© Copyright 2019

Douglas George Smith

# FPGA Development of an Emulator of the RD53A Prototype Chip and its Integration with Various Readout Systems

Douglas George Smith

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical and Computer Engineering

University of Washington

2019

Committee:

Scott Hauck

Shih-Chieh Hsu

Program Authorized to Offer Degree:

Department of Electrical and Computer Engineering

University of Washington

# Abstract

FPGA Development of an Emulator of the RD53A Prototype Chip and its Integration with Various Readout Systems

Douglas George Smith

Chair of Supervisory Committee:

Scott Hauck

Department of Electrical and Computer Engineering

In 2024 the Large Hadron Collider will be shut down for a major upgrade of the particle detectors and collider systems that will increase the number of collisions occurring in the LHC. As part of this upgrade the front-end particle detectors will be replaced with the RD53 chip [6], which combines the analog pixels which detect the particles with digital control, data processing, and readout systems that control the chips behavior. The RD53 project has released a prototype version of the chip, the RD53A, and has begun the process of designing its successor, the RD53B. As part of the upgrade new readout systems are being designed that can handle the new data rates as well as communicating with the RD53A. To assist with these efforts, the Adaptive Computing Machines and Emulators (ACME) lab has designed an FPGA based emulator of the chip in Verilog. The emulator is built to produce realistic hit data and can be used as a substitute for the real RD53A chip for testing/debugging purposes. More recently work has been done on the emulator to make it compatible with several of the prominent readout systems being developed for the coming upgrade. These include YARR [8], RCE [9], and FELIX [10]. The necessary background, the current state of the emulator, and the work done on it regarding the listed readout systems are discussed in this thesis.

**Table of Content**

1: The Large Hadron Collider - 1

1.1: What is the Large Hadron Collider - 1

1.2: The Coming ITk Upgrade - 2

1.3: The RD53 Project - 3

1.4: RD53A - 4

2: The RD53A Emulator – 4

2.1: Motivation – 4

2.2: TTC Data Processing - 6

2.3: Command Processing - 8

2.4: Trigger Processing and Hit Data Generation – 11

2.5: Data Handler - 14

2.6: Data Framing - 16

2.7: Data Output - 17

3: RD53A Readout Systems - 17

3.1: YARR - 17

3.1.1: What is YARR – 17

3.1.2: YARR Hardware - 18

3.1.3: YARR Modifications for the Emulator - 20

3.2: RCE - 22

3.2.1: What is RCE – 22

3.2.2: RCE Hardware - 22

3.2.3: RCE Modifications for the Emulator - 23

3.3: FELIX - 24

3.3.1: What is FELIX – 24

3.3.2: FELIX Hardware - 25

3.3.3: FELIX Modifications for the Emulator - 26

4: Next Steps - 27

5: Acknowledgements – 27

6: References – 28

# 1: The Large Hadron Collider

## 1.1: What is the Large Hadron Collider

To understand the motivation behind the work discussed in this thesis some knowledge of the Large Hadron Collider and the coming upgrades to its performance is required. The Large Hadron Collider (LHC) is the largest particle accelerator in the world and is located at the European Organization for Nuclear Research (CERN) which is an international research center devoted to the study of high energy particle physics located on the France/Switzerland border. The initial purpose of the LHC was to help complete our understanding of the Standard Model of physics, which is the current and accepted model of particle physics and describes how the basic particles of the universe behave. The role of the LHC in completing this model is that the Standard Model predicted the existence of several particles including the Higgs Boson, but experimentally the Higgs Boson could not be proven to exist as the conditions needed to create it could not be accomplished with the technology and particle accelerators that existed. In order to determine if the Higgs Boson did exist a much larger particle accelerator than had ever been built previously was required [1]. The result after years of planning and construction, with numerous different nations contributing was the LHC. With it, physicists were able to create the necessary conditions to generate and detect Higgs Boson particles. With the standard model completed the LHC now focuses on the search for physical phenomena beyond the standard model called new physics. This is because there are still phenomena that are not fully explained by the Standard Model, so physicists are using the LHC to see if new particles can be found that explain these phenomena [2].

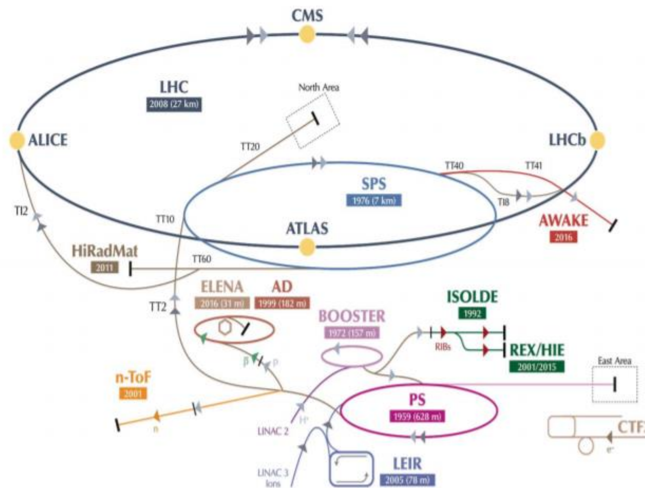


Fig. 1. An outline of the LHC particle accelerators showing the various stages used to accelerate particles up to near light speed and the location of the detector equipment [1].

The LHC is located at the border of France and Switzerland in a 27 km long tunnel. How the LHC works is that the particles (usually hydrogen but sometimes larger particles such as lead are used) are first ionized using electric fields and are then accelerated up to near light speeds, first in several smaller stages and then in the full LHC ring (see Figure 1) [1]. As the particles circle the LHC ring they do not move as a continuous stream but instead are grouped into clusters of particles called bunches. Once the bunches have reached maximum speed, they are then made to collide with bunches circling in the opposite direction at each of the four collision points where detector equipment is located. These are called ATLAS, CMS, LHCb, and ALICE. The collisions create an intense amount energy which in turn converts back into mass via Einstein's

mass-energy relation E= $mc^2$. This created mass can form as rare and/or possibly undiscovered particles which are what the physicists are interested in finding. This is where the detectors come in, as their purpose is to record what happens during collisions and send that data to the back-end systems where it can be translated into useful information for the physicists. Of the four collision points the one most relevant to this paper is ATLAS (A Toroidal LHC Apparatus, see Figure 2). It is one of the more general-purpose detection points and consists of many different layers of detectors, each of which has a different means of particle detection to help detect all the particles being generated in the collisions. Of these layers the one related to this work is the inner detector layer. This group layer sits closest to the beam and consists of the Transition Radiation Tracker (TRT), the Semiconductor Tracker (SCT), and the Pixel detector. Of these three layers the one closest to the beam, and most relevant to this paper, is the Pixel Detector [3].

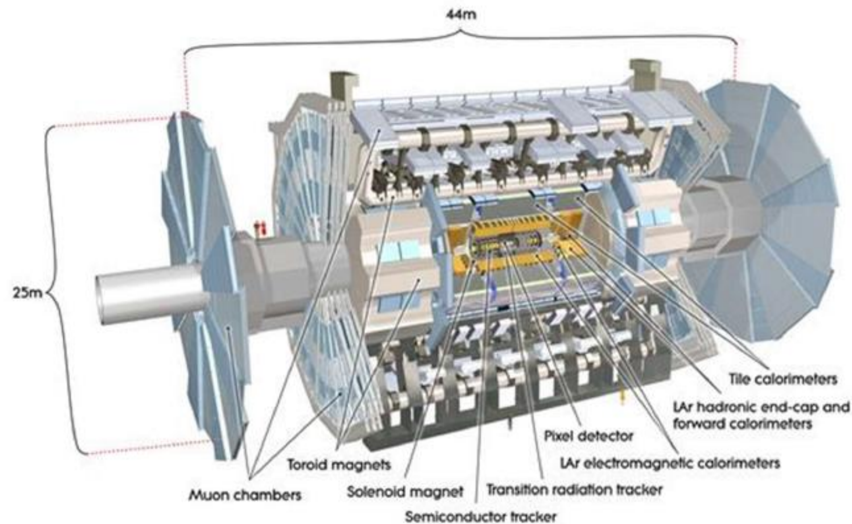


Fig. 2. An image of the ATLAS detector highlighting the dimensions and the different layers contained within the detector [3].

Being the closest layer to the particle beam, the Pixel Detector is the first part of ATLAS to observe the decay products of the collisions and as a result has the highest density of sensors. Overall this layer has over 80 million-pixel units, where pixels are the individual sensors used for particle detection [3]. How a pixel works is that when a particle passes through the pixel, charge is deposited on the pixel. The pixel converts this charge to a digital value and the length of time the digital charge has been over a threshold value is recorded. This measurement is called Time Over Threshold (ToT). The chips that contain the pixels are called front-end (FE) chips and the purpose of the FE chips is to collect the ToT values being generated by the pixels, do some processing on that data, and then send it up the readout chain for further processing. The Pixel Detector consists of four different overlapping layers of front-end chips. This overlap ensures that as particles pass through the detectors at least one sensor picks it up. The separate layers also make it possible to track the arc of particles as they pass through the detector [4]. The next generation of FE chips that make up these layers is called the RD53 and is what the emulator discussed in this paper is emulating.

