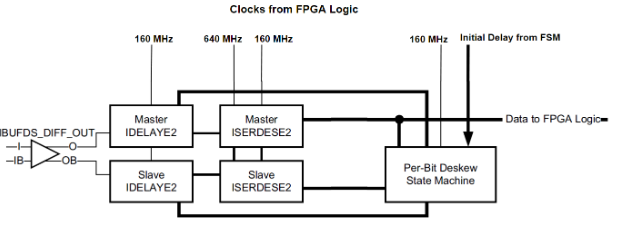


*Rx DeSerdes (Deserializer)*

The receiving end deserializer is functionally very similar to the transmission serializer except that it performs the inverse operation. The input serdes utilizes a Xilinx IP, the XAPP1017, as well as some peripheral logic to assemble serial data into 8-bit quarter chunks and assemble the quarter chunks into 32-chunks that can be passed onward toward the Rx Gearbox. The configuration settings are very similar, except for the data bus direction.

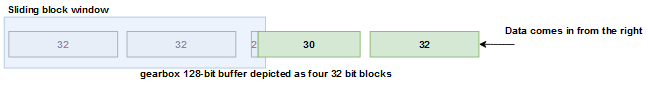
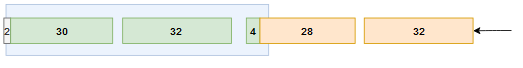
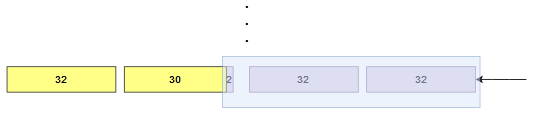
|  |  |
| --- | --- |
| **Property** | **Setting** |
| Bandwidth | 1.28 Gbps |
| Interface Template | Custom |
| Data Bus Direction | Input |
| Data Rate | DDR (Dual Data Rate) |
| Serialization Factor | 8 |
| External Data Width | 1 |
| I/O Signaling | Differential (LVDS) |
| clk\_in (freq) | 640 MHz |
| clk\_div\_in (freq) | 160 MHz |

The physical implementation of the reserialize is more involved, however, than the serializer was, utilizing a modified XAPP1012, an amalgamation of IDELAY2 primitives, ISERDESE2 primitives and a deskew state machine. The ISERDES2 primitive on its own can support 4x asynchronous sampling at 1.25 Gbps whereas the Aurora Protocol demanded 1.28 Gbps. The XAPP1017 IP utilizes the IDELAY and Deskey to control the delay of incoming data which finds the best fit sampling solution for the incoming serial data. Its worth noting that the ISEREDES2 primitive has a built-in “bitslip” mechanism, which will be revisited in later sections.



*Tx Gearbox:*

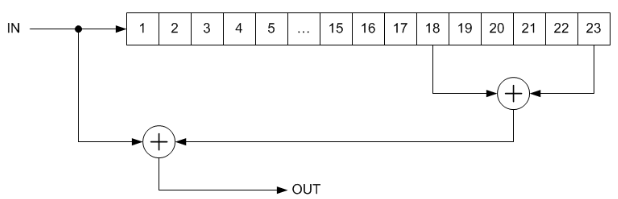
Just like in the receiver the gearboxes’ purpose is to do a transformation between the 32-bit chunk format suitable to a (de)serializer and the 66-bit frame the descrambler was built for. In this case, again, the reverse of the transmitter side is accomplished. The gearbox reads in two 32-bit chunks and produces a single 66-bit block and a block valid signal. Although, the behavior is like the tx gearbox, understanding the rx gearbox is vital to understanding the resynchronization scheme in place as well as the proposed replacement. While the tx gearbox read in individual blocks and was okay with mixing blocks when assembling chunks, the rx gearbox has the job of sorting the chunks and retrieving exactly the 66 bits which make up individual blocks. With that in mind the illustration below shows the mechanism with which the gearbox compensates for the 64-to-66-bit mismatch between the input and output:

1. 
2. 
3. 
4. A picture containing timelineDescription automatically generated
5. 
6. ShapeDescription automatically generated
7. Chart, funnel chartDescription automatically generated

In the above illustration, the two 32-bit writes, and single 66-bit read are combined to show the net effect. Tracking just the data streaming through the buffer, it’s clear that the same residual bit problem is in the rx gearbox, and they accumulate two bits per read and write cycle (2-4). In this case, however, rather than taking the 66 most significant bits the way the tx gearbox did, there is a sliding 66-bit window that selects the data block from within the buffer. Just as in the tx gearbox, this would eventually result in the window traveling past the buffer (5), and so every 33rd word has no block extracted from it and the data valid is set low (6), before restarting at the beginning again (7). If the bit range selected were followed, it’d be apparent that the window simply slides over 2 bits to the right to compensate for the residuals, and resets at the top once its least significant bit reaches the least significant bit of the buffer. The movement of the window is the second method through which bit slipping and resynchronization are accomplished.

