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

Anatoliy Martynyuk

A thesis

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2022

Committee:

Scott Hauck

Shih-Chieh Hsu

Abstract

The Large Hadron Collider will undergo an upgrade to become the High Luminosity – Large Hadron Collider after a long shutdown through 2029. This upgrade will significantly increase the particle collision rate, which will result in more adverse radiation effects, called single event effects, in the electronics close to the collisions. To reduce data lost after a functional upset, the inter-chip communications synchronization mechanism must be improved. A new synchronization system is developed and compared with the original in term of performance and hardware resource utilization. The final product shows enormous benefits in performance, reducing blocks lost to nearly 1% of the original losses with only a moderate increase in FPGA resources to accommodate the improvement. The necessary background, design, decisions, and results will be outlined and discussed in this thesis.

Table of Contents

1.0 Introduction

The Large Hadron Collider (LHC) is currently the largest 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) [1]. This particle accelerator sits across 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 Higgs Boson's existence, resulting in the completion of the standard model of physics [1]. Since then, the LHC has been used to explore new physics, beyond the standard model it had been built to support.

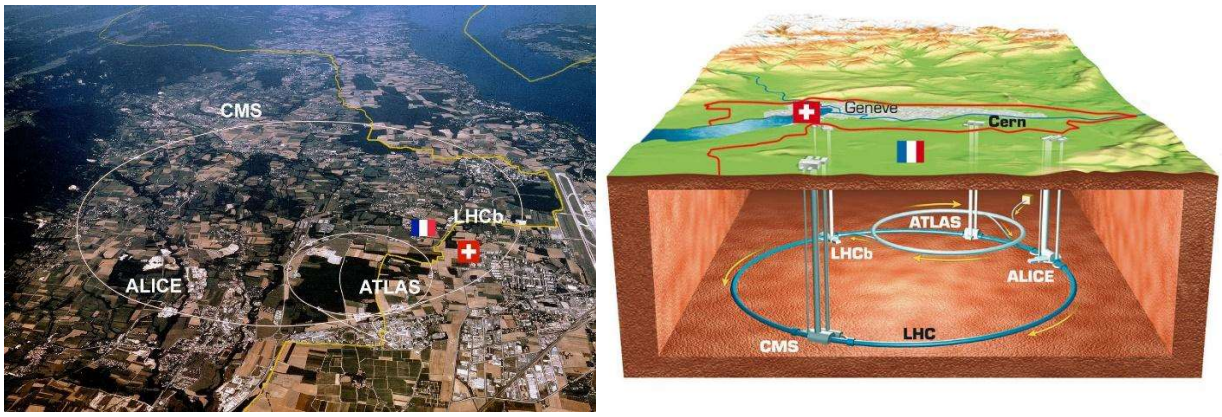

Figure 1: Aerial view of where the LHC sits on the border of France and Switzerland. [1]

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 electrons, 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 where it's joined with other protons into bunches of $1.1 * 10^{11}$ [2]. The bunches, once at the correct energy level, 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, or 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 [1]. 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 [2].

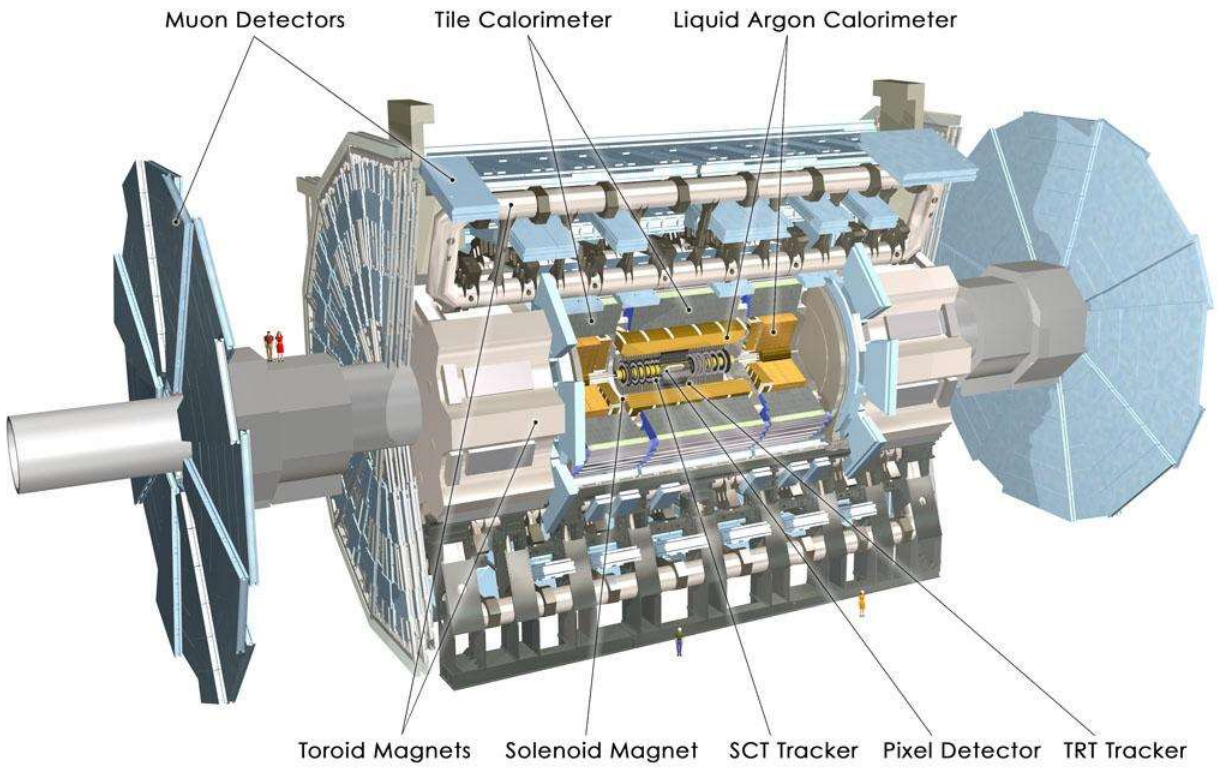

Figure 2: ATLAS Detector [2]

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[3]. 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 Muon Spectrometer, Magnet System, Calorimeters, and the Inner Detector. The Inner Detector is the innermost layer and is further broken down into its layers with its own innermost layer being the pixel detector.

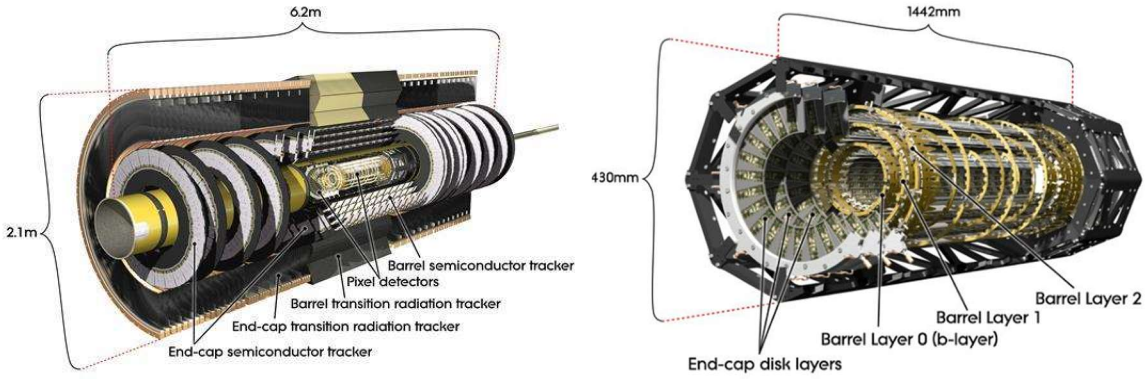

Figure 3: The Inner Tracker (left) and within it is the Pixel Detector (right). [2]

The Pixel Detector contains a silicon chip called the Front-End (FE) chip, whose basic operation is to convert charge from passing particles into a digital value [3]. It also records the length of time this value was above a threshold value, referred to as the Time over Threshold (ToT). Both values are joined into a data packet and passed through to a data acquisition (DAQ) module; in ATLAS's case, this is Yet Another Rapid Readout (YARR) [3]. Unlike the Front-End chip, the DAQ modules are kept far away from the collision point. They further process the data before forwarding it to software databases for analysis.

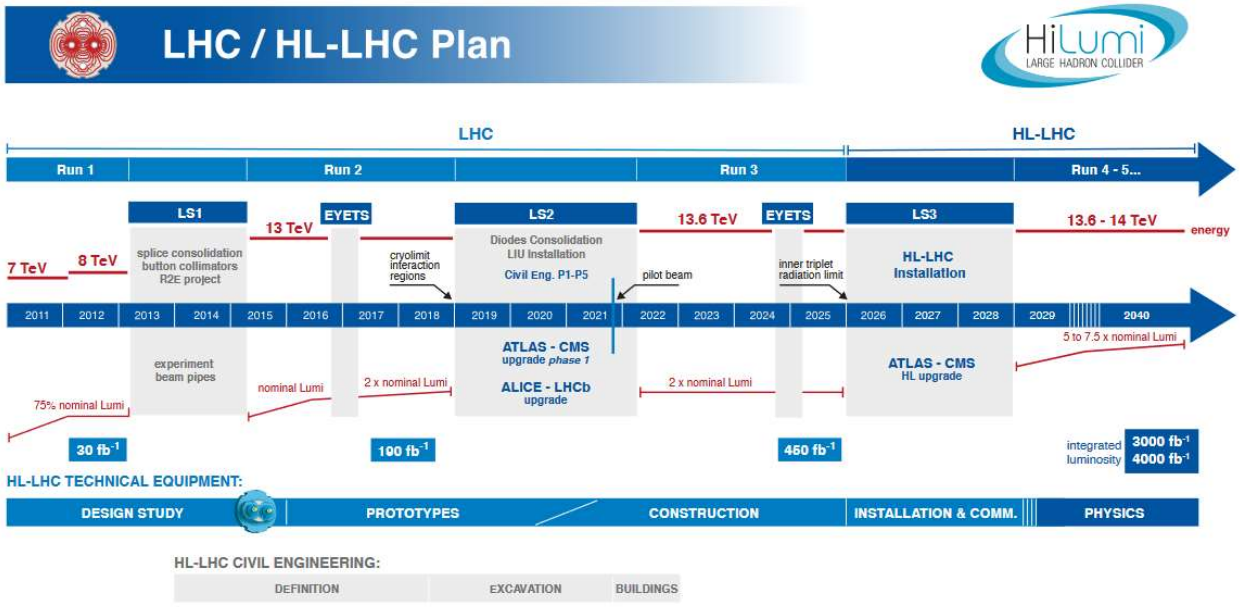

Figure 4: LHC Shutdown and Luminosity timeline. [4]

Since its opening in 2010, the LHC has undergone long shutdowns (LS) in 2013 and 2018 and is expected to undergo another through 2029. This upcoming LS aims to increase luminosity (the number of collisions per bunch) by an order of magnitude, from 300 fb^{-1} to 3000 [4]. This new High Luminosity LHC (HL-LHC) will demand upgrades to pixel detectors within the ATLAS [4]. 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.

The development of the pixel readout chips for ATLAS is led by the RD53 collaboration i.e. Research and Development -53. The pixel chip being developed is for both the ATLAS and CMS pixel detector upgrades, with variations customized for each detector: The CMS pixel detector is closer to the beam and the ATLAS pixel detector is bigger [5]. This paper will continue to reference the RD53 as the transmitter responsible for passing data onto the YARR DAQ [7].

While the inner layer electronics are being upgraded 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 on the downstream DAQs that not only filter and process the incoming data but will also need to detect and correct the failures and corrupt data resulting from radiation effects on the electronics.

2.0 Single Event Effects

2.1 Background

Single event effects (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 [13].

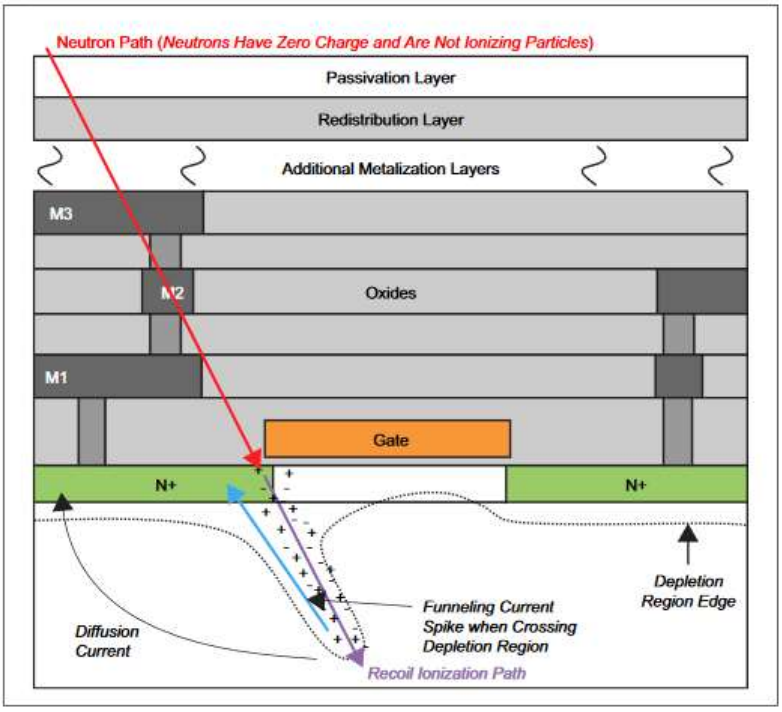

Figure 5: High-Energy Neutron generating an ionizing path after collision. [13]

SEEs are not a new problem in electronics but have been recorded as anomalies in electrical equipment in nuclear testing from as early as 1954 [8]. 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 [8].

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 [12]. 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 [12]. If these neutrons retain energy, generally greater than 10 MeV, they can induce an SEE [9]. The density or flux of these neutrons is dependent both on altitude and latitude. 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 [8]. 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 creating 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 logic 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 [12].