## 1.2: The Coming ITk Upgrade

Occasionally the LHC is shut down to allow for maintenance and repair as well as performing upgrades to the system's sensors and infrastructure. In the year 2024 the LHC is going to have a several yearlong shutdown to allow for large amounts of the system to be upgraded. The reason for this upgrade is that after the shutdown is complete the luminosity of the particle beam is going to increase ten-fold as this will greatly

increase both the amount and intensity of collisions. This in turn will make it more likely to generate and observe interesting particles [5]. At the current luminosity the system already produces more than 60 terabytes of data per second that the system must filter and process. The massive increase in luminosity is going to make most of the current sensors and readout systems inadequate. As a result, a good chunk of the LHC needs to be replaced with equipment that can handle the upgrade. As part of this upgrade the entire inner detector layer will need to be upgraded and reconfigured which will in turn impact the pixel detector layer as the current three layers will be replaced with just two, ITk strips and ITk pixel detectors. In upgrading the pixel detectors, the entire readout chain will be affected and most of it will need to be replaced. The two major portions of the upgrade that this paper will focus on is the rework of the readout systems (the YARR, FELIX, and RCE projects) and the newly developed front-end pixel chip (RD53A).

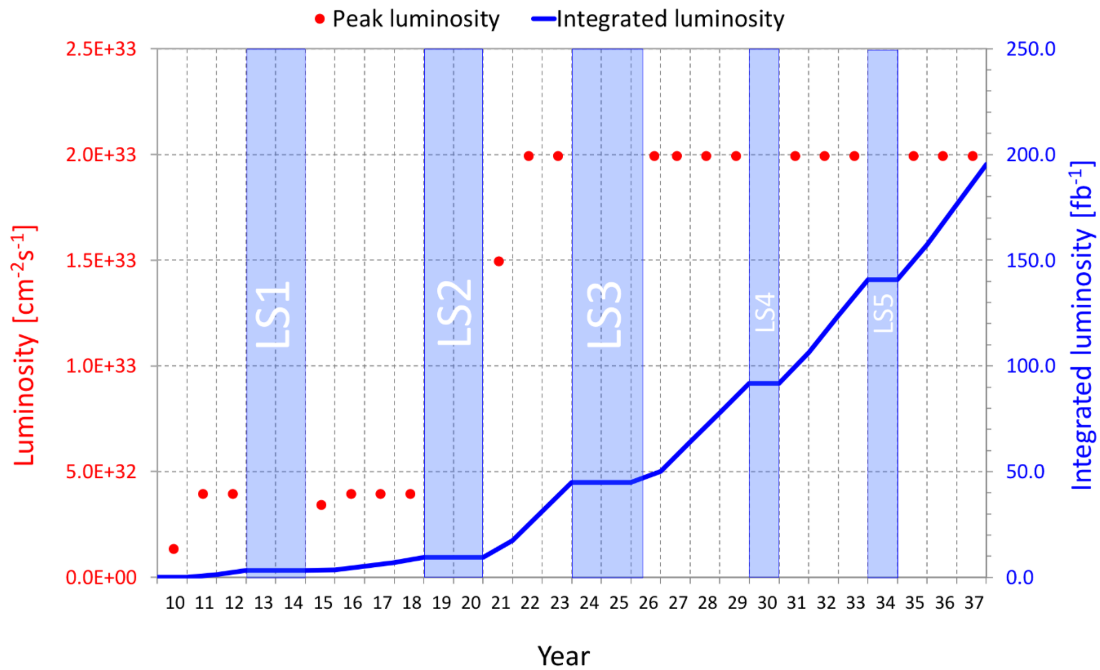


Fig. 3. An outline of the target luminosity per year and the time period of upgrades [5].

## 1.3: The RD53 project

Part of the effort to prepare for the coming ITk upgrade is the design of a new FE chip that will be able to handle the new levels of luminosity. The design of this chip is being managed by the RD53 collaboration group whose mission statement is "RD-53 will design and produce the next generation of readout chips for the ATLAS and CMS pixel detector upgrades at the HL-LHC." For the past few years the primary focus of the project has been on the development and testing of the RD53A test chip which contains all the primary features required from the front end pixel chip including bump bonding, test beams, irradiations with final pixel geometry sensors, high hit rate operation, high speed I/O, serial powered module chains, etc. The RD53A has been available since 2018. Due to the success of the project the RD53 collaboration is now working on the design of the RD53B test chip which will take the lessons learned from the RD53A to improve upon the design of the FE chip [6].

## 1.4: The RD53A

The RD53A chip was designed as a prototype chip to serve as both a test platform for different designs as well as a demonstration chip that the requirements of the latest generation of FE pixel chips could be met. A figure showing the layout of the chip can be seen below (Fig 4). The chip itself consists of 50 by 48 pixel cores, with each of these cores consisting of an 8 by 8 grid of pixels, resulting in a total of 76,800 pixels on the chip as a whole. While this is fewer pixels than the final production level chip will have, it was deemed suitable to be representative of the behavior seen in a large-scale chip and could be used to accurately validate the performance. The pixels on the RD53A consist of three different types labeled differential, linear, and synchronous. The three designs are vastly different from each other and allow for the testing of different models to observe their performance and determine the best model to use for the final production chip. The important performance metrics being compared include tolerance to radiation, especially important in this environment due to the high amounts of radiation from the particle collisions, as well as having low power consumption with acceptable noise and detection thresholds. Each pixel core is surrounded by what is referred to as a sea of digital logic which performs readout and control for that pixel core. The top part of the chip consists primarily of these pixel cores and the bottom of the chip, called chip bottom, is dedicated to high level digital and analog control as well as digital readout [7]. The next section discusses an FPGA emulator of this chip.

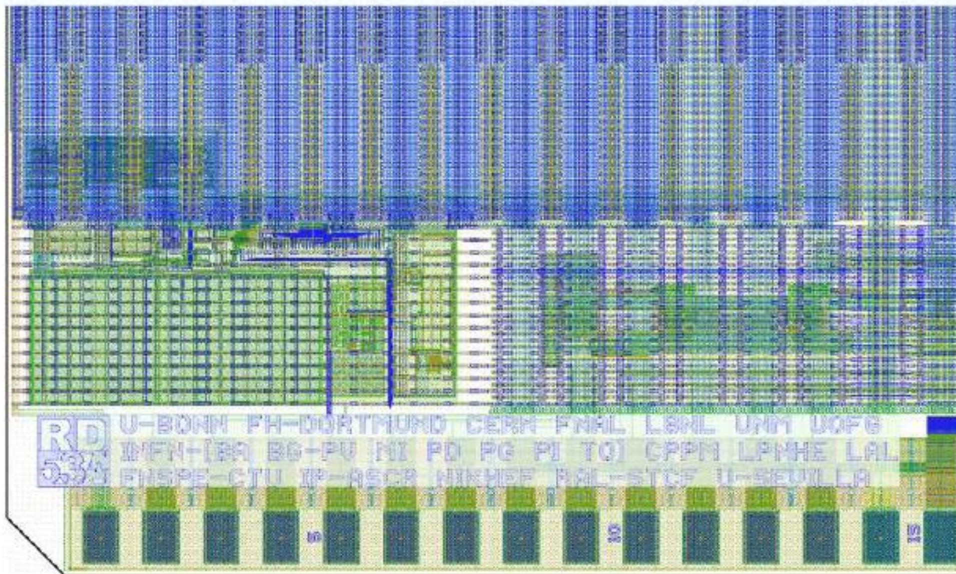


Fig. 4. The layout of the RD53A chip. The pixels can be seen at the top with the digital chip IO at the bottom [7].

# 2: RD53A Emulator

## 2.1: Motivation

The motivation for the RD53A emulator has changed over the years. Originally the RD53A emulator was going to serve as a stand-in for the real RD53A chip before it was released, as the emulator project started before any RD53A chips had been fabricated. The usefulness of having a stand in for the real chip is because

as part of the coming LHC upgrade the entire readout chain needs to be revamped and redesigned, not just the front-end chips. So, the data acquisition (DAQ) readout systems needed to be developed to be able to both communicate with the new front-end chip as well as handle the increased amount of data being produced. To that end several labs and groups have been developing new DAQ readout systems including YARR, RCE, and FELIX (more discussion on these in section 3). The emulator could then be used to help test and debug the new DAQ readout systems and prepare them for the chip's arrival. However, this use case has become mostly obsolete as the real RD53A chip was completed and released to the general public for testing purposes before the emulator was ready for public release. There was a brief period were only a few labs had access to RD53A's, but now most groups that wanted an RD53A have an RD53A. With this shift in mind the primary focus of the project changed from acting as a stand-in for the real chip to producing realistic data.