*Tx Descrambler:*

The descrambler receives 66 bits from the gearbox, of which the two most significant bits are the unscrambled header and the remaining 64 are scrambled data bits. The descrambler is the second half to the scrambler and therefore has identical properties to it. It is multiplicative and self-synchronizing and utilizes the same polynomial, , to derive the unscrambled data. A smaller descrambler is shown in the figure below. Note that the descrambler also utilizes memory to undo the scrambling operation.

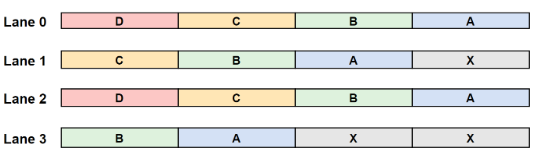


The descrambler further expands on the gearboxes data valid by evaluating the header of the data block. If invalid, or if the gearbox has its own data valid not set, the descrambler substitutes the block with 0s before passing it out of the channel and onto the channel bonding block.

*Channel Bonding:*

The final receiver component of relevance to issues surrounding synchronization is the channel bonding module. The purpose of the channel bonding module is to ensure that the parallel data blocks that pass through to the rest of the DAQ are aligned in the same way that they were when passed from the pixel readout chip.

There are two ways to evaluate whether the blocks are synced. The first is by watching the ignored blocks. As a result of the 64-bit to 66-bit transform in the gearbox, every 33rd block is flagged and ignored. If the channel bonding module receives these ignore flags at different times it knows that one or more of the channels are misaligned.



The second way alignment is evaluated is through channel bonding blocks. Aurora protocol state that a channel bonding block needs to be sent across all four lanes of the tx, and if the lanes are aligned the channel bonding module expects to receive them simultaneously.

The channel bonding module uses a set of FIFOs, one per lane. When the channels are aligned, data passes through them without or delay, as if it was purely wires. However, when channel bonding or “skipped” blocks arrive out of sync the FIFO will latch onto the sync block and wait until every lane sees the sync block.

There is clear symmetry between the primary datapaths of both the pixel readout chip transmitter and the DAQ receiver, and this symmetry would be all there is to it if communication was perfectly reliable, and no functional interrupts or data corruption occurred. However, even in the discussion of the primary datapath blocks, it’s clear that the receiver is built to detect and recover from any failures in communication. While the final channel bonding module resolves entire blocks getting misaligned, the detection of SEEs in the form of bits added, dropped, or flipped have been left unresolved.

***Detection and Resynchronization:***

As a synchronized system, the communication between the transmitter pixel readout and receiver should run smoothly, taking a pause only every 33rd cycle to allow gearboxes to catch up. However, the synchronized state is fragile, and depends on the receiver and successfully delimiting the individual blocks sent by the transmitter. The header bits serve as this delimiter and the receiver expects to see them, exactly 66 bits after the previous header. If the valid header bits are shifted even a single bit, because of an SET, or they are corrupted because of an SEU, the receiver will declare desynchronization and begin to search the stream for those header bits. The functional mechanism for this search is to utilize buffers and bitslips to shift the entire stream, or the relative expected position of the headers, left or right until the header bits are seen back in position.

To have been functional, the original system was built with some form of a recovery scheme. As will be mentioned throughout the text, the recovery scheme has many flaws which degrade performance and result an unnecessary amount of data lost because of any SEE. A proposed replacement is introduced and described.

The original detection and sync recovery scheme are made up of three primary components, the block sync FSM, the gearbox and the SERDES.