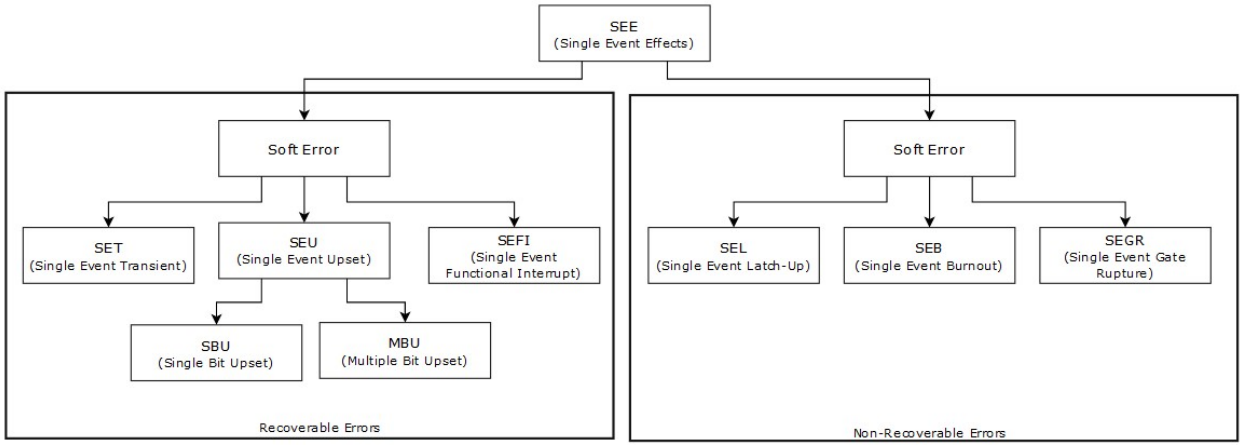

Figure 6: SEE Categorizations [12]

Single event upsets (SEUs) occur when high-energy particles impact transistors in memory cells, which result in an immediate bit flip. A single bit flip 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 [8].

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) [12].

SEEs are characterized using several metrics. Failures in time (FIT) is a measure of the rate at which events occur, where the unit of time is 10^9 hours. Mean time between failures (MTBF) is closely related but describes the average length of time a system is operational between individual functional upsets. The mean down time (MDT) is then the length of time for which a system is interrupted or acted erroneously after an upset.

Since their discovery, efforts have been made to mitigate, detect, and resolve single event effects. Solutions and recovery schemes exist to fit a variety of failures but come with tradeoffs. In mitigation or prevention systems the difference in FIT rates and mean time between failures are standard measures of efficacy. In recovery schemes, the detection time of a failure as well as the mean down time are the primary metrics. Both however also consider the costs of their implementation, including device performance implications because of these added complexities, which often result in device resource, time to market, and monetary costs.

2.2 Triple Modular Redundancy

The simplest and most implemented hardware mitigation method is triple modular redundancy (TMR). 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 Flip Flop (FF) or other memory storage, as well as a voter circuit, which forwards whichever value appears twice or more on its input lines, single errors can be automatically ignored (see Figure 7). If any single FF encounters a bit flip or other upset, the remaining two will hold a majority and will outweigh the single corrupt FF. The corrupted FF can eventually expect to be rewritten or refreshed to the correct value and will no longer be in contention with the other two.

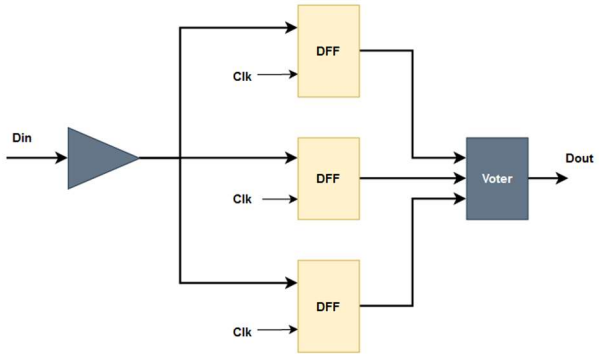

Figure 7: Generic TMR (triple modular redundancy)

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 8), the TMR would be able to mitigate any SET corruptions with glitch lengths of less than *t*.

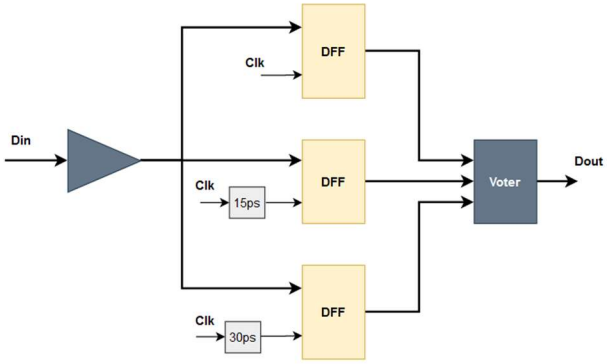

Figure 8: TMR with delayed clocks to mitigate SET interrupts

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 9) 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.

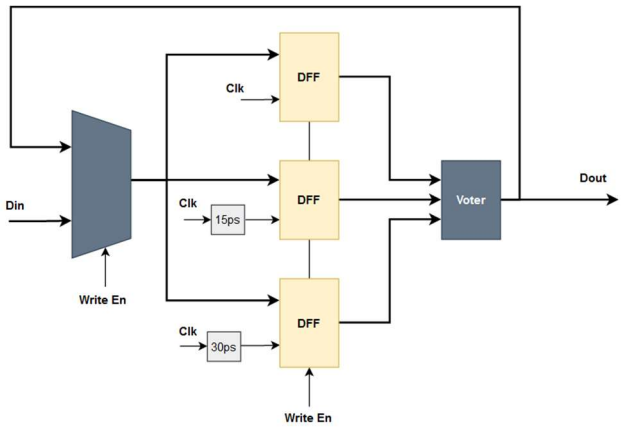

Figure 9: TMR with voter feedback for correction

Additional measures could be explored to improve reliability in TMR. Spacing the duplicate memory components in hardware decreases the risk that two or more flip simultaneously to a multiple bit upset. Similarly, having the components wired to different branches of global signal trees prevents multiple duplicates from experiencing the same transient effects. These put constraints on the layout of the FPGA and may result in non-optimal placement. Steps in the other direction can be taken as well. Triple modular redundancy can be substituted with dual modular redundancy in applications where mitigating upsets isn't as important as simply detecting them, or resource constraints limit the resources available for mitigation.

2.3 Application to the LHC DAQ

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

The circumstances of LHC DAQ don't allow for conventional SEE mitigation and recovery techniques. Unlike most electronics dealing with SEEs, the DAQs at the LHC experience very little radiation, practically none at all. This is because while the pixel readout chip is located directly at the collision site, the DAQ is bunkered 80 meters away and lies 100-meter underground [21]. Instead, the DAQs inherit corrupted data from the pixel readout chips they communicate with and must resolve lasting communication interruptions that result from them.

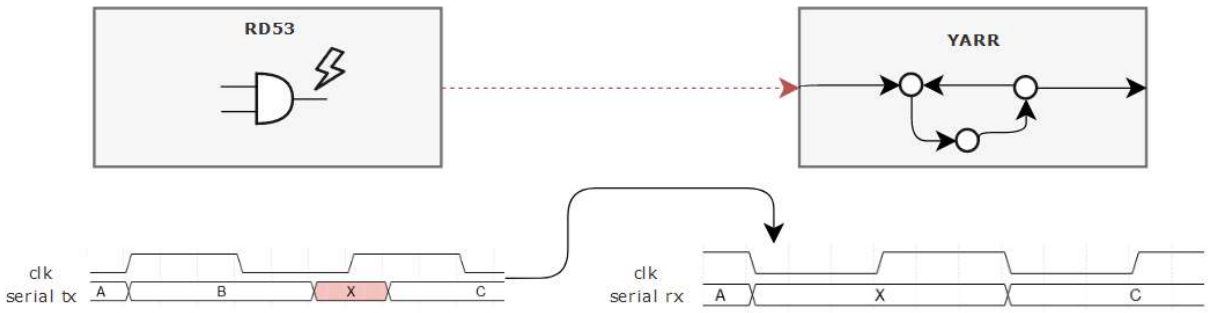

Figure 10: Illustration of how SEEs on the RD53 result in errors seen by the YARR.

The RD53B pixel readout itself is also hardened against radiation. However, current testing predicts that a significant number of effects are expected within chip memory. In protected and corrective memory an estimated 0.2 bit flips occur per hour whereas in the worst case as many as 230 flips occur per minute [14]. Memory aside, the serialized data on the readout links have been shown to be especially susceptible to single event effects [14]. It's this very sensitivity that the YARR DAQ aims to resolve through early detection and rapid recovery.

3.0 YARR Tx/Rx

3.1 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 multiple differential serial data connections. For simplicity, a simple case with just a single differential data line will be considered and is shown in Figure 11. This connection has no control, handshaking, or reference clock signals but a single serial wire transmitting a bit at a time. Communicating serially without these additional signals poses several analog and digital challenges.

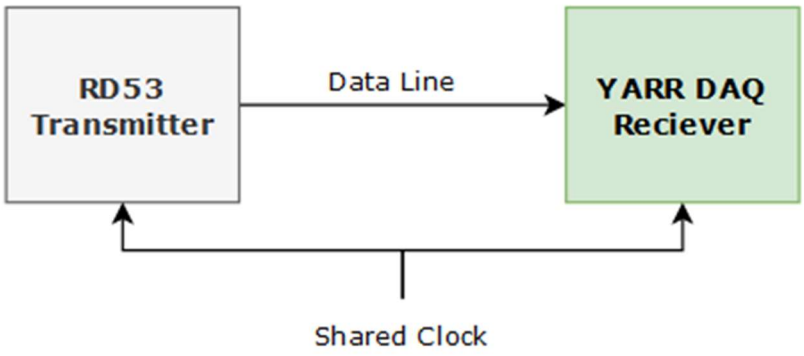

Figure 11: High level diagram of the communication link from RD53 to YARR.

To address these issues, communication uses a custom 64b/66b line encoding. 64b/66b protocol offers a variety of features that support this single-line communication model:

- **Clock recovery**. Clock recovery utilizes incoming data transitions to allow a receiver to determine the timing of incoming data without a separate clock. This feature was relevant previously but isn't utilized by the YARR and RD53.
- **Steam Alignment**. Stream alignment allows a receiver to determine packet boundaries in a continuous stream of data. This is accomplished by the receiver's synchronization scheme.
- **DC Balance**. DC balance standards require an even number of 1s and 0s in a transmission. This mitigates the biasing of voltage which has adverse effects on level detection threshold circuitry.
- **Transition Density**. Transition density is the ratio of 0 to 1 or 1 to 0 transitions within a stream to the number of bits transmitted.

- **Run Length**. Run length is the maximum consecutive 0s or consecutive 1s in a stream. Although 66b/64b encoding doesn't guarantee a maximum it imposes statistical bounds.

In 64b/66b encoding, data blocks are 66 bits and have two distinct elements. The two most significant bits in the blocks are the header, which is strictly a 01 or 10. This sets a maximum upper bound on run length and guarantees that one transition will occur every 66 bits. The remaining 64 bits are scrambled data. The data needs to be unscrambled on the receiver end to recover the original 64-bit payload. Xilinx provides an IP core called 'Aurora 66B64B', however, it is not compatible with all Xilinx FPGA families and utilizes specialized hard silicon as well as a lot of additional logical overhead to interface it. Instead, a custom protocol is used which utilizes more basic and accessible primitives while keeping the block codes the same as that of the Xilinx implementation [19]. For more information on 64b/66b protocol see Appendix A.

Figure 12: Field makeup of a 66b block.

3.2 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: the scrambler, the gearbox, and the serializer. These 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.

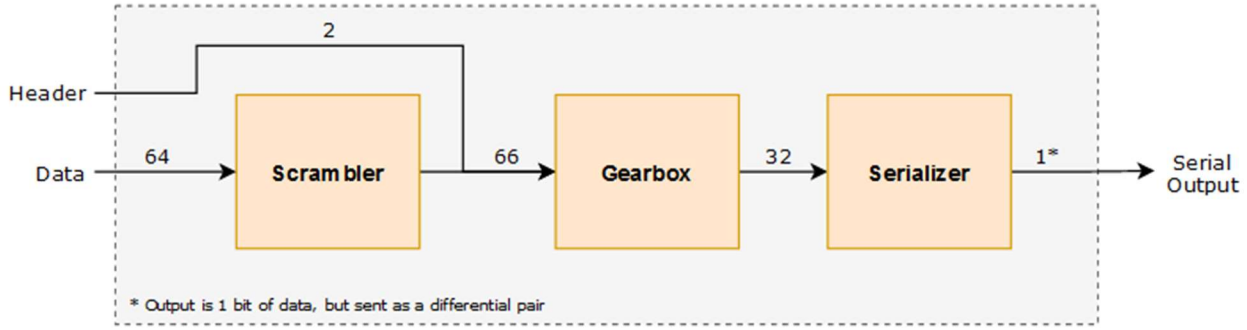

Figure 13: High-level block diagram of RD53 Tx logic.

3.2.1 Tx Scrambler

To accomplish the DC balancing, transition density, and run-length requirements set by the line encoding algorithm, the data being transmitted must be modified. Data stored in registers, and data produced from atomic collisions don't tend to already have the characteristics required. To transform it so that it does, the data is passed through the scrambler. The scrambler takes a block of a set bit width and applies a function to it to produce a second block that is statistically likely to have a short run length, a high transition density, and an equal number of 1s and 0s.