As stated above the current primary purpose of the emulator is to provide realistic hit data that will help debug and test the readout systems. What is meant by realistic hit data is that the data coming out of the emulator is consistent with the data that would come out of a real RD53A chip if it was installed in the LHC at CERN. This is useful because in order to have a real RD53A chip produce interesting data it needs to be stimulated by high energy particles, which are generally lacking in an average lab setting. While some labs have access to on-site particle accelerators and could use them to excite their RD53A, most do not and are limited to testing their DAQ systems on an RD53A sitting in a lab. The emulator helps solve this issue by being able to produce realistic data without needing a system to generate high energy particles. This can allow researchers to do testing on more interesting and realistic data which could be useful for both stress testing their systems (by configuring the emulator to output data at near maximum rates) and ensuring their systems work on realistic data and are not biased towards the "fake" data collected in a lab setting.

Beyond producing realistic hit data, the emulator has several other useful use cases. The first use case is general debugging and testing of DAQ systems. In addition to the debugging help provided by realistic hit data, the emulator can be used as a point of comparison for debugging issues that arise in trying to communicate with the real RD53A. Since the RD53A is essentially a black box for the purpose of testing internal signals it can be difficult to determine if the source of an issue is coming from the RD53A or the DAQ system. In this situation the emulator can be swapped with the real RD53A to help determine where the issue is coming from. If the issue disappears the RD53A is likely at fault and if not, then the DAQ is the most likely source of the issue. The second use case is that the RD53A emulator is easily modifiable. What this means is that if a bug is found or a small change in functionality is desired in the real RD53A, the interest group would have to wait until the RD53B is released. The process for a new integrated circuit to be fabricated and released can take months so this change would not be seen for a long time. However, the emulator is made of Verilog code targeting an FPGA so it can be easily modified and regenerated and as a result changes can be implemented over a period of days. Finally, since the emulator is targeting an FPGA, new groups interesting in the RD53A can much more easily acquire and use the emulator than the real chip as a starting point.

The rest of this section will discuss the flow of information through the emulator starting from the TTC data input all the way to the 1.28 Gbps four lane output. The design choices made and how aspects of the RD53A chip were incorporated into the emulator will be discussed along the way. To briefly overview the architecture of the emulator first a serial 160 Mbps input data stream is processed and converted into 16-bit frames which are then further split into 8-bit symbols. These symbols are then split into two separate paths depending on whether they are commands or triggers. Commands are processed into the appropriate behavior and triggers generate hit data. The output of these two paths is then merged back together and then

framed into groups of four to be outputted at a rate of 1.28 Gbps per lane using the Aurora 64/66B encoding. This same outline can be seen in the figure below.

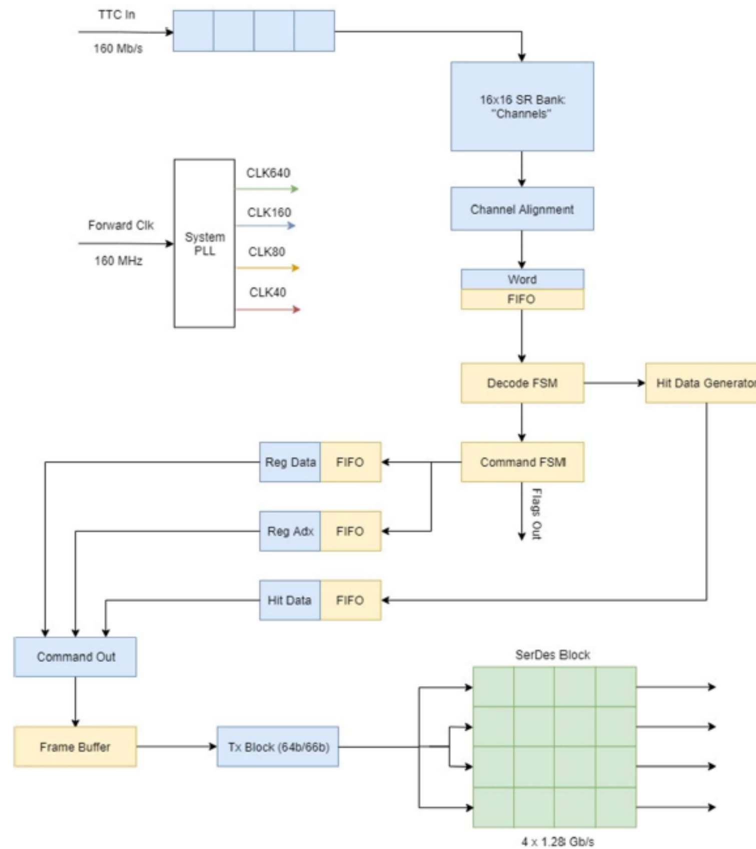


Fig. 5. High level architecture of the RD53A emulator [13].

Before continuing it is important to mention that the RD53A emulator has been a team project and numerous graduate and undergraduate students have worked on the project including myself. To properly give credit an outline of who has worked on what is listed below.

- TTC Data Processing: Joseph Mayer, Logan Adams
- Command Processing: Dustin Werran, Joseph Mayer, Logan Adams, Douglas Smith, Tony Faubert
- Trigger Processing and Hit Data Generation: Douglas Smith, Tony Faubert, Jessica Lan
- Data Handler: Dustin Werran, Douglas Smith
- Frame Buffering: Michael Walsh, Douglas Smith
- Data Output: Lev Kurilenko, Dustin Werran, Douglas Smith
- Custom Aurora Protocol: Lev Kurilenko, Timon Heim
- Integration with YARR, RCE, and FELIX: Douglas Smith

## 2.2: TTC Data Processing

As part of the design of the RD53A chip a custom encoding was created that transmits data, commands, triggers, and syncs to the chip at 160 MHz (this incoming data stream is called the TTC data). The RD53A encoding protocol encodes 5-bit data into 8-bit blocks called symbols and those symbols are transmitted in pairs called frames. The protocol was designed to be DC balanced at both the symbol and frame level and

to have a hamming distance of 2 between symbols. Being DC balanced means that for a given interval there are roughly an equal number of 1s and 0s being transmitted across the data line. This is important because transmitting low frequency (DC) data (i.e. when the signal consists primarily of either high or low voltages) is something many real-world communication systems have difficulty achieving. So, making the data DC balanced improves the transmission of the data. Having a hamming distance of 2 means that if a single bit flip occurs it can be detected as having occurred but not necessarily fixed. Furthermore, symbols starting or ending with three or more 0's or 1's (for example 0001_1010 or 0101_0111) were not used to limit the number of consecutive zeros or ones to 4.

Overall there are three types of data frames. The first type is command frames which send two copies of the same command symbol which allows for error correction to be done on commands. The second type is data frames which contain two different data symbols. The third type is trigger frames which contain a trigger and a data symbol. More discussion on what triggers, commands, and data symbols are / mean is discussed in the next subsection. The final important piece of information before discussing the flow of information through the emulator is the sync symbol (which has the form 817E). The purpose of the sync symbol as the name suggests is to allow the RD53A to sync with the incoming data stream and determine the correct order to decipher data. The sync symbol is 16 bits long and is not the same backwards as forwards. This ensures that the sync signal can only be interpreted in one direction and can be used for synchronization. This symbol is sent roughly every 32 frames and after the initial syncing of the RD53A chip it helps keep the system aligned to the incoming data.

In the emulator the TTC data comes in at 160 MHz as an LVDS signal (this helps ensure better signal quality) and is converted to a single ended signal for use in the emulator. The data is also accompanied by a 160 MHz clock which is fed into a PLL and used to generate all the different clock signals used in the emulator. It is important to note that in the real RD53A chip this extra 160 MHz clock is not needed as the system performs clock data recovery (CDR) on the 160 MHz TTC data to retrieve the 160 MHz clock. However, a true CDR system requires analog circuitry (specifically circuitry buried in a PLL) to function and this circuitry is not accessible in a purely digital design project like the RD53A emulator. So as a compromise the 160 MHz clock is sent with the TTC data.

After the data is converted into a single ended signal it then goes into sixteen different shift registers which will be called channels. These shift registers each output a 16-bit block of data and a valid signal. These valid signals are setup such that each shift register outputs a valid signal once every sixteen clock cycles and so that none of the valid signals of the different channels line up with each other. This separates out the sixteen clock cycles in which data is transmitted and makes it so that the channel that has the correct orientation of data at its output when its valid is high can be identified and locked too. After going through the shift registers, the data and valid signals are concatenated together and passed into a finite state machine (FSM) that determines which channel has the correct orientation of data. For each channel a sync counter and a locked status are recorded and there are two overall parameters, lock level (which determines the number of sync patterns needed to lock a channel) and unlock level(which determines the number of rival sync patterns needed to unlock a channel) that help drive the behavior of the FSM. When a channel is valid, if the output data is not the sync pattern the sync count and locked status are left unchanged. However, if a sync pattern is seen the sync count for that channel is incremented by one and what happens next depends on the value of the sync count and the locked status. If none of the channels are locked, then the first channel that reaches a sync count equal to the lock level is put in the locked state and all other channels are reset. The output of the locked channel and its valid signal are then sent to the rest of the system. Furthermore, every time the locked channel sees the sync pattern, all other channels are reset. After one of the channels reaches the locked state if a different channel reaches a sync count equal to the unlock level, then all

channels are reset. Overall this setup ensures that the correct channel of data is entering the system and that if the data frames shift to a different channel the emulator can follow it. Below is a figure outlining the flow of data through the TTC data processor.