*The Serdes:*

The ISERDESE2 Xilinix primitive has a built in “bitslip” module. The purpose of the bitslip module is for word alignment, where a repeating pattern, called a training pattern, serially sent through the serializer can be reordered on the parallel side. This operation can be repeated until the correct training pattern is seen at the output of the serializer. This mechanism comes in two flavors, one for SDR (Single Data Rate) and one for DDR (Double Data Rate). In SDR mode bits are slipped in a single direction where the 7th bit is dropped and a new 0th bit is added. In DDR mode, slipping occurs in each direction, where each slip of one is followed by a slip of three in the opposite direction.

Diagram, table

Description automatically generated

Although with a repeating pattern this operation appears to be a rotation, this is not the case. Rather, the bits are really dropped until all variations are examined, at which point there is a “snapback”. This mechanism can be thought of in a similar way to how the receiver gearbox works, with a moving widow, using the same value is seen above.

A screenshot of a computer

Description automatically generated with medium confidence

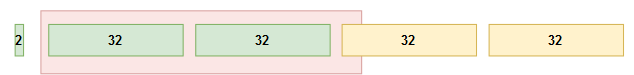
In the illustration above we are using the same repeating pattern and snapshots are taken after 8 bits are transmitted so the buffer appears to contain the same contents each time. From (2-6) this window shifting is most clear, going left 1 for every right 3, meaning this is in DDR mode. There are snaps from the (initial) to (1) and from (7) to (initial) that are less obvious, however careful inspection will make it apparent that the window shifts identically by rotation, where a shift left of 1 is equivalent to a shift right of 7.

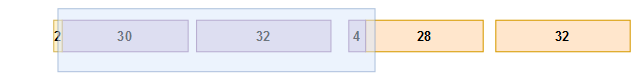
Although the ISERDESE2 bitslip module was meant for repeating patterns it still works adequately as a bit drop for the purposes of shifting the stream, provided this shift is at most 8 bits. This slipping shifts the expected position of the headers left 1 bit and right 7, meaning it favors and recovers from stream bit drops more effectively than bit adds.

*The Gearbox:*

8 bits of search range is inadequate for block sizes of 66 bits, and so a second mechanism for slipping is required. The gearbox fills this role by taking advantage of its larger buffer where it can search a much greater range compared to the Serdes. The gearbox slip is simpler and intuitive, it simply stops compensating for the residual bits for a read-write cycle. As a reminder, the gearbox has a sliding window to account for the accumulation of residual bits that result from the two 32-bit chunk to 66-bit block transform. If the window were to pause rather than slide, the gearbox will add two bits from the previous block to its block. With enough slips the gearbox would be able to search a full 66-bit block, with the only constraint that it has a granularity of 2 bits.

1. **Chart, waterfall chart

   Description automatically generated**
2. ****
3. **A picture containing icon

   Description automatically generated**
4. ****

The illustration above demonstrates how, gearbox slips would operate independently, with the buffer shown as four distinct 32-bit chunks and any block bits that were not correctly selected shown to the left of the buffer. In (1) its clear that the sliding window is not correctly selecting a unique block given that only 62 of the pale blocks is selected and 4 green block bits were mixed in. The header is evaluated as invalid, and rather than shifting the window, it stays put. In (2) we see that it happens again with the green block, however, this time only 2 bits were left out and we get yet another invalid block. In (3), the gearbox pulled in a sufficient number of bits by simply being passive to identify a block correctly. Since this block was valid, the gearbox resumes sliding its window to continue to stay in sync as residual bits continue to build up.

The gearbox slip has the benefit of having an unlimited bit range it can slip, given the time. However, it suffers from only being able to slip in a single direction, favoring stream bit drops, and slips two bits at a time, meaning that alone it would be unable to recover from a stream slip of 1 bit, either an add or a drop.

*The Block Sync FSM:*

The final piece and the one tying the last two together is the block sync finite state machine (FSM). This FSM has two purposes, to evaluate whether the system is synchronized or not and to allocated slipping to the two blocks capable of it. Both purposes are effectively achieved by tracking three counters: the gearbox data valid counter, the sync counter and the serdes bitslip counter.

Chart, diagram, box and whisker chart

Description automatically generated

The gearbox data valid counter counters the number of gearbox data valid signals seen since the previous slip. The gearbox produces a data valid every time a 66-bit word, except for each 33rd word, regardless of whether the header is considered valid or not. The gearbox counter’s primary purpose is to slow down the rate at which slips can occur. Following a slip, the counter is reset, and it takes another 16 words, as well as an invalid header prior to seeing another slip. The reason for this requirement is not entirely clear and was a target for improvement.