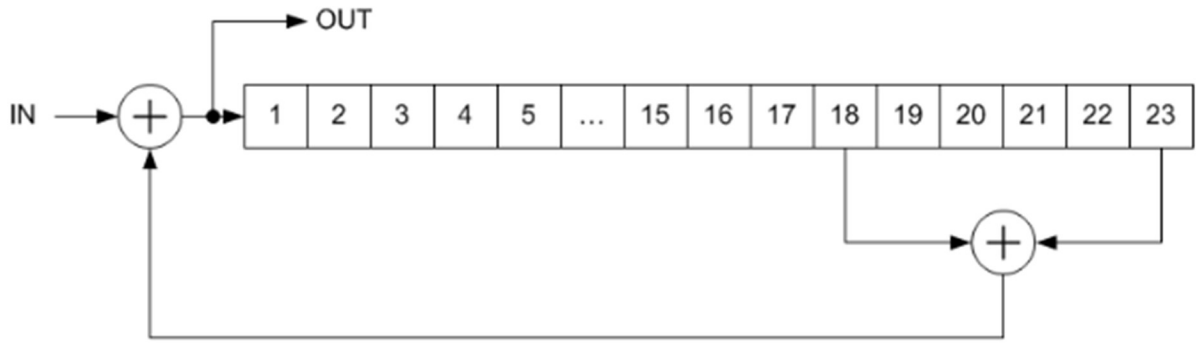

Figure 14: Example 23-bit scrambler. [6]

Scramblers are described with a polynomial which tells you the tap values to use for feedback. Figure 14 shows an example scrambler with the function $1 + x^{18} + x^{23}$. 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. Feeding the scrambler the same block continuously (for example an IDLE) will produce different outputs cycle by cycle.

The scrambler used by the RD53 pixel readout chip uses a $1 + x^{39} + x^{58}$ polynomial as per Xilinx documentation for its own IP 64b/66b protocol [15]. This scrambler is multiplicative self-synchronizing, meaning that this transmission scrambler and its receiver counterpart can be initiated at different times or with different states but still achieve synchronization. The scrambler in the RD53 receives a 64 block of payload data, which it transforms into 64 bits of scrambled data. This data is appended with two header bits (which are not scrambled) and passed forward to the gearbox.

Before moving to the gearbox, it may seem like scrambling the data to introduce more transitions will work counter to the stream alignment requirement. With more transitions, how can the receiver be certain which transition is indicative of the header and which is simply well-scrambled data? However, since the scrambler produces essentially randomized data, each pair of bits, except the designated header, in the stream is 50% likely to not contain a transition. Therefore,

if we watch a position for *n* blocks, there is only $\frac{1}{2^n}$ probability of having a transition appear in all *n* blocks unless, of course, it is the correct header.

3.2.2 Tx Gearbox

Between the scrambler and the serializer is the gearbox. The gearbox receives the 2 bits of header and 64 bits of scrambled data and packages them into 32-bit chunks to prepare them for the 32-bit serializer. The data is shifted out of the gearbox at twice the clock rate as it is shifted in. Therefore, for every input clock cycle, 66 bits are shifted into the gearbox and 64 bits are shifted out. A result of this input-output size mismatch is that there is an accumulation of 2 bits every input cycle. To address this there is a pause every 33rd cycle to shift out the accumulated bits and allow the gearbox to "catch up".

3.2.3 Tx Serializer

The final step in the output sequence is the serializer. The serializer shifts in 32-bit chunks and shifts out each serially, meaning a single bit at a time. To accomplish this the serializer uses a faster, clock as well as DDR data transmission to transfer data on both the rising and falling edges of the clock. The abstract behavior can be described as a parallel in, serial out register with a bit width of 32. Serializer parameters are given below.

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

Table 1: Tx Serializer configuration. [6]

3.3 DAQ Rx Logic

The Rx logic within the DAQ mirrors that of the pixel readout Tx. Each of the blocks used to implement the line encoding has counterparts to undo the encoding in the reverse order. Built into each block as well as some periphery, however, is additional logic to ensure that stream alignment is maintained. Just like with the Tx, multiple duplicate channels can simultaneously service multiple input lines; however, for simplicity, we consider just a single channel.

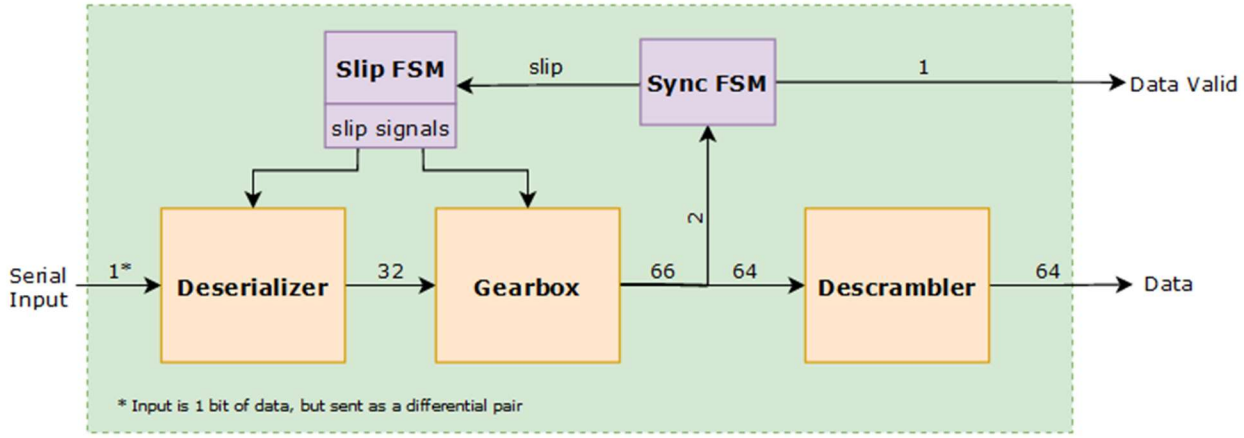

Figure 15: High level block diagram of YARR Rx logic.

3.3.1 Rx Deserializer

The receiving end deserializer is functionally very similar to the transmission serializer except that it performs the inverse operation. The deserializer receives serial data at a high rate which is accumulated into a 32-bit chunk and passed inward to the gearbox. Just like the serializer, the deserializer can be modeled as a serial in parallel out register. The data is grouped into four 8-bit sets. These sets are then assembled into a 32-bit chunk. Just as in the Tx serializer, the deserializer's parameters are given below:

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

Table 2: Rx Deserializer configuration. [6]

3.3.2 Rx Gearbox

Just like in the transmitter, the receiver gearbox's purpose is to perform a transformation between 32-bit chunks from the deserializer, back into the 66-bit RD53B block. In this case, the reverse of the Tx side is accomplished. The gearbox shifts in two 32-bit chunks and shifts out a single 66-bit block along with a block valid signal every cycle. Although the behavior is similar to the Tx gearbox, understanding the Rx gearbox is vital to understanding the resynchronization scheme in place as well as the proposed replacement.

The basic behavior of the gearbox is to shift stream data through a buffer. The deserializer produces and passes 32-bit chunks to the gearbox, where it is shifted into an internal 128-bit buffer every half cycle. On the output, a 66-bit block is shifted out every cycle, containing the original scrambled data and unscrambled header. Because of this 64-to-66-bit transform, 2 bits more are taken out of the buffer every cycle than is shifted in.

Figure 16: Gearbox component behaviors: (left) Data shifted in as 32-bit chunks and (right) data shifted out as 66-bit blocks

The 66-bit block on the output of the gearbox is selected by a sliding window. While the buffer is 128-bits wide, a select 66 bits are chosen every cycle for the output word. In the first cycle, bits 127 down to 62 of the buffer are chosen for the first 66-bit block. In the second cycle, bits 125 down to 60 are read out for the second 66-bit block. This pattern continues until bits 65 down to 0 are read out. The reason for this sliding window is because the gearbox has to make up for the 2 extra bits that are shifted out from the buffer than are shifted in by adjusting the expected position of the next block.

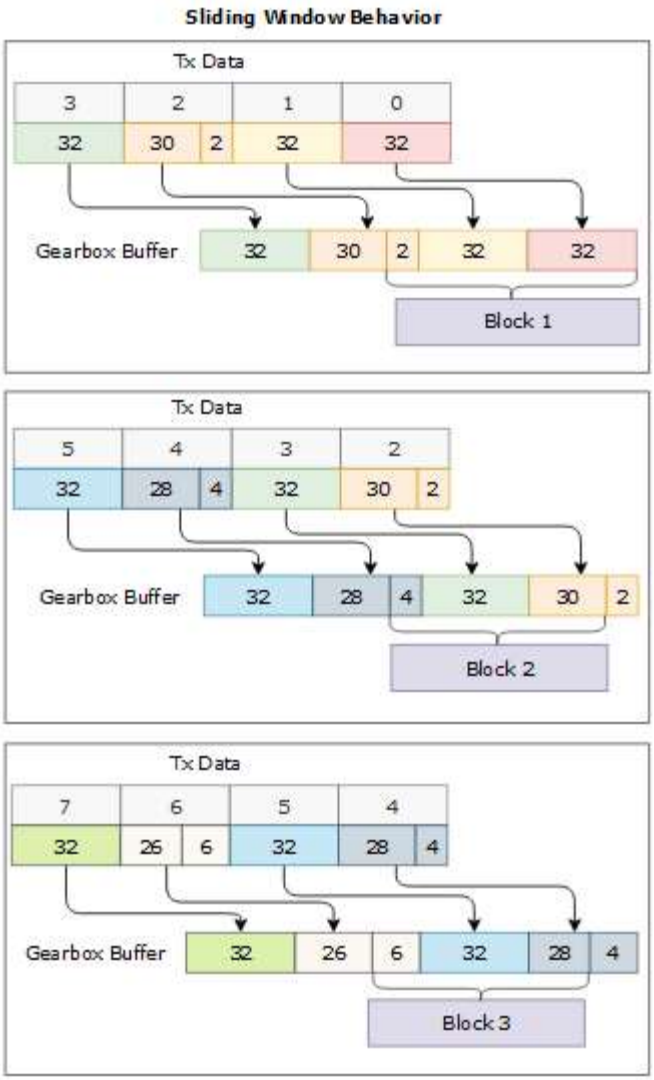

Figure 17: Cumulative effect of the reading and writing has 2 residual bits and a sliding window to accommodate them.

Every 33rd cycle there is a pause, and no 66-bit block is produced. Once the sliding window reaches bits 65 down to 0, there is no further it can go, and the gearbox has to wait for the buffer to fill up again through data being shifted in from the serializer. After the pause, the sliding window is reset to its initial position at bits 127 down to 62, where another 32 66-bit blocks are produced before the next pause.

3.3.3 Rx Descrambler

Of the 66-bit block produced by the gearbox, the two most significant bits (the header) are checked for validity, while the remaining 64 scrambled bits are passed through the descrambler. The Rx descrambler is the reverse of the Tx scrambler and therefore has identical properties to it. It is multiplicative and self-synchronizing and utilizes the same polynomial, $1 + x^{39} + x^{58}$, to derive the unscrambled data. A smaller example descrambler is shown in the figure below.

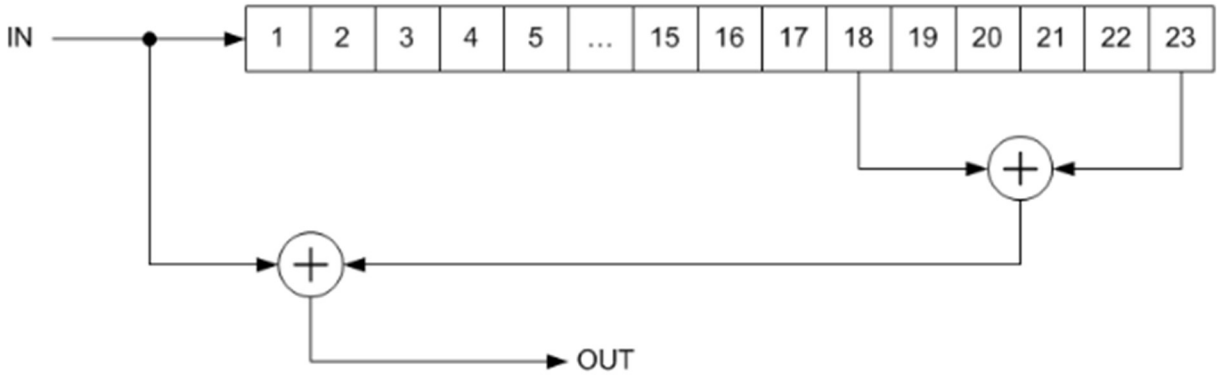

Figure 18: Example 23-bit descrambler. [6]

4.0 YARR's Original Detection and Resynchronization Scheme

Built into the Rx logic of the YARR is a set of logic to verify and maintain stream alignment. Both the RD53 Pixel Readout and YARR send and receive data using the same shared clock. Communication between the two takes a fixed but unknown delay, but since the clock is shared the sending and sampling are expected to happen at the same rate. All that the YARR receiver must do then is search and find the packet boundary in the incoming stream. This boundary is set by the header bits and is differentiated from the remaining data by always containing a 0 to 1 or 1 to 0 transition. Since blocks are 66 bits wide, the receiver checks a pair of bits periodically for every 66 streamed through to ensure that the same bit indices of each block are checked. If the bits checked consistently contain a transition, then the boundary is found, the sender and receiver are synchronized, and the receiver will correctly distinguish blocks within the stream. However, if the bits checked in some blocks do not contain a transition, i.e. they are from the data field rather than the header, then the YARR needs to change the block bit indices it is periodically checking to search elsewhere. This is accomplished by shifting the stream data, effectively dropping or duplicating a bit in the stream, so as to change the indices being periodically checked. With enough shifts, the YARR will be able to scan through all 66 possible bit indices.

In order to shift the stream data, the Rx logic utilizes a control logic FSM to allocate the shifts of the stream by leveraging features of the deserializer and the gearbox. The FSM also determines when synchronization is obtained or lost.

4.1 Serializer Bitslip

The deserializer in the YARR DAQ is made up of a series of Xilinx primitives, including the ISERDESE2. Built into this primitive is a "bitslip module" whose original purpose was to lock onto a repeating pattern, called a training pattern, to ensure that the deserializer was ordering the received bits in the correct orientation. Since the slip effectively drops bits, the alignment control unit leverages this feature to shift the position in which the header bits appear. To keep up with the data rate required, the deserializer is operated in DDR mode, which has a somewhat complex bit slipping behavior. The bit slipping model can be depicted as a 15-bit buffer from which an 8-bit window determines the data that is read out.