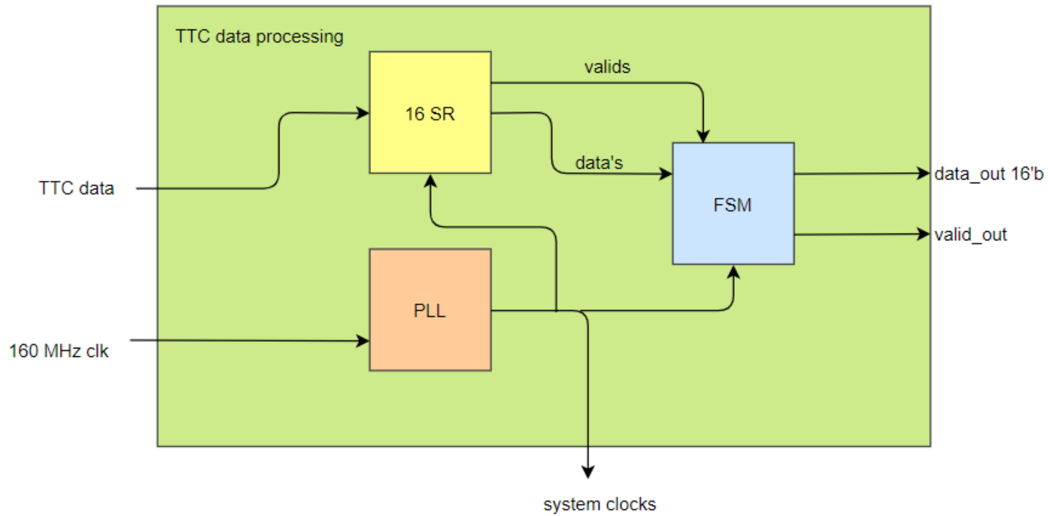


Fig. 6. Data flow of the TTC data processing unit where the incoming TTC data and 160 MHz clock are processed and converted into the system clocks and the correctly aligned 16-bit data frame.

## 2.3: Command Processing

After the TTC data is inputted into the system and gotten into the correct orientation it is then converted into meaningful commands that the emulator can process and response to appropriately. The RD53A transmission protocol has an 8 to 5 decoding structure which takes in the 8-bit symbols and outputs 5-bit pieces of data. The conversion results can be seen in the tables below.

| Symbol Name | Encoding | Data Value | Symbol Name | Encoding | Data Value |
|---|---|---|---|---|---|
| Data_00 | 0110_1010 | 5'b00000 | Data_16 | 1010_0110 | 5'b10000 |
| Data_01 | 0110_1100 | 5'b00001 | Data_17 | 1010_1001 | 5'b10001 |
| Data_02 | 0111_0001 | 5'b00010 | Data_18 | 1010_1010 | 5'b10010 |
| Data_03 | 0111_0010 | 5'b00011 | Data_19 | 1010_1100 | 5'b10011 |
| Data_04 | 0111_0100 | 5'b00100 | Data_20 | 1011_0001 | 5'b10100 |
| Data_05 | 1000_1011 | 5'b00101 | Data_21 | 1011_0010 | 5'b10101 |
| Data_06 | 1000_1101 | 5'b00110 | Data_22 | 1011_0100 | 5'b10110 |
| Data_07 | 1000_1110 | 5'b00111 | Data_23 | 1100_0011 | 5'b10111 |
| Data_08 | 1001_0011 | 5'b01000 | Data_24 | 1100_0101 | 5'b11000 |
| Data_09 | 1001_0101 | 5'b01001 | Data_25 | 1100_0110 | 5'b11001 |
| Data_10 | 1001_0110 | 5'b01010 | Data_26 | 1100_1001 | 5'b11010 |
| Data_11 | 1001_1001 | 5'b01011 | Data_27 | 1100_1010 | 5'b11011 |
| Data_12 | 1001_1010 | 5'b01100 | Data_28 | 1100_1100 | 5'b11100 |
| Data_13 | 1001_1100 | 5'b01101 | Data_29 | 1101_0001 | 5'b11101 |
| Data_14 | 1010_0011 | 5'b01110 | Data_30 | 1101_0010 | 5'b11110 |
| Data_15 | 1010_0101 | 5'b01111 | Data_31 | 1101_0100 | 5'b11111 |

Fig. 7. A table showing the conversion of 8-bit data symbols to 5-bit values [7].

Overall there are three types of symbols: commands, triggers, and data. Data symbols just represent data and the meaning of the data changes based on what it is associated with. Commands are the set of high-level functions that perform certain operations inside the emulator. The list of commands for the RD53A chip is as follows.

- ECR: This command causes the hit data path inside the chip to be flushed, clearing all prior pending triggers and hits.
- BCR: This command resets the internal counter of bunch crossing clock cycles which is used for synchronization purposes.
- Pulse: This command sends a global pulse to the emulator.
- Cal: This command performs digital and analog injection.
- Wr_reg: This command writes data to one of the systems global registers.
- Rd_reg: This command reads data to one of the systems global registers.
- NOOP: This command is a filler command that is sent when there is nothing to be sent.
- SYNC: This is the sync command.

Currently all these commands are properly received by the emulator, though ECR, BCR, Cal, and Pulse do not influence the system at large. Pulse and Cal were dropped as these influence analog parts of the chip which have not been included in the emulator while BCR/ECR are currently being implemented. The ability for the wr_reg command to write to pixel registers has also been removed as the pixels have been implemented as a single unit rather than individual pixels. This was done to reduce both the complexity and the overall size of the emulator. Each command also has some data associated with it, with the amount and meaning of the data being outlined in the table below. ID represents the chip id which in turn is a number assigned to each emulator connected to the system. In a full setup of the readout chain, the readout system will be connected to several FE chips. For certain commands in order to distinguish which chip it is meant for, the command will be sent with the chip id of that chip. D represents data whose meaning changes based on the command. For example, in wr_reg data is the value to write into the registers while in pulse data it is the length of the pulse. Finally, A represents the address of one of the global registers.

| Command | Encoding | ID/(A)ddress/(D)ata 5-bit Fields | | | | | |
|---|---|---|---|---|---|---|---|
| ECR | 2× 0101_1010 | | | | | | |
| BCR | 2× 0101_1001 | | | | | | |
| Glob. Pulse | 2× 0101_1100 | ID<3:0>,0 | D<3:0>,0 | | | | |
| Cal | 2× 0110_0011 | ID<3:0>,D15 | D<14:10> | D<9:5> | D<4:0> | | |
| WrReg | 2× 0110_0110 | ID<3:0>,0 | A<8:4> | A<3:0>,D<15> | D<14:10> | D<9:5> | D<4:0> |
| WrReg | 2× 0110_0110 | ID<3:0>,1 | A<8:4> | A<3:0>,D<15> | D<14:10> | 9×(D<9:5> | D<4:0>) |
| RdReg | 2× 0110_0101 | ID<3:0>,0 | A<8:4> | A<3:0>,0 | 00000 | | |
| Noop | 2× 0110_1001 | | | | | | |
| Sync | 1000_0001_0111_1110 | | | | | | |

Fig. 8. A table showing each of the commands, its encoding, and the data associated with it [7].

Triggers are a special type of command that represent whether data should be sampled at a bunch crossing (bunch crossings are the points in time when particle collisions have occurred) or not. Each trigger encodes which of the next four bunch crossing should produce hit data. The trigger commands are interpreted right to left.

| Symbol Name | Encoding | Trigger Pattern | Symbol Name | Encoding | Trigger Pattern |
|---|---|---|---|---|---|
| | | | Trigger_08 | 0011_1010 | T000 |
| Trigger_01 | 0010_1011 | 000T | Trigger_09 | 0011_1100 | T00T |
| Trigger_02 | 0010_1101 | 00T0 | Trigger_10 | 0100_1011 | T0T0 |
| Trigger_03 | 0010_1110 | 00TT | Trigger_11 | 0100_1101 | T0TT |
| Trigger_04 | 0011_0011 | 0T00 | Trigger_12 | 0100_1110 | TT00 |
| Trigger_05 | 0011_0101 | 0T0T | Trigger_13 | 0101_0011 | TT0T |
| Trigger_06 | 0011_0110 | 0TT0 | Trigger_14 | 0101_0101 | TTT0 |
| Trigger_07 | 0011_1001 | 0TTT | Trigger_15 | 0101_0110 | TTTT |

Fig. 9. A table showing the different encodings and patterns of the triggers [7].

With this information outlined the emulator implementation can be discussed. After the incoming TTC data has been converted to a 16-bit frame and gotten into the correct orientation, valid data is passed into a FIFO. This FIFO serves two purposes. The first is to split the 16-bit data frame into 8-bit data symbols so that the 8 to 5 conversion can be done. The second is to convert the system from the 160 MHz clock needed for the TTC data to a slower 80 MHz clock needed for internal data processing. After the symbol passes through the FIFO it can follow one of two paths. If it is a trigger symbol (or data associated with a trigger symbol) it will go to the hit data generator unit (this is discussed in the next subsection). Otherwise the symbol passes to the command processing unit.