The sync counter tracks the number of valid headers seen consecutively, or at least it was meant to. The sync counter is incremented every time a valid word from the gearbox has valid header bits. However, since its reset to 0 behavior is gated behind the gearbox data valid counter, the first 16 words leaving gearbox can have as many or as few valid headers, which the sync counter will simply count. Only after the gearbox data valid counter reaches 16 does the sync counter have to see strictly consecutive valid header bits. The second flaw is that beyond those first 16 words, the block sync FSM is vulnerable to bit flips in the header position. Such a flip would most likely resolve in the following cycle, and the odds of the flip happening in the header multiple times in a row is extremely unlikely. However, the FSM would immediately perform a slip, resulting an actual desynchronization which will require wrapping all the way around the 66 bits to resolve. This is yet another target for improvement.

The serdes slip counts the number of serdes slips that have occurred so that a set ratio of serdes to gearbox slips is maintained. This ratio is 8 to 1, which, is results in 75% of slips being redundant. Since the serdes slip searches 8 bits before snapping back into position and the gearbox slip adding only two new bits, following serdes slips will revisit six expected header positions out of the 8 possible. Once again, this is a flaw, and most likely not the intention of the developer, and was a major target for improvement.

***The Proposed System: Header Seeker and Modified FSM***

The evaluation and performance of the original recovery scheme left a lot to be desired. The two-driver alignment system is unnecessarily complicated, and favors bit drops significantly more than bit adds. The allocation of bit slips between the two drivers is optimal. The individual header positions are checked for too long, and the criteria to gain and lose synchronization are ineffectual. A replacement recovery system developed improves on or removes each of these flaws and attains large performance boosts most metrics.

That said, the replacement recovery scheme is constrained in the resources it can utilize. The RX logic described is only a small component of the DAQ, which is hosted on an FPGA with limited resources and a defined clock rate. While only a single channel is being considered, multiple channels are instantiated on each FPGA simultaneously and so resource utilizations need to be scaled to accurately represent utilization. Any new scheme would need to simultaneously maintain a low chip footprint and operate with a clock period of 6.25 ns.

*Moving from Bit Slipping to Parallel Evaluation:*

The first issue addressed is the header alignment methodology. The original sync recovery scheme utilized two-bit slipping drivers, the gearbox and the serializer’s bitslip module, to shift position of the data stream as seen by the DAQ. Since the DAQ only considers a single position where the header bits might be at a time, it must search 1 position at a time, and wait at each position for several blocks to stream through. Only after it has seen those blocks can it determine whether the header is there, at which point it declares synchronization or moves over a bit to check elsewhere. This method is slow and can be immediately improved by evaluating multiple possible header positions simultaneously. Checking *n* locations simultaneously should reduce the time to find the correct header location equally by a factor of *n*. If *n* is chosen so that every possible position can be evaluated simultaneously, the correct header position can be evaluated very quickly. It would only be limited by the time it takes to verify that all other positions are incorrect by counting the number of consecutive valid headers in each possible position. However, to accomplish this, the system must retain additional bits from the stream as the original schemes 128-bit gearbox buffer stores just 1 possible header position in its worst-case state.

To track every possible header position, the replacement scheme expanded the gearbox buffer to allocate 67 bits for the evaluation of 66 possible header positions. 67 bits are required because each block is 66 bits wide, and the header is expected to appear in the top two bits the block. Since the worst-case scenario has 65 bits dropped, or equivalently 1 bit added, not only do the top two bits of one block get considered, but also the 65 bits of the previous block, resulting in 67 bits checked. If 66 bits were dropped, that would be a while block and the header would appear exactly where expected, albeit a block ahead of expected. The only caveat is that in the case of a bit(s) add, the header would not appear in the window of possible positions, until the next block is shift in.

[image demonstrating the above paragraph]

By tracking every bit where the header might appear, the serializer and gearbox can be simplified to their primary purposes. Since every possible position is known to the new resync scheme, bit slipping is no longer necessary. One benefit of this is that the blocks that had been responsible for bit slipping, the gearbox and serializer, can be simplified and reduced to their core functions. This has not only the benefit of simpler logic, easier maintainability, and more flexibility, but also removes some of the flaws associated with their slipping mechanisms. In the replacement scheme, the only purpose of the serializer is to transform a serial stream into 32-bit chunks. The only purpose of the gearbox is to transform 32-bit chunks into 66-bit blocks. This does however mean that a new aligner block must be developed to evaluate the header positions and guide the gearbox window to read the correct 66-bit block.