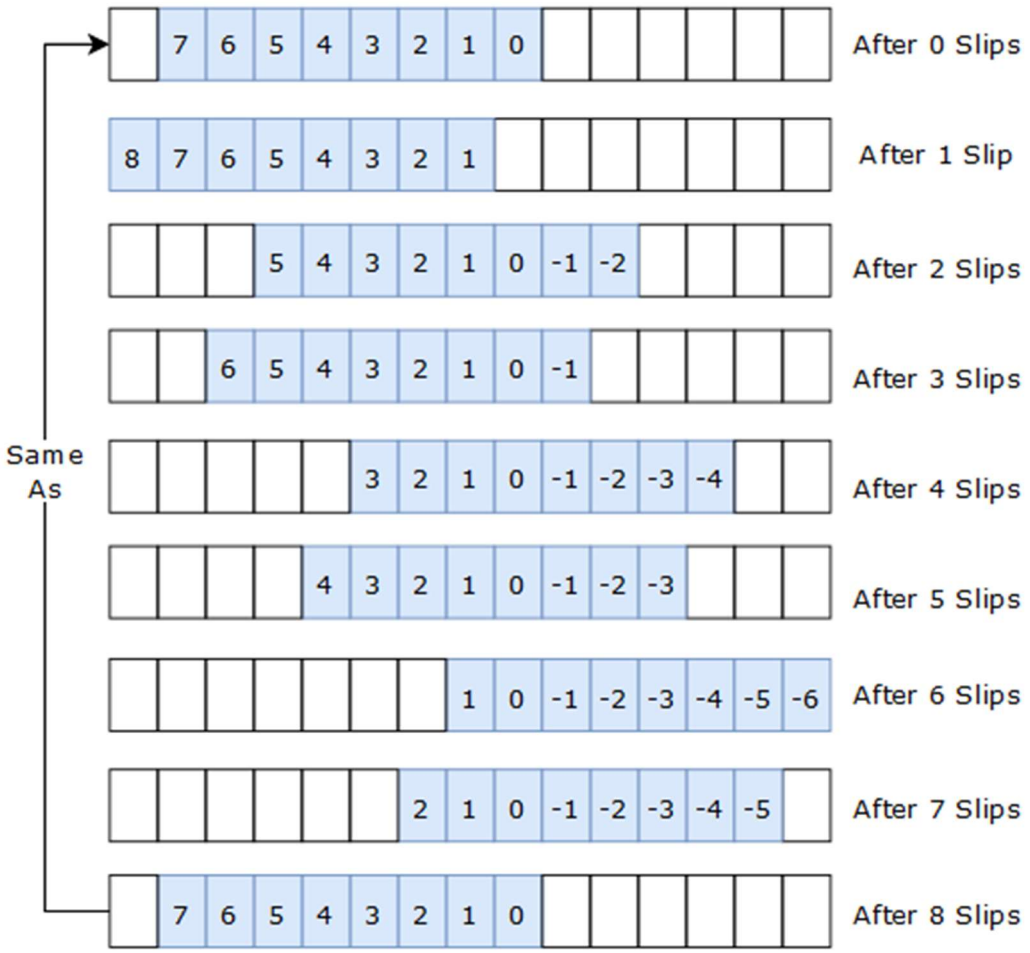

Figure 19: Slipping behavior of the DDR deserializer. Bits -6 through 8 are effectively buffered.

Since the buffer available to the deserializer is limited in size, at most 7 bits can be dropped, and 8 header positions can be evaluated. Once all 8 are seen, the deserializer snaps back into its original position. Despite the back-and-forth slipping behavior of the DDR bit slip, the bit slipping generally shifts the data stream in a single direction, which, as we will show later, results in alignment being recovered more quickly for SEE-induced stream bit drops over stream bit adds.

4.2 Gearbox Slip

8 bits of range is inadequate with block sizes of 66 bits, and so a second mechanism for shifting the data stream 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 deserializer. The gearbox uses a sliding window which it uses to select 66 bits out of its buffer to produce a block. By allowing the alignment control FSM to actively move this window, the entire 128-bit buffer can be scanned

over a series of cycles. Since the gearbox block selection window moves 2 bits each cycle to keep up with the uneven input out and output shifting, the control FSM can just freeze the window in place and prevent it from sliding. This has the equivalent effect of shifting the entire stream by two bits. For example, if the gearbox is incorrectly selecting bits 127 down to 62 in the first cycle, then it expects to incorrectly select bits 125 down to 60 in the second. However, if the gearbox instead selects bits 127 down to 62 in the second cycle, the gearbox will have moved the expected position of the header bits over by two bits. The gearbox "slip" is then accomplished by freezing the window's position.

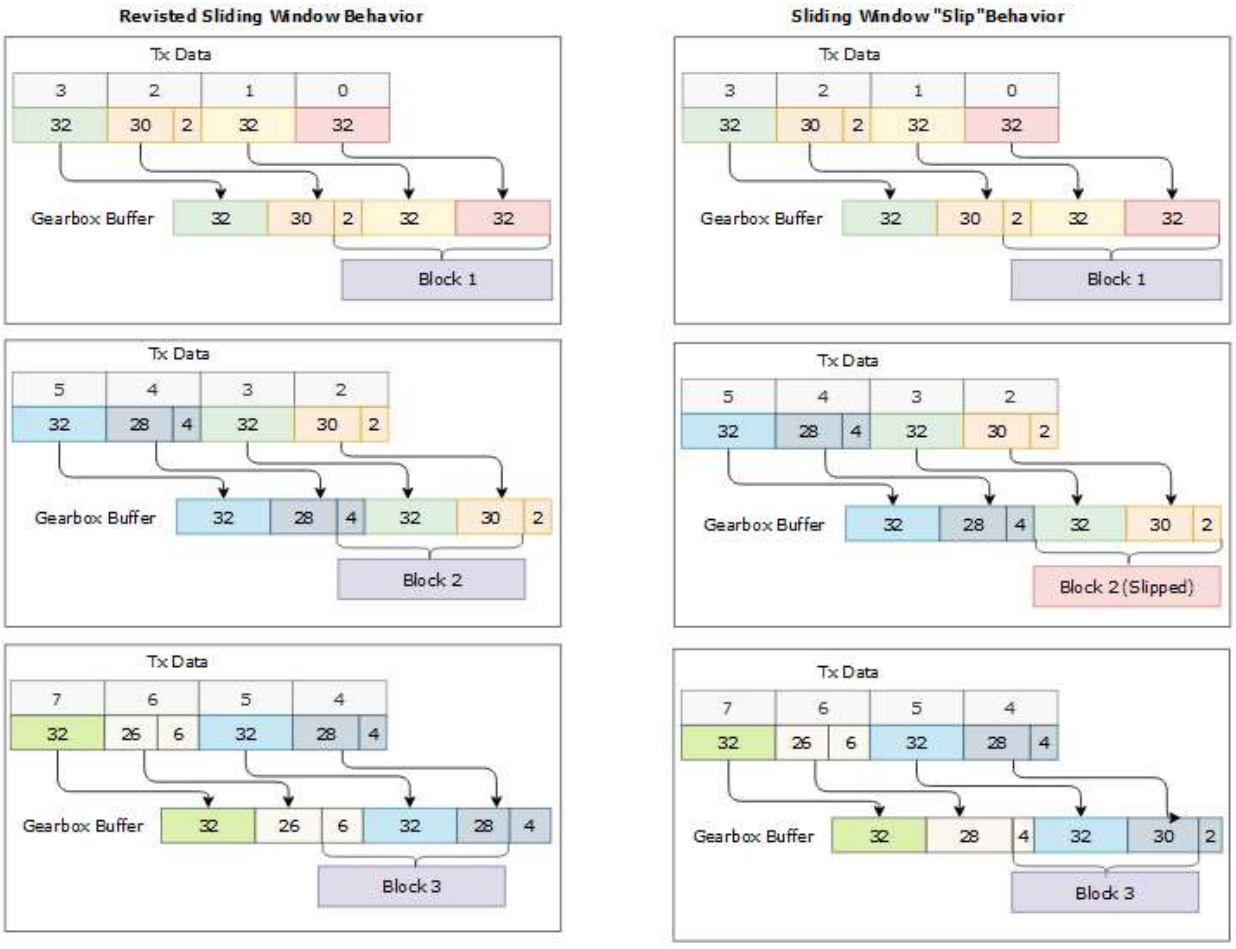

Figure 20: Gearbox's bit "slip" freezes the sliding window resulting in an effective 2-bit shift.

The gearbox slip has the benefit of having an essentially unlimited bit range it can search, given the time. However, it suffers from only being able to shift the data stream in a single direction, favoring the recovery from SEE-induced stream bit drops. It also shifts the stream by two bits at a time, meaning that alone it would be unable to recover from an SEE-induced single bit added or dropped.

4.3 Control FSMs

The detection and resynchronization are driven by a pair of control FSMs. While the serializer and gearbox can do the realignment, the FSMs determine when realignment is necessary as well as which of the two should perform a slip. The FSMs each take one of the two roles and are differentiated as the synchronization FSM and the slip FSM.

The synchronization FSM represents a three-state realignment process. The first state, Deadtime, is where the system starts at initialization. This state does not accept data blocks but instead counts them until 16 have been seen, at which point the FSM moves into the synchronizing state. The synchronizing state starts with a count of valid headers seen from the previous 16 blocks and continues counting subsequent valid blocks. If any invalid header is seen prior to a count of 32, a slip is performed, and the FSM moves back into the deadtime state. If instead 32 valid blocks are counted the FSM moves into the Locked state, where the payload of the blocks are finally accepted. However, if an invalid block is seen in the Locked state the FSM will restart the cycle and move back to the Deadtime state.

The synchronization system represents the synchronization requirement and the desynchronization tolerance. This FSM has effectively no tolerance in the Synchronizing and Locked states, as just a single invalid header would reset the synchronization process. The synchronization requirement is more complicated and guards against accidentally valid blocks being accepted as data. Since an invalid block has a 50% chance of passing off as a valid block, requiring consecutive valid blocks decreases this risk exponentially. Due to how valid headers are counted, we can determine that once in the Synchronizing state, we'd need a minimum of 16 consecutive valid blocks to reach the Locked state. This corresponds with a maximum $(50\%)^{16} = 0.0015\%$ probability of having a sequence of invalid blocks result in locking in on and accepting invalid data.

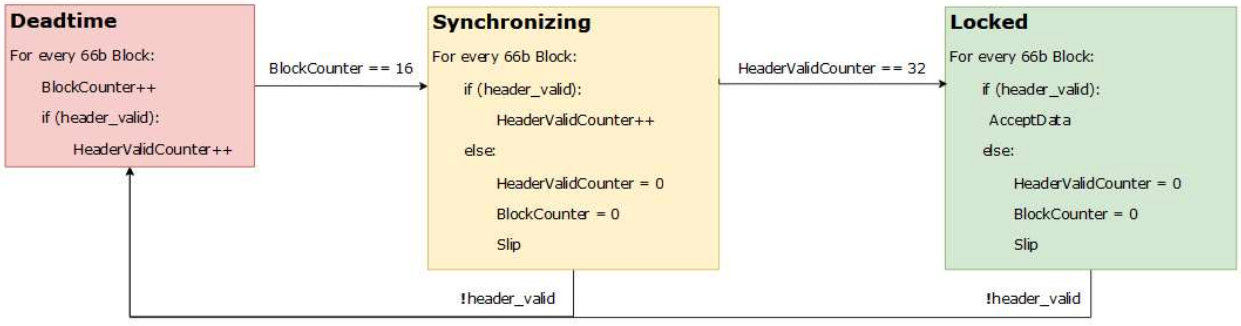

Figure 21: High-level synchronization FSM.

The slip FSM represents the slip allocation between the gearbox and serializer. This two-state FSM allocates the number and order of the slips. Since the serializer's bitslip snaps back to its original position after 8 slips, the FSM has 8 serializer slips executed prior to each gearbox slip. Since the gearbox slip has an effective shift of two bits, and the serializer's slips are equivalent to a shift of one bit the combined slipping of the two suffers some redundancy.

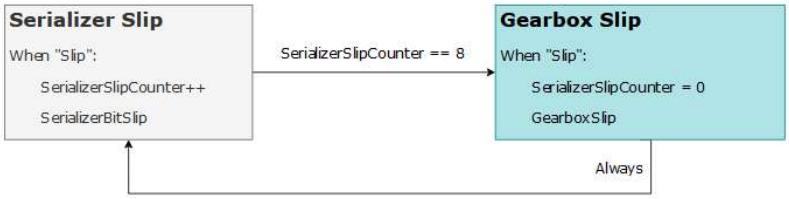

Figure 22: High-level slipping FSM.

This section details the abstract highlights of the FSM systems. The source code for the two FSMs is in appendix B if additional technical detail interests the reader.

4.4 Original Scheme Performance Evaluation and Results:

Evaluating the scheme comes down to determining the data lost as a result of interrupts, and the speed of recovery from these errors. As discussed previously, SEE handling is described by the rate at which SEEs occur (FIT), the average time between them (MTBF), and the average time for which a system is in recovery (MDT). Of the three, the first two are set by the pixel readout chip which is the source of errors. The MDT, mean downtime, is however set by the response of the DAQ to an erroneous block, i.e., the resynchronization scheme. However, rather than evaluating the amount of time spent in resynchronization, which changes with the clock frequency, we measure by the number of blocks lost during resynchronization, which otherwise would have reached the DAQ and been processed. This value is portable to other systems which might have different clock frequencies, use a different size for their blocks or change other parameters of the system. It also has a much more practical implication for the consequences of an SEE occurring.

The performance of the resynchronization scheme is evaluated by the response to a misalignment of the header position. The header is expected to appear every 66 bits, but by dropping bits of a preceding block the subsequent blocks' header bits will be effectively shifted out of their expected position and into the preceding block's data field. Sweeping the number of bits by which the header is shifted from 1 to 65 will allow the resynch time in blocks lost for every possible instance to be evaluated and graphed.

SEEs can be categorized as bit flips, bit drops, and bit adds it's important the simulation evaluates the response to appropriate effects. Since the simulation performs the effective shift by

dropping bits, the response to bit drops is exactly as it appears in the simulation results. Bit adds similarly perform a shift to the header location, however, it is in the opposite direction to bit drops. Fortunately, the stream is periodic, and shifting the header a single bit in one direction is equivalent to a shift of 65 in the other. While this is effectively true, adding bits guarantees that there will be no header at all in a 66-bit block which generally results in a single more block lost during resynchronization as compared to bit drops.