The command processing unit consists primarily of a large FSM. This FSM converts the incoming symbols into meaningful commands (or rejects the incoming data if an error is detected) and manages the global registers. At the core of this FSM is the neutral state which acts as the transition state between processing different commands. After any command has been completed the FSM returns to the neutral state and the next state is decided based on the current symbol seen at the input. If it is a command symbol it will transition to a command state, otherwise it will remain in the neutral state. After the system has transitioned into a command state the FSM will stay in that command state until either an error is encountered such as the command symbol isn't seen twice/the chip id doesn't match or the command has been completed. The only exception to this movement process is the SYNC and NOOP commands. These symbols can occur in the middle of another command and need to be processed, but the system also needs to be able to return to the command it was previously on. The system accommodates this by saving the previous state so that once the NOOP or SYNC are finished, the system can return to what it was doing before. The structure of each command state is roughly the same with the major difference being how long the command runs for, which depends on the number of data symbols associated with that command. An outline of the resulting behavior of each command state is discussed below.

First for every command the state checks if the command is seen twice, breaking out of the command state if not, and continuing otherwise. For the SYNC, NOOP, BCR, and ECR commands this is the end of the state with a pulse being sent out to mark these commands as having occurred. For the rest of the commands, the next step is to check that the chip id matches the one assigned to the emulator. If it matches the FSM, then remains in that command state until the appropriate number of data symbols have been received. For read and write reg commands the global register at the given address is either written into or read out into the emulator system. For the pulse command a counter is set equal to pulse length which then counts down to zero and a pulse signal is held high while the counter is non-zero. Lastly for the calibration command three separate counters are set to a value specified by the command and count down to zero with calibration signals staying high while the counts are non-zero. Overall this setup allows commands to be properly

processed and responded to by the emulator. Below is a figure outlining the flow of data through the command processor.

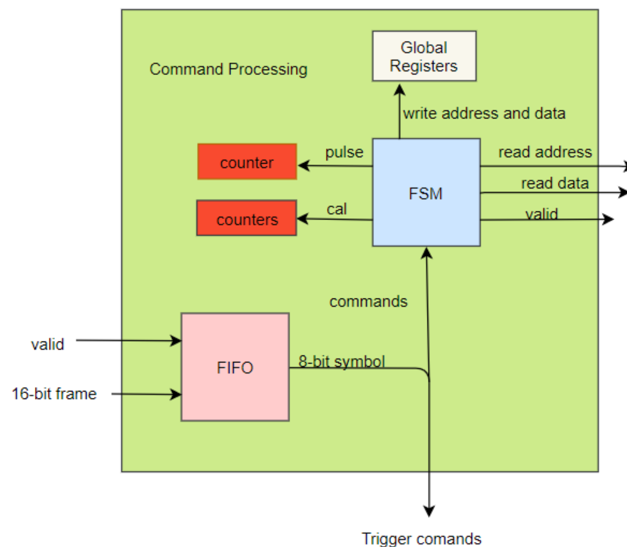


Fig. 10. Data flow of command processing unit where the incoming frames from the TTC data processor are split into symbols, with the commands being processed by the FSM and the trigger commands, read address, and read data being outputted to the hit generator and data handler respectively.

## 2.4: Trigger Processing and Hit Data Generation

In the previous section the process for handling commands was discussed. In this section more details about what triggers represent, the behavior of triggers in the emulator, and the resulting hit data generation will be discussed. In the LHC there are events called bunch crossings which are the points in time when particle collisions occur in the LHC. Bunch crossings occur every 40 MHz and a trigger represents whether data acquired during a specified bunch crossing should be recorded. As seen in figure 9 when a trigger symbol is decoded it produces a 4-bit value. This value tells the RD53A chip which of the next 4 bunch crossings it should trigger on. The reason for this 4 to 1 ratio is that only one trigger symbol can be sent per frame which arrive every fourth cycle of a 40 MHz clock. So, the 4 to 1 ratio keeps things in sync.

In the RD53A chip triggers interact with pixels, the front-end analog circuit that detects when particles pass through the chip. These pixels record a time over threshold value, and when a trigger comes in, the pixels outputs this value to be packaged and sent to the output of the RD53A chip. Since there are so many pixels, 76,800, in the real RD53A chip it is impractical to try and digitally emulate the behavior of every single pixel. So, a simpler model that emulates the behavior of the pixel array as a whole has been designed. How this was done was that first real data from the LHC was acquired and then analyzed using image analysis techniques such as clustering and line detection to determine what kind of shapes were commonly occurring in the data. Through this analysis it was determined that most of the data coming out consisted either of lines or simple Tetris like pieces, shown in the figure below. Using this information, a simple hit data generator module was designed to randomly output these types of shapes at the expected rates.

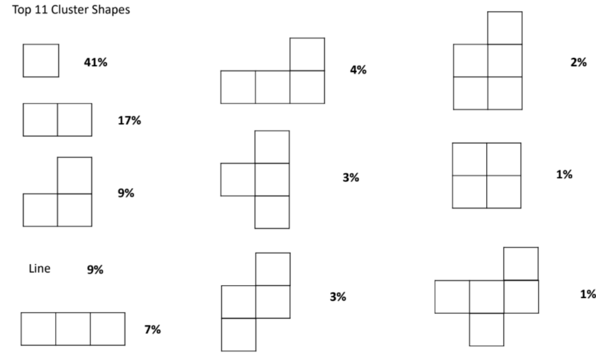


Fig. 11. The most common shapes seen in the LHC data and their occurrence rate.

In the emulator after commands and triggers have been split apart, the trigger and accompanying data symbol are sent to decoding units which use the RD53A decoding scheme to decode them. Then the trigger and data symbol go into the trigger table which is outlined in the figure below. In the RD53A chip the trigger table stores up to 32 triggers and keeps track of the trigger ID, the row of the trigger, the trigger tag, the data symbol that accompanied the trigger, and the BCID counter value, an internal counter of the bunch crossing clocks. To implement this, several FIFOs and some FSMs where instantiated. A 4 to 1 FIFO was created to strip the full 4-bit trigger value into 1-bit values as well as act as a storage unit for the triggers. Another FIFO was added to store the trigger tags and the output of both FIFOs are then fed into an FSM (this FSM will be referred to as the trigger FSM). This FSM controls the output of the two FIFOs and ensures that new triggers and trigger tags are extracted when the hit data generation unit is ready for them, ensures that the two FIFOs stay in sync with each other, and that the correct data tag is coming out with the correct triggers. The system also has two counters which maintain the trigger ID and the BCID value. The BCID counter is incremented with the 40 MHz clock and the trigger ID counter is incremented every time a trigger is extracted from the FIFOs.

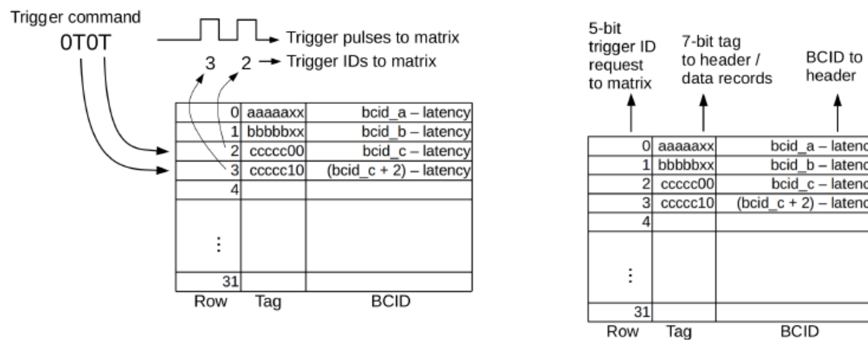


Fig. 12. The shape of the trigger table, the data it holds, and its outputs [7].

The trigger, trigger tag, triggerID, and BCID value are then grouped together into a 32-bit header and sent to the hit data generator. The hit data generator consists primarily of three FSMs; one for top level control, one to generate clusters, and one to generate lines. The first FSM controls the top-level behavior of the hit generator and determines when to output new data and when to process new triggers. This is done through a handshake mechanic between the top-level hit data FSM of the data generator and the trigger FSM. The hit data FSM consists of four states, IDLE, START, WORK_l, and WORK_c. When in the idle state the system outputs no data and remains in the idle state until a trigger is seen. At this point the system transitions

to the start state and a done flag is set to zero to signal the trigger FSM to wait. This is the first part of the handshake. While in the start state the system will take one of two paths. The first path is that the system will output a block of data, the upper half of which is the 32-bit header block and the lower half is a data block and will then transition to either the WORK_l or WORK_c state. The structure of these two types of data block is shown below. The second path is that the system will output no data and jump back to idle. The determination of which path is taken is controlled by the configuration register. The configuration register is compared to a random number and if the configuration register is larger data is outputted. This config register is not in the original RD53A chip spec but was an added feature to the emulator. It makes use of one of the registers used for analog circuitry that was not included in the emulator and allows for outside user control over the amount of data the generator produces. For the first path the choice between WORK_l or WORK_c is controlled by another random variable. In the case of WORK_l, line data is outputted until there is no more line data and for WORK_c cluster data is outputted until a delimiter bit is seen. In either case once the end point is reached the system goes back to idle and the done flag is set high. This completes the handshake and lets the trigger FSM know that a new trigger can be sent to start the process all over again.