The naïve implementation of the aligner is to simultaneously evaluate each possible header position. A count of the consecutive valid could be maintained for each possible position and the position with the largest count would be determined to be the correct header position. While effective and fast, this method suffers two main hurdles. First, comparing 66 values and determining the greatest demands an enormous combinatorial path which is unachievable in the clock period required. To resolve this pipelining is necessary which complicated the design. The second is that the combinatorial resources, the pipelining costs associated result in an enormous number of resources utilized, which violates the second of the two constraints assumed. Instead, a tradeoff is made between performance and resource utilization, with the final replacement scheme still being significantly more attractive than the original scheme.

[image demonstrating the above paragraph?]

*The 6 Seeker Aligner*

The 6-seeker aligner utilizes fail fast principles to minimize hardware costs while maintaining similar performance. If the aligner can quickly recognize that a bit position does not contain a header, it can ignore it and look elsewhere. With multiple seekers, each with a unique set of bit positions to search, the aligner can evaluate all 66 possible positions while only evaluating a few positions simultaneously.

The 6-seeker aligner starts resembling a partially parallel header search, with 6 positions evaluated, and 66 header positions available for evaluation. Just like in the naïve implementation a count of the number of consecutive valid headers seen is maintained for each position. However, as soon an invalid header is seen, the seeker changes the position it is evaluating. By distributing the 66 positions possible between the 6 seekers, each seeker rotates between 11 positions. Due to the randomness of the scrambler every incorrect position still has a 50% chance of seeing what appears to be a valid header. However, the probability of seeing multiple accidentally valid headers drops exponentially and we can reasonably expect that a seeker will not usually get stuck at any incorrect position for more than a few blocks.

[image demonstrating the above paragraph]

While parallelizing the search is valuable, the fail fast principle at work offers the most significant benefits to performance. A key feature of this search mechanism is that there is no tolerance whatsoever for invalid headers. As soon as a seeker reads a 11 or 00 it rotates to the next position it is responsible for. The original recovery scheme also had no tolerance, however this only kicked into effect after having seen 16 blocks pass through, meaning that even if the first 16 showed invalid headers, the recovery scheme would not move on to search the next possible position until after the 16th word was seen. The original scheme has the exact same probability of determining that the current position is invalid after the 17th block passed through as the seeker has on its 1st block.

Failing fast doesn’t come without consequences, however. If the stream were prone to bit flips, where the correct header position doesn’t change but in an incorrect header is seen for one block, the scheme could be constantly pushed into recovery mode*.* Luckily that can be ignored for this application since the ratio of erroneous blocks to valid ones is vanishingly small*.* Further having a tolerance built in would have additional negative effects on performance. The increased tolerance works against the fail fast methodology used since allowing invalid headers to be seen along with consecutive valid ones increases the time spent evaluating an invalid position.

Any factor of 66 could have been used for the seeker count, however 6 seekers provided the best performance-cost product among those tested. Since 66 positions encompasses all possible header positions, a factor of 66 is used for the seeker count so that the positions can be split evenly among them. The most attractive counts were 11, 6 and 3 as they appeared to have the best balance of performance seen against resources utilized. To determine the best of the three, the minimum product of the blocks lost to recovery and resources utilized on an FPGA was used.

[table showing results of the above paragraph]

With bit slipping done away, and tolerance handled by the aligner, the responsibilities of the block sync FSM are reduced. This immediately resolved some of the major issues within the FSM, and, just like with the gearbox and serializer, improve the maintainability and testability of the logic. The remaining purpose of the block sync FSM is to determine whether the RX is synchronized. This is achieved by tracking the number of consecutive valid headers seen up until it reaches a configurable value SYNC MAX. This purpose of setting a minimum number of consecutive cycles is so that invalid data isn’t labelled as valid if an incorrect position happens to see a few consecutive correct values. The value defaults to 16, and as a result guarantees that at least 16 blocks, aside from the first corrupted one, are lost during resynchronization. All other blocks lost are a result of the seekers attempting to find the correct position and having its consecutive valid count pass all incorrect position counts.

*Performance and Resource Utilization*