Bit flips are a more complex issue. In 64/66 bit flip cases (when the flips occur in the data field) only the corrupt block and the block following it will be lost as a result of the corruption and the error lingering for a cycle in the scrambler. The system is not desynchronized because the effect does not shift the header bits out of alignment. In the 2/66 cases (when the header bits are flipped) the lack of tolerance to invalid headers forces desynchronization and the system will begin the resync process. Since adding tolerance harms the recovery performance from bit adds and drops, the system assumed no tolerance from the beginning and the likelihood of bit adds and drops against bit flips is unknown, the remainder of the thesis will focus on the evaluation and improvement of recovery from bit adds and drops.

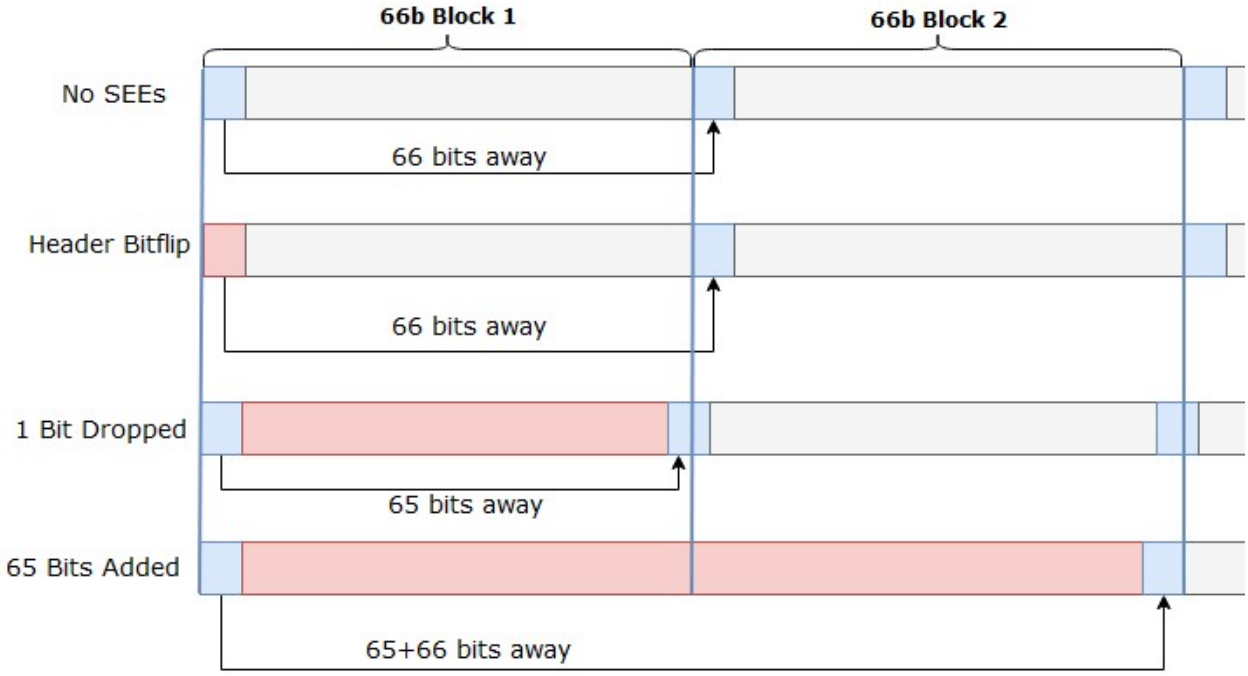

Figure 23: Drops and adds equally represent shifts but in opposite directions. Header bitflips don't misalign the header and are equal to 66-bit drops

A bit drop simulation was developed to evaluate the response time of the resync scheme following a block where *n* bits of a block are removed. This value *n* is then swept from 1 to 65 to

determine the resynchronization duration for any number of bits dropped or added. Although bits can be dropped from or added to any point within the block, the new position of the header bits is the same regardless of the specific bits duplicated or removed in the corrupt block. The only difference is whether the corrupt block is evaluated as valid or not by the system. In this simulation, the corrupted block is always considered lost, whether or not the DAQ system would have treated it as a valid block. This sweep is done purely in simulation utilizing Siemens Modelsim, an HDL simulation tool. Each value of n is evaluated 25 times and the average is used as the final value. The number of blocks sent was tracked with a 32-bit counter and each block's payload was set to the concatenation of two copies of the counter. This made determining how many and which blocks were dropped very easy. Having two duplicates of the same data in the data field also served as a verification method to distinguish blocks received that were invalid but had seemingly valid header bits from valid blocks. Since the gearbox effectively randomizes the data, the results should be just as accurate as if realistic data were sent. However, to be certain, random seeds for the starting count were used as well. For all performance simulations, the effective number of consecutive valid blocks necessary is 16, so the number of blocks it takes for the scheme to detect an error and determine the correct position is 16 less than the number of blocks it took to recover synchronization.

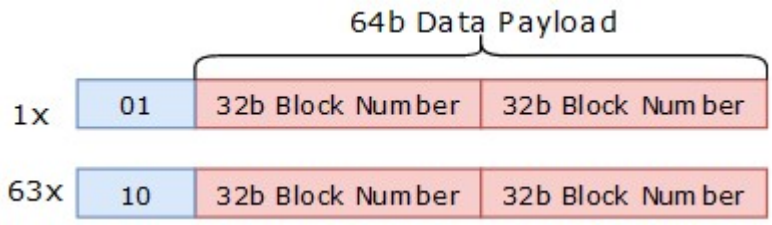

Figure 24: Contents and frequency of blocks used in the simulation.

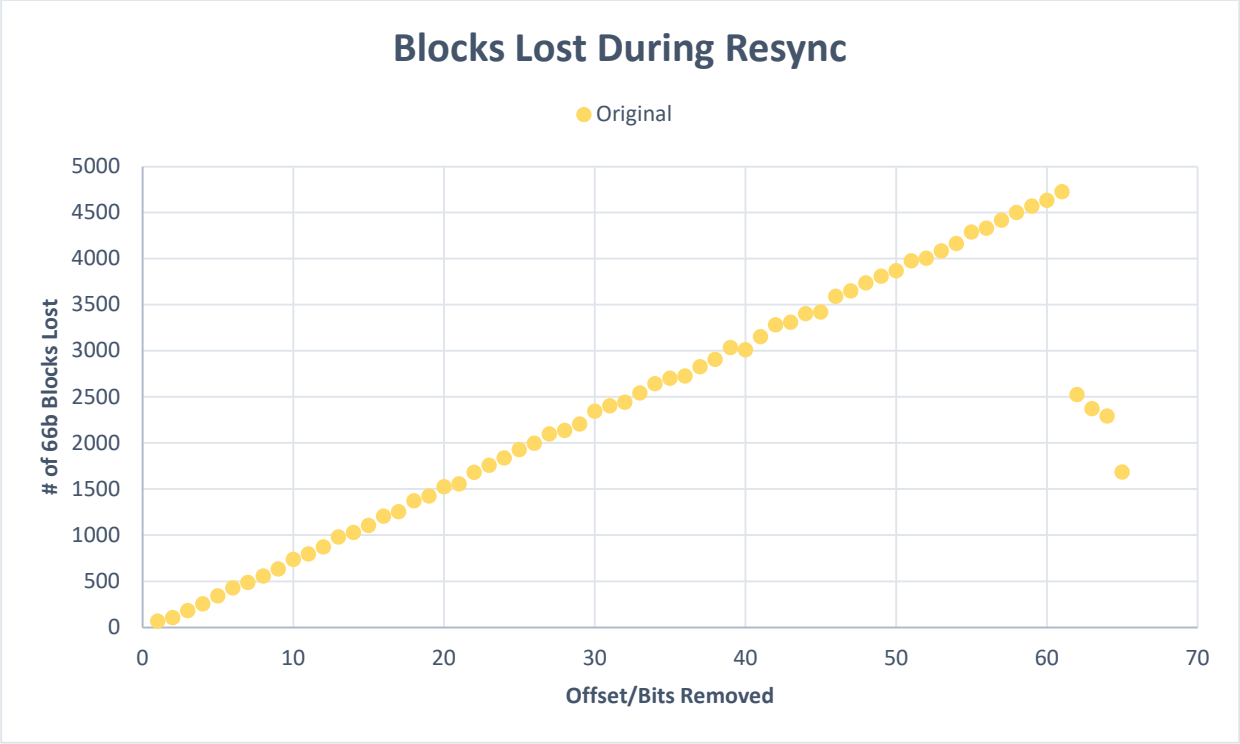

Figure 25: Performance results for the original resynchronization scheme.

The number of blocks lost while realigning the header bits under the original scheme is a linear trend against the number of bits dropped. This should come as no surprise, this scheme examines a single position at a time and checks position mostly unidirectionally. The exception is in bits 62 through 65, which are a consequence of the serializer's bit slipping pattern. The nonuniformity or noise in the graph above comes down to random factors with scrambling, where an invalid block may pass off for valid for several cycles and is ultimately a random variable. The noise is reduced due to the large number of samples at each offset taken (25) but it is still evident. The amount of noise is expected to increase the more bits are examined and that appears to be expressed in the above.

The primary measure of the cost was the resource utilization of the channel when targetting a Xilinx Kintex T160 FPGA. The primary constraints of the YARR DAQ design are the resources utilized as well as a 6.25 ns clock period. By targeting a Xilinx FPGA and using the Vivado design software passing timing constraints can be verified and resource utilization reports for different designs can be obtained and compared. The Kintex T160 FPGA was chosen because it is a likely candidate for the YARR DAQ used at the LHC. Since the resync scheme doesn't use any hard silicon resources, the comparison will be made purely with LUT and FF utilization.

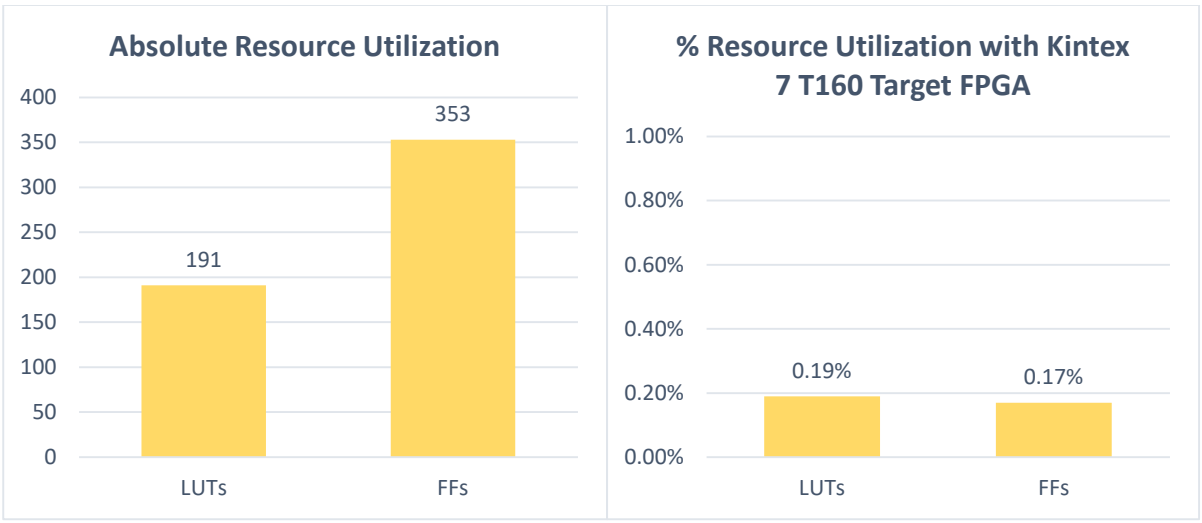

Figure 26: Relative (left) and absolute (right) resource utilization for the original resync scheme.

The resource utilization of the original system sets the bar for comparison against other resync schemes. The absolute resource utilization quantifies exactly the amount of LUTs and FFs used for the YARR RX channel and is more portable to other FPGAs, however, there may be variance if the FPGAs are produced through a different vender, are configured using different software, or are simply a different generation within a vendor and use different primitives. The % utilization is a more intuitive quantification of the resources used but is specific to only the Kintex 7 T160 FPGA. Both above include the resource utilization for an entire channel rather than the logic directly related to the resync scheme because the logic for resynchronization is distributed across multiple components.

5.0 Proposed Detection and Resynchronization Scheme

The performance of the original recovery scheme leaves a lot to be desired. The slipping mechanisms utilized are complex and result in unnecessary overhead. The bit slipping allocation between the gearbox and serializer has redundancies. The individual header positions are checked for too long, and the criteria to gain and lose synchronization are ineffective. A replacement recovery system developed improves on or removes each of these flaws and attains large performance boosts.

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 the real cost. Any new scheme would need to simultaneously maintain a low chip footprint and operate with a clock period of 6.25 ns.

5.1 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 mechanisms (the gearbox and the serializer's bitslip module), to shift the position of the data stream as seen by the DAQ. Since the DAQ only monitors 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 actually 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 by up to 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. Resynchronization would only be limited by the time it takes to detect that an SEE occurred and 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 in memory as the original scheme's 128-bit gearbox buffer stores just 1 possible header position in its worst-case state.

To track every possible header position, the replacement scheme augments the gearbox buffer to allocate 67 bits for the evaluation of 66 possible header positions. The 67-bit requirement is derived from the periodicity of the stream. Since the header is in the top two bits of every 66-bit block, tracking 68 consecutive bits would result in two valid headers being monitored simultaneously. To avoid sorting out which to treat as the correct header at least one fewer bit must be tracked so that only one valid header is monitored at a time, resulting in 67 bits tracked. The

same result can be reached when considering the worst-case possibility of a header shift of 65 bits. If a block were to drop 65 bits, or equivalently gain 1 bit, through SEEs the header position will have shifted 65 bits over. Since the replacement scheme will need to be able to monitor the two bits where the header is expected to be, as well as up to 65 bits over, a total of 67 bits need to be monitored.