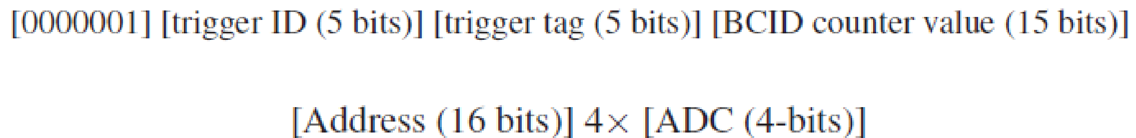


Fig. 13. The structure of header blocks (top) and data blocks (bottom) [7].

The second FSM generates cluster data and operates independently of the first FSM. The generated data is stored in a FIFO that the first FSM pulls from to get cluster data. The states of this FSM are RST, DELIMIT, WAIT, GEN, and PADD. The FSM starts in the reset state and once the FIFO has finished resetting it transitions to the delimit state. The delimit state inserts a dummy variable into the FIFO and transitions to either the generate state if there is enough space in the FIFO or the wait state if not. The purpose of the dummy value is to act as a padding unit. This is because the FIFO the data is feeding into is a 1 to 2 FIFO and requires two 32-bit values to output a 64-bit value, the shape the first FSM needs the cluster data to have. For most cycles the first FSM needs to output two blocks of hit data so putting two blocks of cluster data into the FIFO is fine. However, when the first FSM is in the start state and pulls a value from the FIFO, part of the data needs to be a dummy variable so that one piece of data can be replaced with the 32-bit header. While in the WAIT state the system remains in the WAIT state until enough space is available in the FIFO to store the next set of cluster data, at which point the FSM transitions to the GEN state. While in the GEN state the FIFO receives cluster data from the cluster maker unit which uses a similar start/done handshake mechanic as in the first FSM to let the second FSM know when it is finished. The cluster module utilizes many LFSRs to randomly create a group of 35 clusters of random Tetris shapes at random locations on the pixel array using the data encoding described in figure 13. These cluster regions are then fed into a Bloom filter which quickly determines if that region has been seen before. If it has then that data is not added to the FIFO as it would not make sense for the same region to appear multiple times for one trigger. Upon receiving the done signal from the cluster module, the FSM then goes back to either the DELIMIT state or the PADD state. The PADD state is reached if the amount of data sent to the FIFO was not an even amount as it needs to be even due to its 1 to 2 nature. The PADD state adds a single point into the FIFO and then goes back to the DELIMIT state.

The third and final FSM generates line data and uses a start/done handshake mechanic between the line FSM and the top-level FSM that lets the line unit know when to start making a line and that lets the top-level FSM know when it has finished generating a line (it is contained in the line maker unit in Figure 14). After receiving a start signal the FSM will randomly choose to generate either a vertical or horizontal line. The system then uses more LFSRs to choose where to place the line and then outputs the line point by point back to the top-level FSM. Finally, as data comes out of the top-level FSM it goes into an output FIFO which stores the data and passes it to the data handler. Overall this structure allows triggers to be correctly processed and converted into hit data. An outline of the data flow can be seen in the figure below.

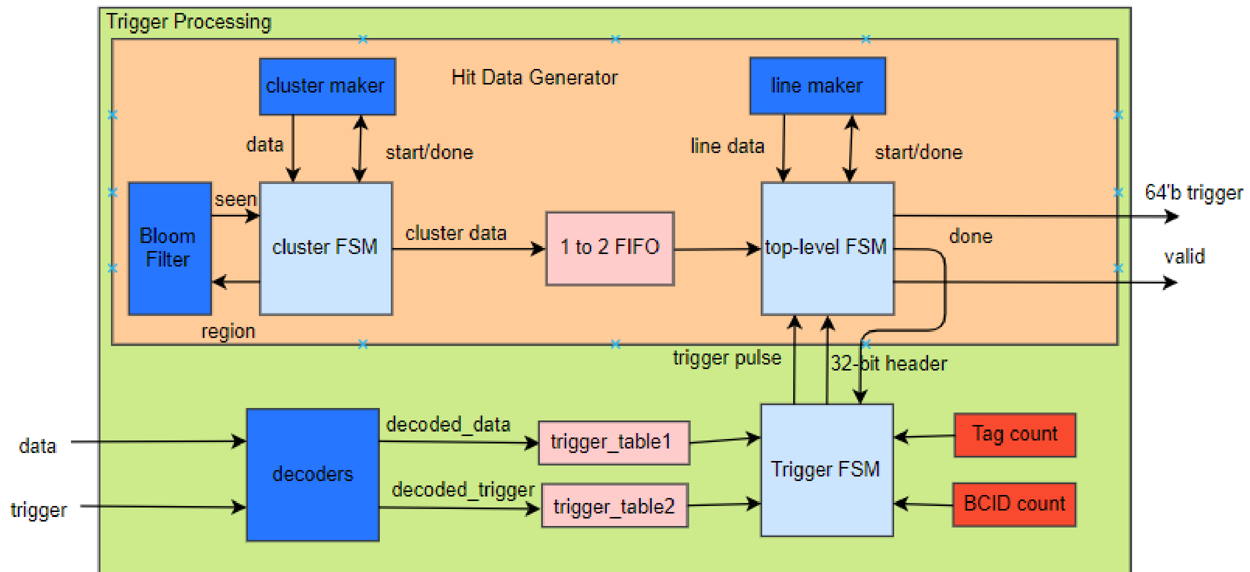


Fig. 14. Data flow through the hit generation module with data and triggers coming from the TTC data processor and the 64-bit trigger and valid signal being sent to the data handler.

## 2.5: Data Handler

After the commands have been processed and the hit data has been generated the two data streams need to be recombined to be sent to the output. The rate of hit data transmission to command data transmission is an N to 1 ratio as shown in the figure below (the value of N can be configured but is usually set to 48 so that 98% of the output data stream is dedicated to outputting hit data). The data contained in the command frame consists entirely of the output of the read register command. When there is register data to be transmitted it will have the format shown in the figure. Otherwise it will come from one of the auto read registers in the global register bank. The value of ZZ depends on the number of read registers that need to be outputted. If there are none it is B2, if there is one it is 99, otherwise it is D2. An important note is that only the first lane outputs read register values, the other three lanes use the auto read registers only (the four lanes are discussed more in the next subsection).

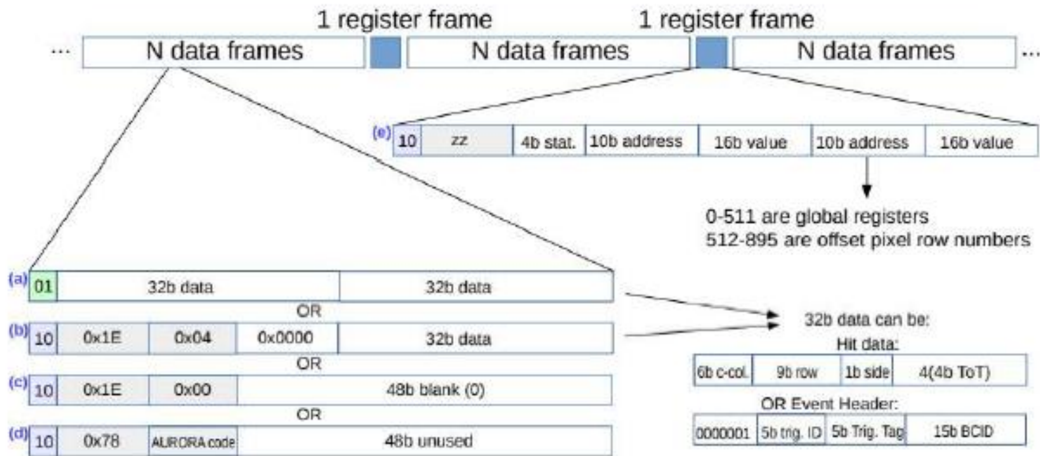


Fig. 15. The rate of data to register frames as well as the structure of data and register frames [7].

The fusing process of the hit data and the command data in the emulator is primarily driven by a configurable counter with the appropriate behavior occurring at different values of the counter. The configurability of the counter allows for the adjustment of the ratio of hit data to register data. Before data goes into the counter the read register address, read register data, and the hit data are stored in three separate FIFOs. These FIFOs allows the data to be stored until the cycle of the counter where it is needed. From the counter cycles 0-379 of the counter the system checks if there is valid data in the hit data FIFO. If there is it is passed on further through the system and a valid flag is set to high to indicate to units downstream that there is new hit data available. Then during the counter cycles 380-390 the register data is processed and transferred out to the data framer unit. The first three cycles extract the read data from the FIFOs (if there is any) with the next two cycles formatting the data using the format shown in the figure above, sending the read data if there is any and sending the auto read register for lane 0 if there isn't. The remaining six cycles then output the auto read registers for the remaining three lanes. As part of this transfer of data a service frame flag is set high to indicate that this is a register frame and not regular hit data. After the register data has been handled the counter then checks if a channel bonding frame needs to be sent. At the 391 cycle of the counter if the value of the CB counter is high enough the CB data frame is sent for the next eight cycles and then the counter is reset to 7. If the CB counter is not high enough the CB counter is incremented by 1 and the main counter is reset to 0. Overall this structure combines the two data streams together and gets the data output stream in line with the RD53A specifications. A figure outlining the flow of data is shown below.