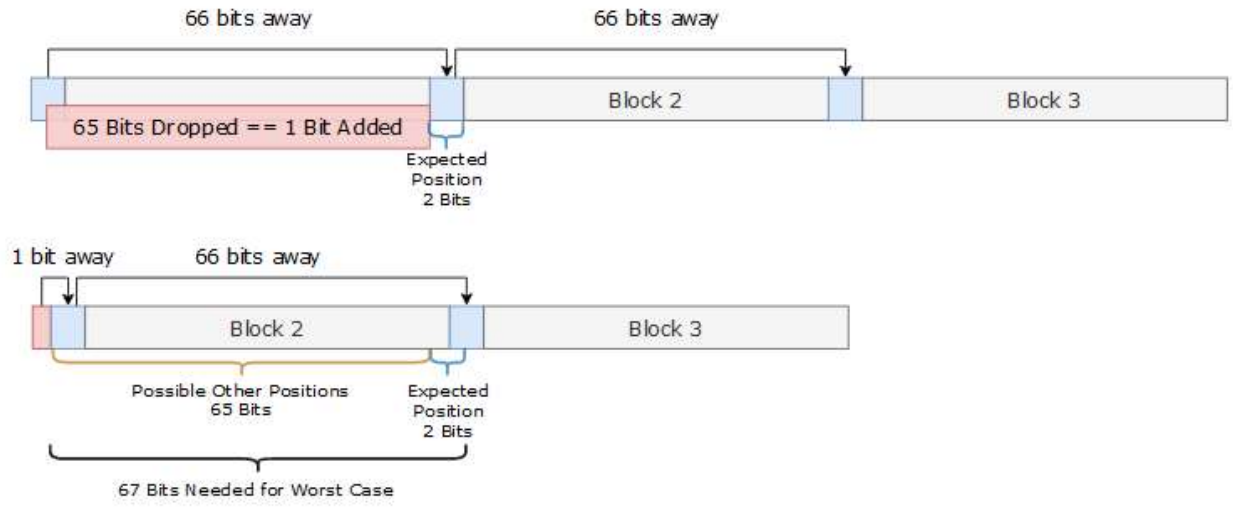

Figure 27: Worst-case SEE scenario requiring 67 bits monitored to avoid needing slips.

By tracking every bit where the header might appear, the serializer and gearbox can be simplified to their primary purposes. Bit slipping is no longer necessary because the header bits are guaranteed to appear in one of the monitored positions in the cycles following an SEE. 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.

A 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, and the pipelining costs associated, result in an enormous number of resources utilized, which might violate 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.

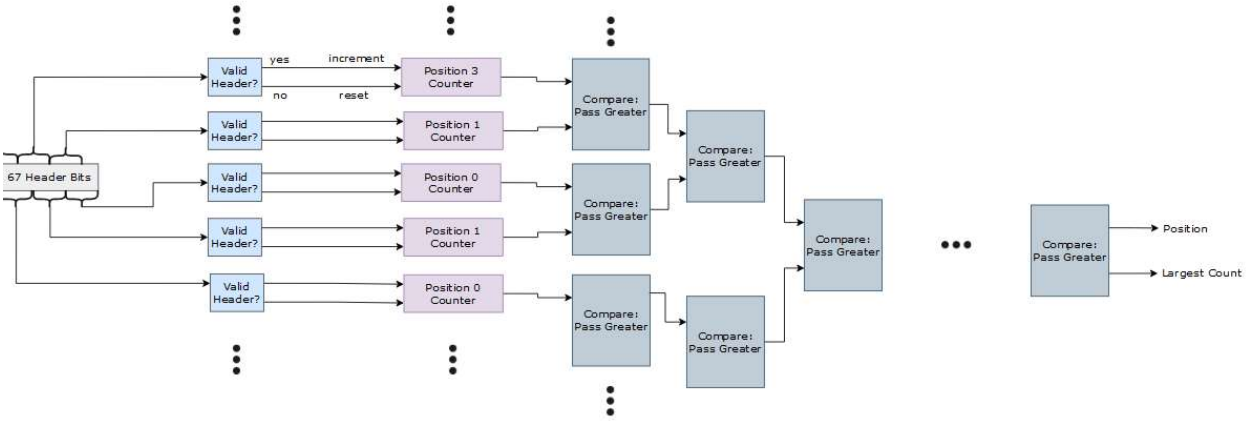

Figure 28: Block diagram of a parallel monitor/seeker aligner.

5.2 Seeker Aligner

The header seeker aligner utilizes fail-fast principles to minimize hardware costs while maintaining similar performance. The fully Parallel Search (PS) implementation described earlier will very quickly determine that 65 of the 66 positions evaluated are invalid and that continuing to monitor those positions is wasteful. If a position being monitored is determined to be invalid, the monitor or seeker can ignore it and move on to a different position. By utilizing a few seekers simultaneously, all 66 positions can be evaluated with significantly fewer resources.

Any factor of 66 could have been used for the seeker count, however, 11 seekers provided the best performance-cost product among those tested. Since 66 positions encompass 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 22, 11, 6, and 3 as they appeared to have the best balance of performance seen against resources utilized. However, depending on the requirements of the application, any one of them can be utilized and results for all are derived.

This header seeker 11 aligner (HS11) starts by resembling a partially parallel header search, with 11 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 as an invalid header is seen, the seeker changes the position it is evaluating. By distributing the 66 positions possible between the 11 seekers, each seeker rotates between 6 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 in any given block. However, the

probability of seeing multiple accidentally valid headers drops exponentially and we can reasonably expect that a seeker will not get stuck at any incorrect position for more than a few blocks. Further, the seekers don't permanently ignore any position, rather they rotate through a predefined set of them. So, if multiple blocks were affected by a prolonged SEE or consecutive blocks had bit drops, the seeker aligner system would inevitably recover from them.

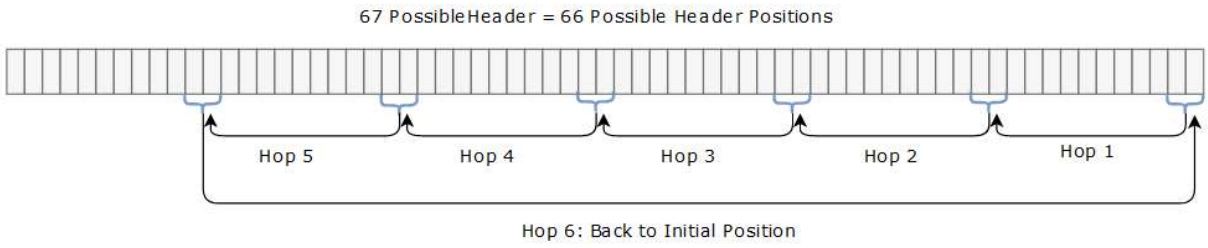

Figure 29: Positions and rotation of seeker 0 in the HS11 resync scheme.

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 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 small regardless of the SEE. That said, the fail fast methodology would be less effective with tolerance, and is why this system is much more effective at recovering from bit adds and drops as compared to bit slips.

With bit slipping eliminated, 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, improved the maintainability and testability of the logic. The remaining purpose of the block sync FSM is to determine whether the YARR 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 labeled 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 detecting the SEE and determining the correct header position.

5.3 Proposed Scheme Performance and Results

The performance improvement made from the original resynch scheme to any of the proposed replacement variants is drastic, and a comparison of the two across the same sweep is shown below.

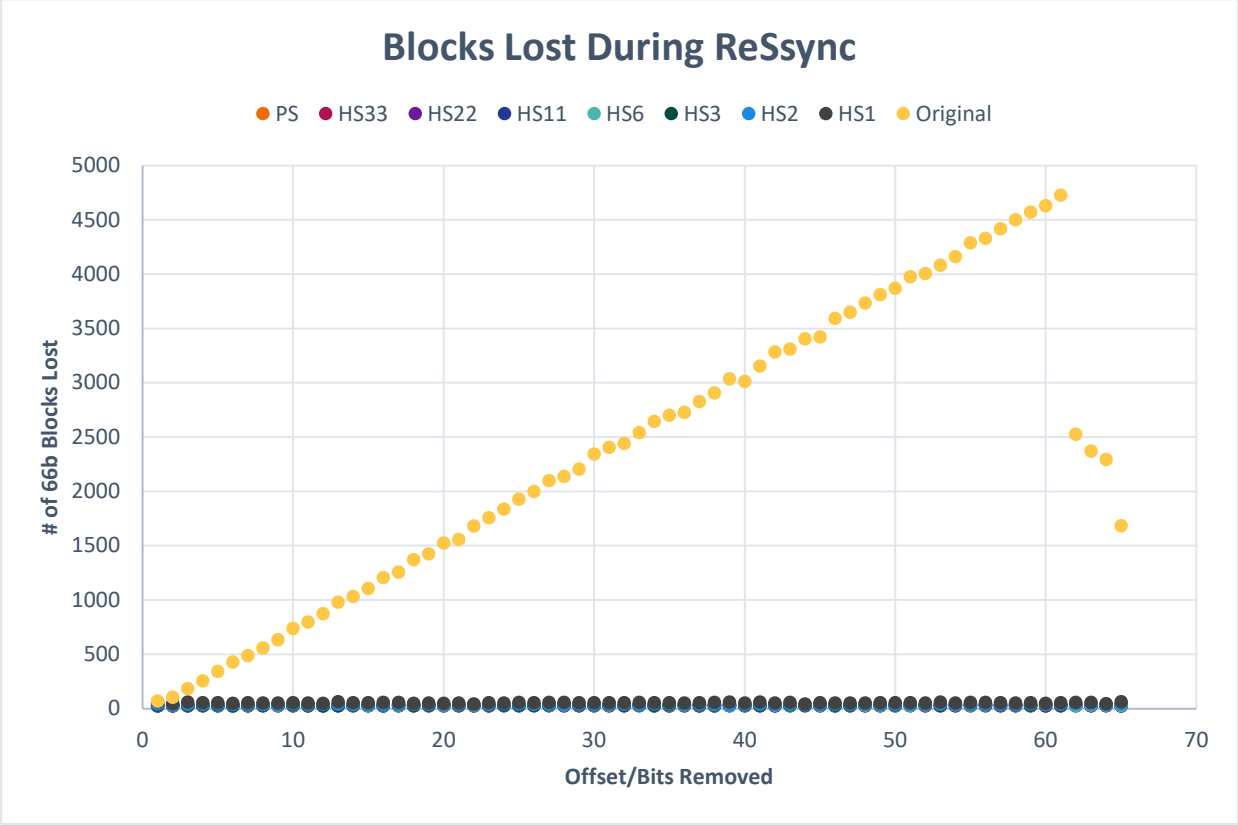

Figure 30: Direct performance comparison of the original to all proposed resync schemes.

While the direct comparison is not especially informative of the header seeker resynch schemes, it demonstrates just how effective the fail-fast and parallel search are to recovery performance. A separate performance graph for the header seeker schemes is shown below. Further comparisons will only include the original system when it is applicable or relevant and will be noted.

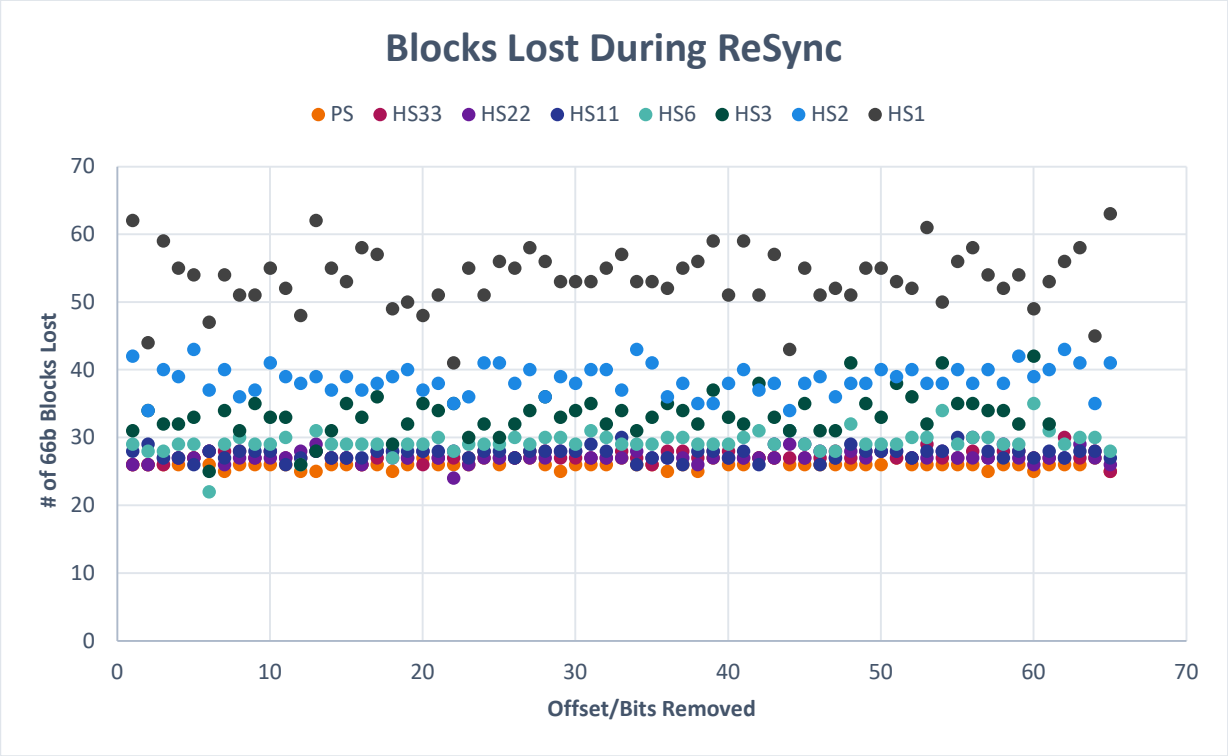

Figure 31: Performance comparison of PS and HSX schemes.

Above are the sweep results of HSX variants, X representing the number of seekers utilized, as well as the full Parallel Search (PS) implementation. Just like in the original system's case, each offset is the average of 25 samples. Barring noise due to random effects from the scrambler, each sweep resembles a flat line with the average getting progressively smaller as more parallelism is utilized. Below is a graph with the unweighted average for each across all offsets, rounded to the nearest 0.5.

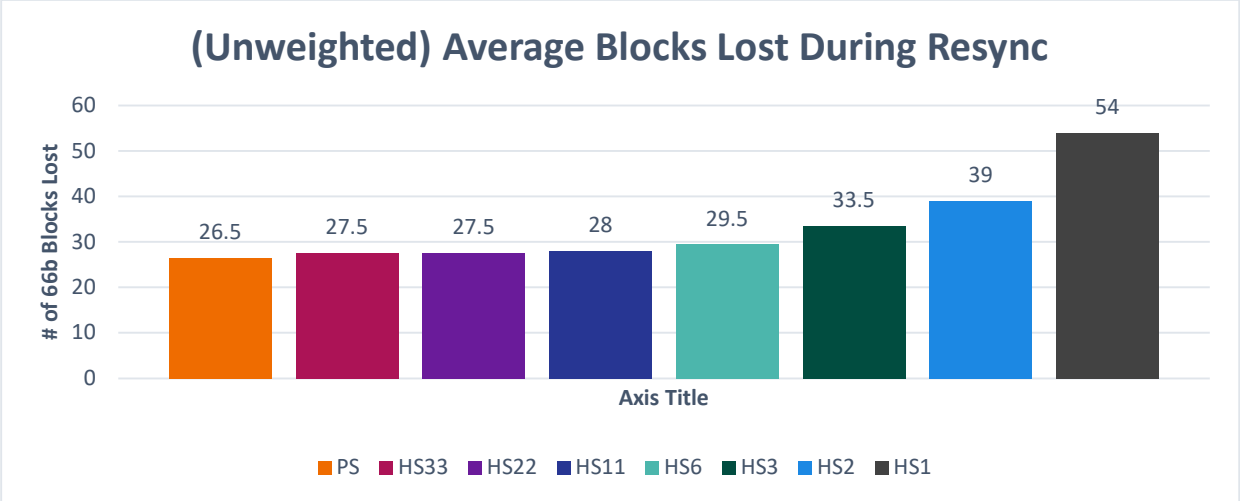

Figure 32: Average (unweighted) performance across all offsets for the PS and HSX schemes.

The header seeker recovery systems differ in both scale and shape. The first and most obvious difference, illustrated in the comparison graph, is that the proposed replacement scheme operates with roughly 2 orders of magnitude fewer blocks dropped during resync. The primary reason for this is that the SYNC MAX constant is now a constant added to the blocks lost rather than a multiplier applied to the number of positions searched. The characteristics of each sweep can be roughly described by the formulas:

$$Orig\ Blks\ Lost = (SYNC_{MAX} + AccidentalValid_i) * PositionsSearched + AccidentalValid \quad (1)$$

$$HSX\ Blks\ Lost = AccidentalValid_i * PositionsSearched_{by\ seeker} + SYNC_{MAX} \quad (2)$$

AccidentalValid is a random variable for each position rooted in the possibility of multiple consecutive invalid blocks being labeled as valid. For the original system, this results in a delay in discovering that an SEE occurred and is added as a constant in formula 1. Then they also appear as a factor because each position searched can have a sequence of accidentally correct header values. In the header seeker implementations, the aligner is constantly searching all bit positions whether the system is considered in sync or not, which erases this term from the constants added. However, this is unavoidable when evaluating a position for whether it's a header or not and so it still factors in as a multiplicative term with the number of positions searched before finding the correct one.

The primary difference as briefly discussed earlier, is how *SYNC_MAX* factors into each equation. In the original system, it's a multiplicative term because it is applied to every position searched. Removing this term from the product as in formula 1 and adding it back as just a constant as in formula 2 is the greatest contributor to the performance of the HSX schemes.

PositionsSeached also plays in even though it appears to be identical in each equation. For the original system, the worst-case value is 66. Being clever about what positions are searched first and last can result in the vast majority of headers being aligned within 7 positions checked and is accomplished by starting the search around bit 63 rather than 0 so that the neighboring 3 bits are checked first, which is where the vast majority of misalignments will be. In the HSX schemes, *PositionsSearch* has a worst case of (66/# of seekers), which is brought on by parallelism. This factor of X improvement contributes significantly to the performance, but once again is overshadowed by the fail-fast principle.

The trend, having accounted for the noise, resembles a flat line. This might seem unexpected, considering we don't have full parallelism in HSX schemes, and the seekers have to sweep through 11 positions the same way the original system swept through 66. However, the difference is really the result of the more passive nature of the HSX recovery schemes. In the original system, recovery only started when the system was found to be desynchronized and stopped when synchronization was achieved again. In HS6 however, the recovery scheme is

always operating and so the positions being evaluated at the moment synchronization is lost are effectively random.

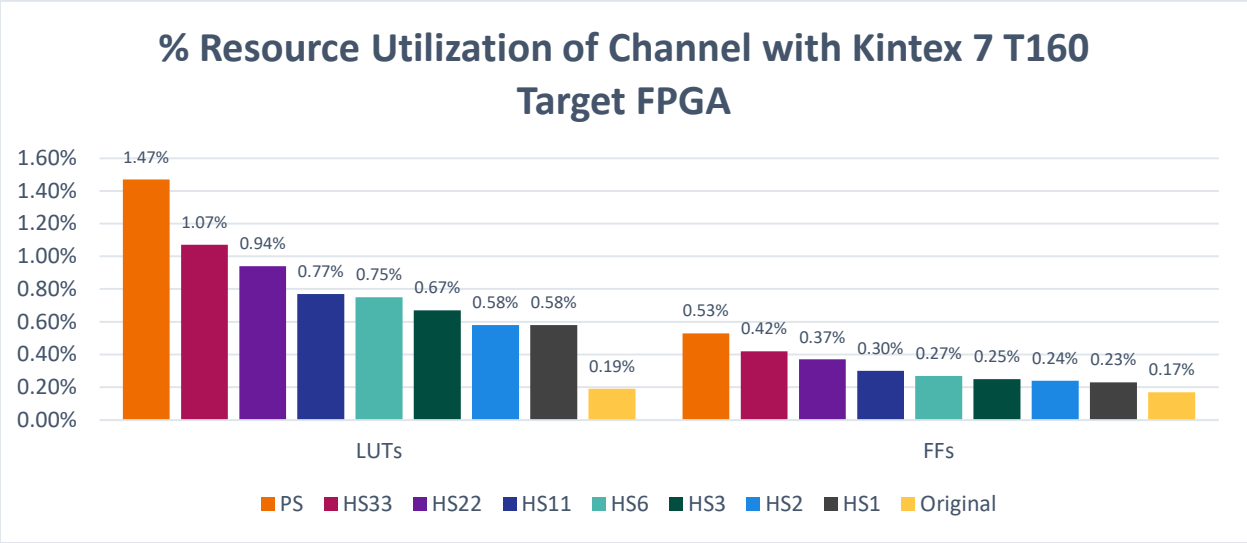

Figure 33: LUT and FF utilization of a single Rx channel with the original and proposed resync schemes.

The resource utilization of the HSX schemes is considerable relative to the original system, however, its performance improvements dwarf this cost. The graph above compares the % of device resources utilized by a channel across all schemes considered, including the original. The schemes have at least triple the LUT resources utilized compared to the original system.

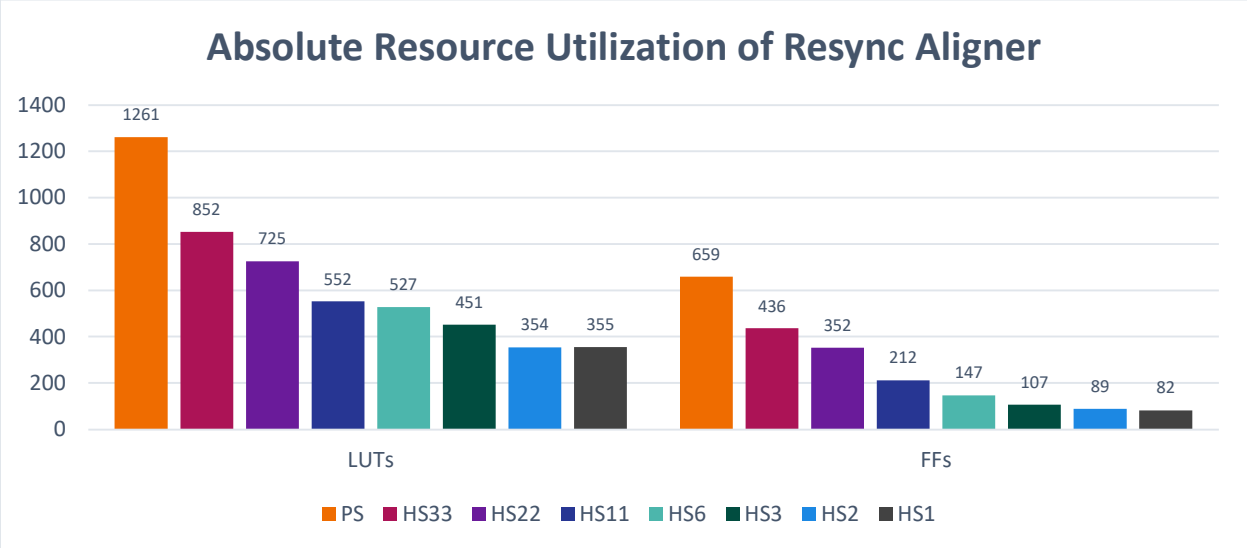

Figure 34: Absolute LUT and FF utilization of PS and HSX resync schemes.

Since the original system distributes its resynchronization logic throughout the channel blocks the absolute resource utilization can't be compared to PS and HSX variations. They can, however, be compared amongst themselves since the only difference between them is the aligner block and logic associated.

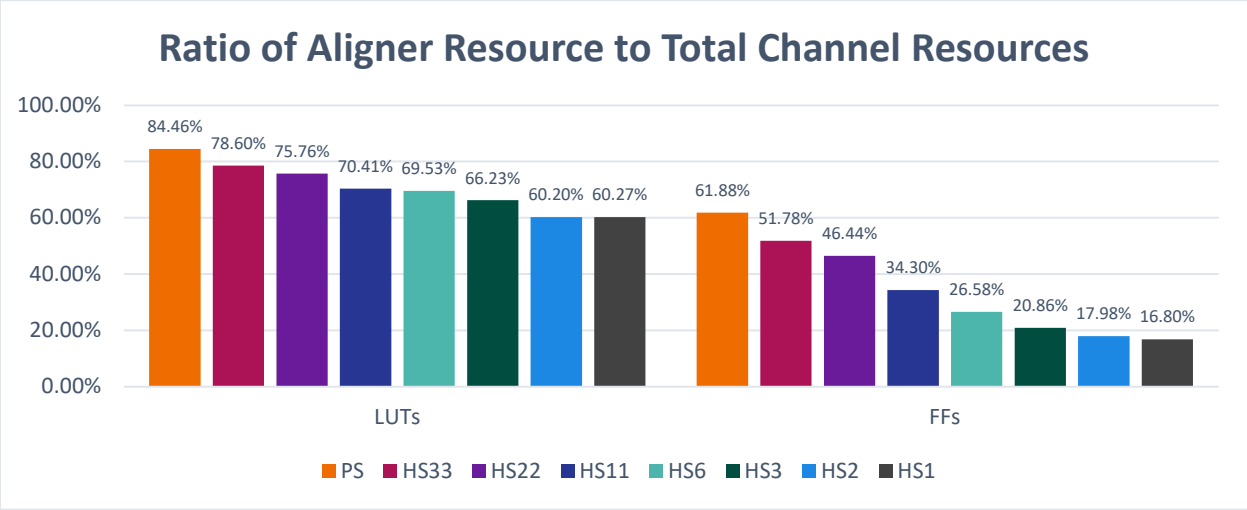

Figure 35: Ratio of aligner resources to total channel resources for all resync schemes.

An additional interesting metric is the % of resources in the channel dedicated to the aligner in each scheme. While the FF resource has a steady increase, it's clear that the comparisons, counter logic, and position decoding take up a significant number of combinatorial resources regardless of the scheme. Once again, since the resources for the original system were distributed across multiple blocks, this value could not be effectively determined for comparison.

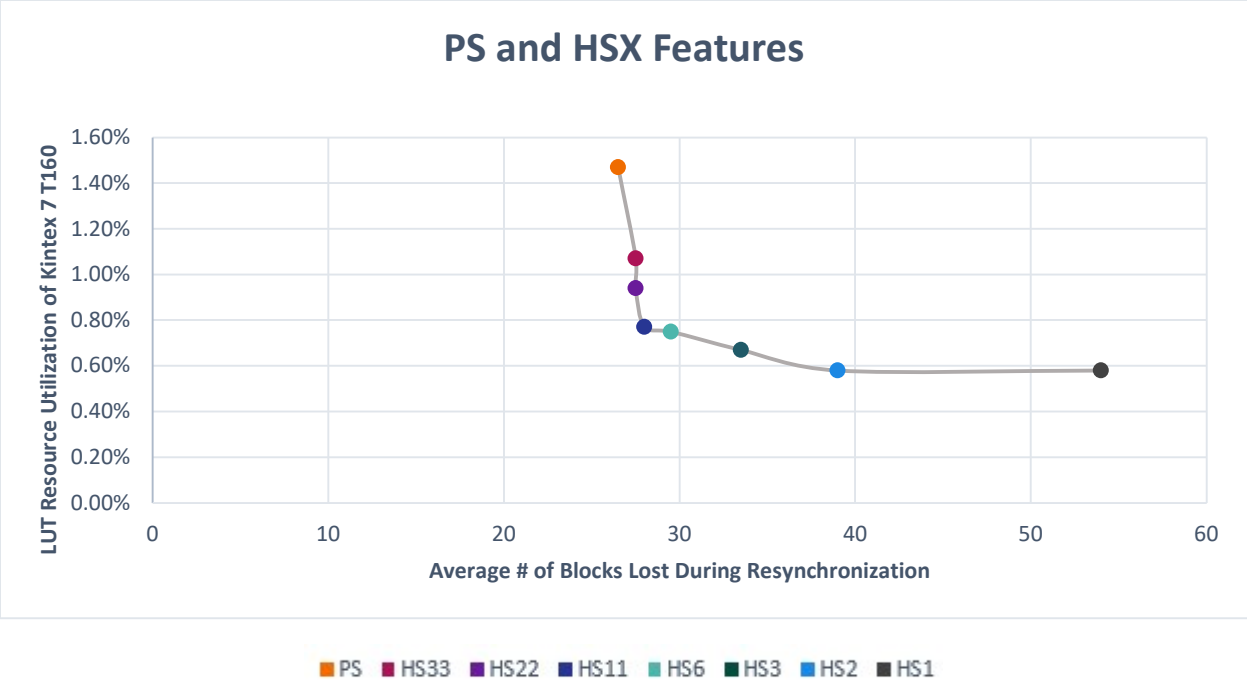

Figure 36: Summary of average blocks lost to resources utilized for PS and HSX aligner schemes.