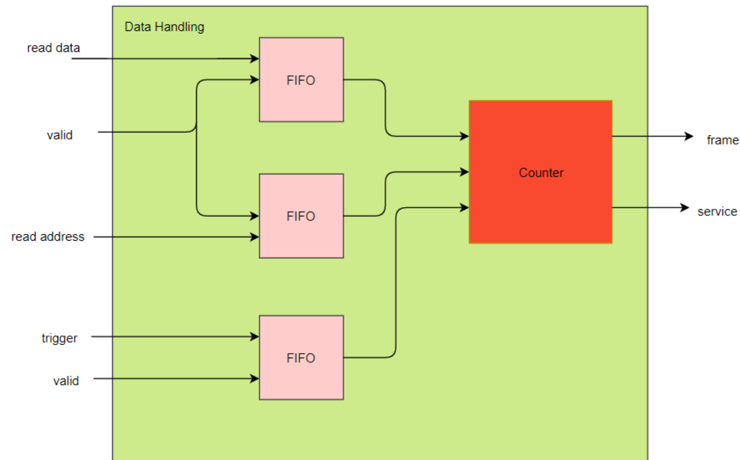


Fig. 16. Data flow through the data handling unit with the read register data and hit data inputs coming from the command processor and data generator units respectively being combined into one output stream to send to the data framer.

## 2.6: Data Framing

After the trigger and command data streams have been merged together into one output stream, the data then needs to be aligned into groups of four. This is required as the RD53A has four data output lines that send data out in parallel, so the data stream needs to be converted from serial to parallel. In order to accomplish this the data is first fed into two FIFOs, one for data frames and one for service frames. This FIFO setup serves several purposes. The first purpose is to do data alignment and parallelization as the FIFOs implemented are one to four FIFOs. This means that the FIFOs will store the incoming serial data in an internal memory buffer and output that data in parallel bunches of four. The second purpose is to act as a buffer between the clock domains of the data generation and the data transmission units. This reduces the complexity of the timing of the system and allows for the transmission system to be run at different clock speeds without worrying about breaking the system. Finally, two FIFOs were used to allow the FSM (discussed in the next paragraph) to distinguish between service and data frames and handle them accordingly.

After the data is processed by the FIFOs, it then enters a finite state machine (FSM) which controls the output side of the two FIFOs and determines what/when data should be sent to the transmission unit. The FSM consists of three states, IDLE, DATA, and SER and is primarily driven by an input pulse from the transmission unit that tells the FSM that the transmission unit needs the next piece of data. Controlling the FSM using an impulse ensures that the output remains static until the next pulse which the transmission unit requires to function properly. When in the IDLE state the FSM outputs the standard RD53A idle data 1E00_0000_0000_000 and a header 10 (more on headers in the next subsection) and when in the DATA/SER state data/service data and the header 01/10 are outputted respectively. The next state is determined by which FIFOs currently have new data. If the service FIFO has data the next state will be SER, if the data FIFO has data and the service FIFO does not the next state will be DATA, otherwise the FSM goes to the IDLE state. This ensures that service data frames are given preference for being outputted and that both service and data frames are being processed as quickly as possible. This overall setup also ensures that the four pieces of output data are aligned to each other and that each piece of data is presented to the transmission unit for the correct period of time. A figure of the flow of data is shown below.

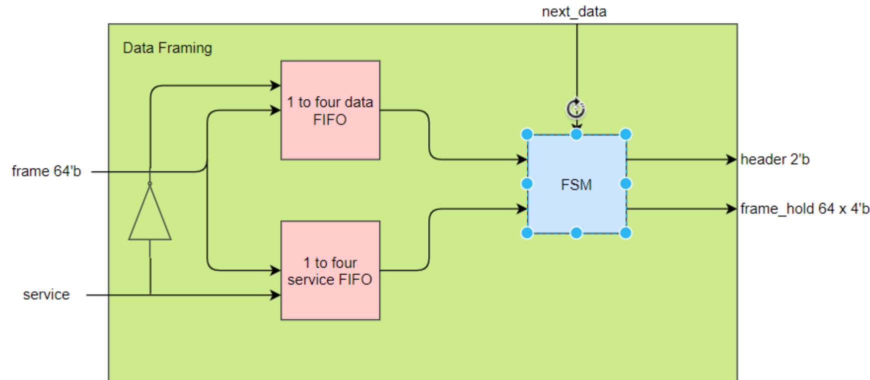


Fig. 17. Data flow of the data framing unit where frame and service correspond to the data stream from the data handler, next_data is the impulse from the transmission unit, and header /frame_hold are the outputs to the transmission unit.

## 2.7: Data Output

In the RD53A chip there are two ways data can be transmitted from the system. The first is to send the data out over four separate communication lanes at a speed of 1.28 GHz. This results in 5.12 Gbps of data being transmitted over the line. The RD53A can also send data over a single communication line at 5.12 Gbps but this was not implemented in the emulator and the 4-lane version was used. The RD53A chip also can adjusted to communicate at slower speeds all the way down to 160 Gbps. The emulator cannot be adjusted after being programmed onto an FPGA, but it can be easily adjusted manually to different speeds. The protocol used to encode the data on these output lanes is the Aurora 64/66B protocol. The Aurora protocol sends data out in 66-bit packets with the first two set as a header and the remaining 64 as data. A header of 01 means the 64-bit data represents hit data and 10 is used for anything else. The advantage of using the Aurora protocol is that a bit flip is guaranteed to occur every 66 bits regardless of the value of the data, which helps the receiver with synchronization. The implementation of this unit and of the Aurora protocol is discussed in section 3.

# 3: Readout Systems

The previous section covered the work being done on the front-end chip part of the ITk upgrade. However as was discussed in section 1.2 the front-end chip is not the only thing being upgraded. In fact, most of the system needs to be replaced or adjusted. This includes the work being done to develop the new readout systems for the RD53A chip which will be the focus of this section. The new readout systems need to be able to both communicate with the RD53A using the new 160 Mbps encoding system, receive data from the RD53A using the Aurora protocol, and be able to handle the increased data flow caused by the upgrade in luminosity. To achieve this end several different groups have been working on and developing new readout systems. The three systems that will be discussed include YARR, RCE, and FELIX.

## 3.1: YARR

### 3.1.1: What is YARR?

YARR (which stands for Yet Another Rapid Readout) is a readout system being developed by Lawrence Berkeley National Lab (LBNL) with Timon Heim spearheading the project. The primary focus of the

project is to move information received from the front-end chip to a host computer where it can be processed into useful information [8]. This project started off being compatible with the older brand of front-end chips, the FEI4, but recently added compatibility with the RD53A. To achieve its primary goal the YARR project consists of two parts, a firmware part and a software part.

The first part of YARR that will be discussed is the firmware portion. The YARR firmware is written in VHDL code and is programmed onto an FPGA. Much like with the emulator, having the firmware work in this manner makes it easy to reconfigure the hardware setup and both debug the system as well as test new things. The firmware's main purpose is to both transfer the data coming out of the RD53A and pass it to the host PC, and take commands sent from the PC and pass them to the RD53A. At the high level YARR takes in four lanes of transmission data from any connected RD53A (YARR can currently handle up to four RD53As), does some processing to combine the incoming data streams, and format it so that it can be passed to the PC over a PCIe bridge, which can handle faster transfer speeds than previous setups allowed. At the same time YARR take the data coming across the PCIe bridge which consists of RD53A commands, triggers, and data, does some processing to get the data to the correct speeds, and passes the information into the RD53A. The second and arguably more important half is the YARR software. At the high level the YARR software sends the relevant commands to the YARR firmware while simultaneously taking in the information coming across the PCIe slot and converts it into observable information. Some of the functionality in these programs includes sending trigger patterns and generating histograms/heat maps of the results, reading and writing to global registers, calibrating the chip, etc. Overall the YARR software suite has become the standard for testing and interacting with the RD53A chip.

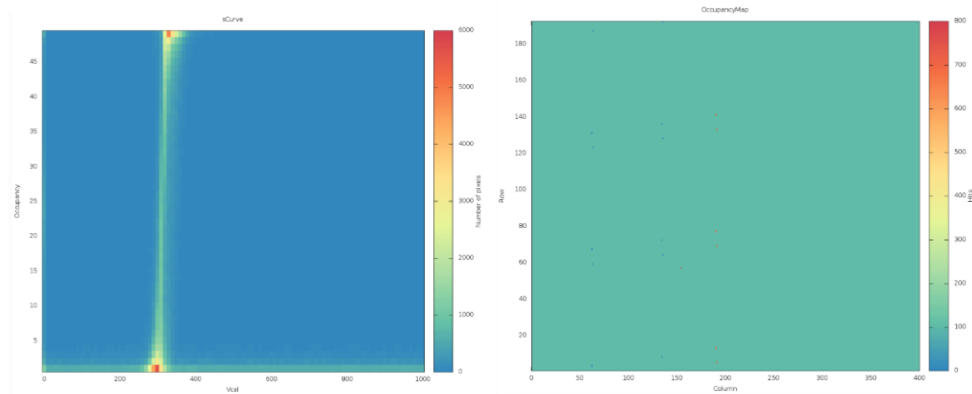


Fig. 18. An example of what can be accomplished with YARR software [8].