Finally, a condensed comparison of the performance and cost of PS and the HSX variations is given above. The best bang for buck variants are HS22, HS11, and HS6. HS33 offers practically no benefit for the additional cost as does PS. HS3, HS2, and HS1 offer meager reductions in resource utilization for progressively increasing costs in performance.

6.0 Conclusion

In anticipation of the upcoming HL-LHC upgrade, hardware and firmware are being upgraded to support the increase in luminosity. While the collision points are being upgraded to support the increase in data resulting from the upgrade, luminosity is correlated to radiation experienced and an increase in single event effects is expected. While the RD53 pixel readout chip will be the device experiencing these single event effects, it is anticipated to forward corrupted data in the form of bit flips, adds, and drops within its data block to DAQs such as the YARR. This thesis investigated the current recovery response scheme built into the YARR DAQ and proposes a replacement system to improve performance by reducing time in the form of blocks lost while unsynchronized.

Synchronization, or header alignment, is done by shifting the data stream so that two prefix header bits of each block appear where the RX expects to see them every 66 bits. When the headers move relative to the expected position, synchronization is lost, and the recovery system starts to search for them. The original system leveraged two bit slipping mechanisms and allocated their execution through an FSM to search for the header bits one position at a time. A set of replacement schemes referred to as Parallel Search (PS) and Header Seeker X (HSX) significantly outperform the original system for only a moderate increase in resources. These variations offer tradeoffs in both performance and resource utilization.

The performance and cost of each recovery scheme were evaluated through simulation and FPGA mapping tools. When targeting a Kintex 7 T160 FPGA the original system had a mean down time of 2406.5 blocks lost and utilized 0.19% of LUT resources for every operating channel. HS11, one of the best bang for buck schemes, targeting the same FPGA has a mean down time of only 28 blocks at the cost of 0.77% of LUT resources. While this increase in resources appears to be significant, in a four-channel DAQ only 3.08% of resources would be dedicated to receiving data and maintaining synchronization. These and other results serve to allow developers to find the best fit scheme for their design.

7.0 Future Work

As with most designs, the HSX alignment and recovery schemes are only a base from which more improvements can be developed, and additional tradeoffs can be made.

A pair of parameters lightly discussed but not justified are the tolerance and synchronization requirements of the system. In this design the tolerance for the original system and HSX is 0, meaning desynchronization occurs immediately after an invalid header is seen. The synchronization requirement is SYNC_MAX, which sets the required consecutive valid headers seen to consider the design synchronized and allow data blocks to pass further into the DAQ. Both parameters are rooted in statistics involving the likelihood of an incorrect header position seeing multiple accidentally correct values in a row. A deeper evaluation of the statistics could result in an improved value for each parameter, depending on the application. This is a non-trivial relative improvement for both schemes presented and comes at a negligible cost in hardware resources.

Both the previous improvements and multiple HSX variants could be unified into a single parameterizable alignment unit. The logical similarities between each header seeker aligner variation are close enough that the design could relatively straightforwardly be transformed to be easily configurable before being synthesized and loaded. This would allow the design to be simplified and easily portable to different DAQs and apply to their various constraints and tradeoff preferences.

The alignment schemes discussed and proposed are biased towards bit drops and adds. This results in suboptimal performance if it were bit flips that made up the majority of SEEs. Both additional data on the frequencies of these events as well as a resynchronization system better suited for bit flips may be valuable.

8.0 Acknowledgements

I would like to thank all the brilliant people I have worked without throughout these years who led to this project as well as the help I received throughout it.

I would first and foremost like to thank Professor Scott Hauck and Professor Shih-Chieh Hsu as my advisors and mentors throughout my entire time at UW and especially during this thesis project. Scott Hauck was a professor of mine who through a single course convinced me not only to completely change the concentration of my Electrical Engineering degree but also to join a lab and strive for a master's. I had never expected to enjoy a subject as much as I did hardware design and am very fortunate to have fought my way into his competitive class in my very first quarter at UW. I remember on my first day I was even offered money for my seat in it. I offer my apologies to Shih-Chieh who I had first assumed was a student like me in the lab. His expertise in the physics and logistics should have tipped me off much sooner and I was always impressed as physics was a favorite subject of mine in my lower-division coursework. I also really appreciate the team they put together and continue to evolve which is the ACME Lab.

I would also like to thank Timon Heim, who has been an enormous help in this thesis project. He was the one who first proposed this project and has also been my go-to source for answering a myriad of questions and resolving confusion as I studied and developed the project. I also appreciate the resources and opportunities that were always available through him.

I would like to thank each person in the ACME Lab, past and present, who guided me with relatable experiences and student and industry perspectives. Special thanks go to Geoff Jones, who ultimately convinced me that VHDL is the superior language and has unintentionally imparted a significant amount of my FPGA knowledge through our various conversations. I'd also like to thank my predecessors and colleagues, past and present, Donovan Erickson, Matt Trahms, Lauren Choquer, Amelia Dumovic, and Sanjukta Roychoudhury who I had close contact with throughout my time at ACME Lab and have been great friends.

Finally, I want to give thanks to my friends and family who have motivated me to succeed on a relatively unconventional path and drove me to do my best. It means the world to hear that my relatively unconventional path is held in such high regard by you all. It's through your encouragement, support, and patience that I am able to accomplish what I have.

References

[1] ”CERN Website”, CERN, [Online], Available at https://home.cern/

[2] “CERN Atlas Website”, CERN, [Online], Available at https://atlas.cern/discover/detector

[3] McMahon, Stephen; Pontecorvo, Ludovico, “Technical Design Report for the ATLAS Inner Tracker Strip Detector”, CERN, April 1, 2017, [Online], Available at https://cds.cern.ch/record/2257755?ln=en

[4] “High Luminosty LHC Project”, CERN, [Online], Available at https://hilumilhc.web.cern.ch

[5] “The RD53B-ATLAS Pixel Readout Chip Manual v2.18”, CERN-RD53-PUB, January 2022

[6] “FPGA Development of an Emulator Framework and a High Speed I/O Core for the ITk Pixel Upgrade”, Lev S. Kurilenko, S. C. Hsu, S. Hauck, 2018

[7] “Development of an FPGA Emulator for the RD53B Chip”, N Mittal, S. C. Hsu, S. Hauck, 2020

[8] “Neutron-Induced Single Event Upset (SEU) FAQ”, Microsemi, August 2011

[9] “Single Event Effects (SEEs) in FPGAs ASICs, and Processors, Part I”, D. White, Xilinx, [Online] Available at https://www.eetimes.com/single-event-effects-sees-in-fpgas-asics-and-processors-part-i-impact-and-analysis/

[10] “Single Event Effects (SEEs) in FPGAs ASICs, and Processors, Part II”, D. White, Xilinx, [Online] Available at https://www.eetimes.com/single-event-effects-sees-in-fpgas-asics-and-processors-part-ii-mitigation/

[11] “Mitigating Single-Event Upsets”, J. Hussein, G. Swift, Xilinx, May 2015

[12] “Considerations Surround Single Event Effects in FPGAs, ASICs, and Processors”, D. White, Xilinx, March 2012

[13] “Introduction to Single-Event Upsets”, Altera, 2013

[14] “SEE Summary”, J. Lalic, M. Menouni, 2022, [Slide Deck], Available at https://indico.cern.ch/event/1145919

[15] “Aurora 64B/66B Protocol Specification v1.3”, Xilinx, October 2014, [Online], Available at https://docs.xilinx.com/v/u/en-US/aurora_64b66b_protocol_spec_sp011

[16] “Custom “SEE Tolerant” Aurora Gearbox”, A. Martynyuk, 2022, [Slide Deck], Available at https://indico.cern.ch/event/1156885/

9.0 Appendix

Appendix A: 64b/66b Protocol

In data networking and transmission applications, data from a source can rarely be transmitted exactly as it appears in the source to the destination. Whether it is for security reasons or much more often for practical reasons the data has to be modified and wrapped in additional information the receiver can interpret. Line encodings are algorithmic transformations that offer a series of features at various costs which must be accommodated by both the sender and receiver. Often the technical implementations of the protocol are done in hardware, and the software of the source and destination are unaware of the protocol being used.

64b/66b encoding is a line code that transforms 64 bits of information into a 66-bit line code. It provides enough transitions after this transformation to allow features such as clock recovery and stream alignment. It also benefits from low overhead, with just 2 coding bits for every 64 payload bits. This is an improvement from its predecessor the 8b/10b encoding but has also since been expanded on with 128b/130b encoding.

The transformation from 64 bits to 66 bits comes in two parts. First, a 2-bit prefix is added to the 64 bits payload. If the prefix is 01, then the 64 bits payload is data, and if the prefix is 10 the 64 bits payload holds an 8-bit type field and 56 bits of control data. Prefixes 00 and 11 are not used and indicate an error if they are seen on the receiver. Once the data has been labeled, the 64 bits are scrambled. The scrambling doesn't encrypt the data but attempts to make an even number of 1s and 0s within the payload of the block. The scrambling does not, however, guarantee an even number of 1s and 0s but rather is statistically likely to have an even number over multiple blocks transmitted.

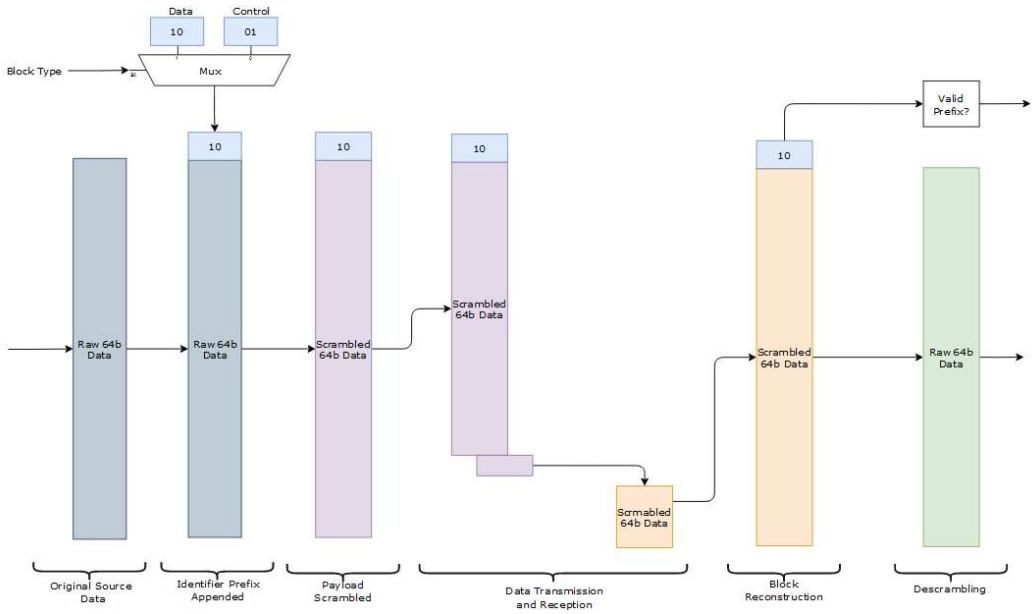

Figure 37: 64b/66b protocol.

Appendix B: Original System Control FSMs Source Code

```
block_sync_proc: process(clk_rx_i, rst_n_i)
    begin
          -- on a reset, clear all counters
        if (rst_n_i = '0') then
            sync_cnt <= (others => '0');
            slip_cnt <= (others => '0');
            serdes_slip <= '0';
            valid_cnt <= (others => '0');
            scrambled_data66 <= (others => '0');
            scrambled_data_valid <= '0';
            gearbox_slip <= '0';
            serdes_phase_adj_en <= '1';

        elsif rising_edge(clk_rx_i) then
            -- set default values
             serdes_slip <= '0';
            scrambled_data_valid <= '0';

             -- remaining behavior only happens for each 66b block
            if (gearbox_data66_valid = '1') then
                -- set default value, done here so that it changes only every 66b block
                gearbox_slip <= '0';

                -- increment for EVERY 66b block up to saturation (16)
                if (valid_cnt < c_VALID_WAIT) then
                    valid_cnt <= valid_cnt + 1;
                end if;

                -- increment for every VALID 66b block up to saturation (32)
                if ((gearbox_data66(65 downto 64) = c_DATA_HEADER) or
                    (gearbox_data66(65 downto 64) = c_CMD_HEADER)) then
                    if (sync_cnt < c_SYNC_MAX) then
                        sync_cnt <= sync_cnt + 1;
                    end if;

                -- slipping and counter resets can only occur after 16 blocks (valid or not)
                elsif (valid_cnt = c_VALID_WAIT) then
                        -- reset counters
                    sync_cnt <= (others => '0');
                       valid_cnt <= (others => '0');

                       -- perform a gearbox slip for every 8 serdes slips
                    if (slip_cnt = c_SLIP_SERDES_MAX) then
                        gearbox_slip <= '1';
                        serdes_slip <= '0';
                        slip_cnt <= (others => '0');
                    else
                        serdes_slip <= '1';
                        slip_cnt <= slip_cnt + 1;
                    end if;


                end if;

                -- Once synced, the data is passed on to the descrambler
                serdes_phase_adj_en <= '1';
                if (sync_cnt = c_SYNC_MAX) then
                    scrambled_data66 <= gearbox_data66(65 downto 0);
                    scrambled_data_valid <= '1';
                    serdes_phase_adj_en <= '0'; -- Disable phase adjustment once locked
                end if;
            end if;
        end if;
    end process block_sync_proc;
```