## 3.1.2: YARR Hardware

To host the YARR software a custom PC was built. The primary driving factor behind this was that the standard DELL and HP motherboards have issues communicating with the YARR firmware across the PCIe bridge, so a PC with a more specialized motherboard was need. The Asus – H1101-PLUS/CSM motherboard was used as Timon has had success communicating with YARR using this brand. The rest of the PC was built using the Noctua-NH-L9i fan, Kingston – HyperX Fury 16 GB memory, Samsung 250 GB SSD, Lian-Li PC Tower, Intel i5-6500 processor, and Corsair- SF 450 power supply. These were chosen because they were compatible with the Asus motherboard.

To host the YARR firmware a KC705 Xilinx board was used. As discussed in the previous section the YARR firmware and software communicate across a PCIe bridge and the firmware targets an FPGA. So, to host the YARR firmware, hardware is required that has both a 7 Series FPGA as well as a PCIe slot.

There are several compatible boards including the ReflexCES XpressK7, Trenz TEF-1001, and the Xilinx KC705. The KC705 was used due to being readily available in the ACME lab, though the YARR firmware needed some mild tweaks and edits (mostly in the constraints) to make it compatible with the KC705. To connect the YARR host PC to the YARR firmware board a PCIe extender cable was used as the KC705 did not physically fit into the host PC. To connect the YARR firmware to the RD53A (and the emulator) an Ohio MMC card [11] was attached to the KC705. This chip connects to the FMC on the KC705 and converts the transmission signals to display port. Display port was used as that is the input port on the RD53A, but also because it provides the necessary five lines of communication: 1 for the 160 Mbps data, and 4 for the Aurora data.

Finally, to host the emulator firmware another KC705 Xilinx board was used. The reason for the use of the KC705 like before was that it was easily accessible. Having the emulator exist on a separate board from the YARR firmware was also useful in that it made the system more realistic to the RD53A in that the signals had to go over cables and not just transmit information internal on the FPGA. To connect the YARR firmware to the emulator two connector boards were used, one that converted display port to HDMI, and then a second board that converted HDMI to display. The reason for this chain is because that was the hardware available, though the use of a second Ohio card would have simplified things.

An important limitation that came from using this hardware setup is that the system could not be run at the full 1.28 Gbps. The source of this limitation is twofold. First is that the FMC outputs of the KC705 are only rated for communication speeds of 1.25 Gbps. While it is possible to overcome this issue with the use of some of Xilinx's delay hardware there is another limitation in the breakout boards used to connect the YARR firmware to the emulator. In particular, the display port to HDMI board was not designed with high speed communication in mind and in observing the board itself it can be determined that the wiring does not have the necessary shielding needed to send clean transmissions. As a result of these limitation for the YARR system the maximum transmission speed was downgraded from 1.28 Gbps to 640 Mbps.

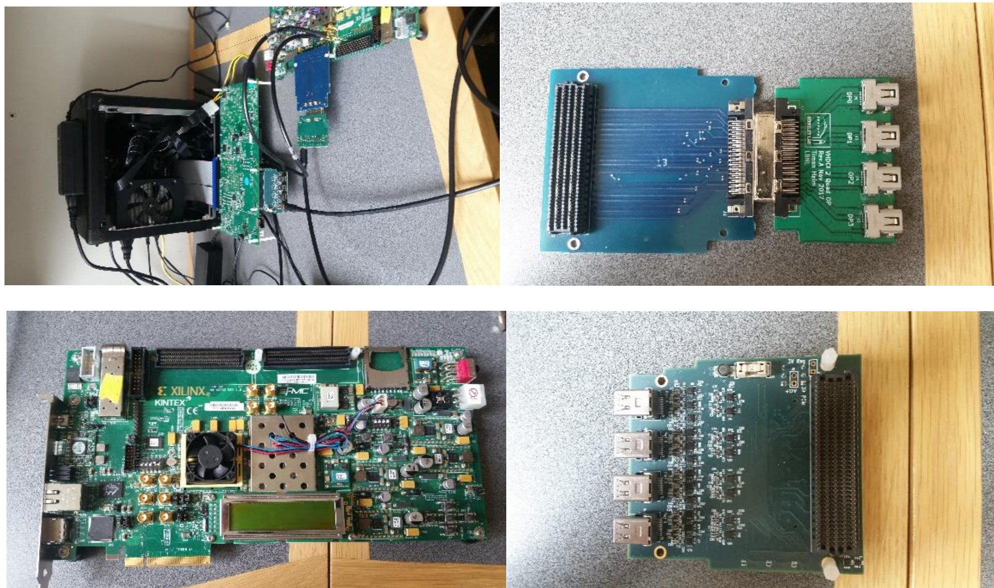


Fig. 19. Hardware used to setup YARR. Top left full setup, top right FMC to DP breakout boards, bottom left KC705, bottom right Ohio MMC card.

### 3.1.3: YARR Modifications for the Emulator

The first major modification made to the emulator to support communication with YARR was implementing a custom Aurora protocol that could use the Xilinx SERDES modules. To facilitate communication between YARR and the RD53A, YARR makes use of the Xilinx SERDES module. How the SERDES module works is that it takes in two clocks (for this discussion they will be called high and low) and a bus of data. New data comes into the SERDES in sync with the low clock and the bus is transmitted serially out of the SERDES in sync with the high clock. This module while easily customizable does not provide the high-level functionality that other Xilinx IP modules do. For example, the most commonly used Xilinx module for high speed communication is the Transceiver wizard. This allows a user to implement a transceiver that already has communication protocols built in such as the Aurora protocol. However, this IP must be routed through a GTX core and out of an appropriate communication channel such as an SFP+ cage. The issue here is that not all boards have this kind of hardware, but any FPGA board can instantiate a SERDES modules. So, to make use of the flexibility of the SERDES modules and to be able to communicate with YARR a custom Aurora protocol was needed and implemented [12].

To understand the implementation of the custom Aurora protocol, first an understanding of how Aurora transmits data is needed. As discussed previously Aurora data consists of two pieces, a 2-bit header and a 64-bit block of data. The header and data are transmitted together across the line as a 66-bit block of data. To facilitate the transfer the data first goes into a scrambler. The purpose of this scrambler is to scramble the bits such that the result is DC balanced which is important for transmission. It does not serve as an encryption method. The scrambler is also set up in such a way that it is self-synchronizing. What this means is that even if the transceiving and receiving scrambler and descrambler are initialized at different times, after two blocks of scrambled data the receiving descrambler will sync with the transceiving scrambler. How the scrambler works is that each bit of the 66-bit piece of data has the following function applied to it, $x^0 \oplus x^{38} \oplus x^{57}$. What this means is that if you imagine the 66-bit piece of data as a circular ring, so that if you were counting through the data after the 66-bit you would go back to 0, the current data bit is xored with the $38^{th}$ and $57^{th}$ bit relative to the current data bit. This function scrambles the data and gives it the desired properties [12].

After the data is processed by the scrambler it then needs to be converted from a 66-bit block of data to a 32-bit block of data so that it can then be more easily divided down to a serial transmission since 32 is a power of two. In order to do this conversion a shift register capable of holding 96-bits is required. How this system works is that when the first 66-bit data blocks comes in it fills the first two-thirds of the shift register. Then before the next block of data comes in the system can transfer two 32-bit blocks out of the shift register, leaving two residual data bits behind with the new 66-bit data block. This cycle repeats for 30 more clock cycles until there are 32-bits of residual data left behind in the shift register. At this point the system needs to halt the incoming data for a clock cycle so that the old data can be flushed out and the cycle can begin anew. The figure below shows this process graphical.

| Clock Cycle | Bits: 95-64 | Bits: 63-32 | Bits: 31-0 |
|---|---|---|---|
| 0 | D1[2b] | D1[32b] | D1[32b] |
| 1 | | D1[2b] | D1[32b] |
| 2 | D2[4b] | D2[32b] | D2[30b], D1[2b] |
| … | | | |
| 30 | D16[32b] | D16[32b] | D16[2b], D15[30b] |
| 31 | | D16 [32b] | D16[32b] |

| 32 | | | D16 [32b] |
|---|---|---|---|
| 33 | D17[2b] | D17[32b] | D17[32b] |

Fig. 20. Flow of data through the 66-bit to 32-bit shift register.

Connecting things back to the emulator, coming from the data framing module are four sets of 2-bit headers and 64-bit data. The header and data are bundled into 66-bit data blocks and each of the four 66-bit data blocks is sent to its own Aurora output channel (though they all behave the same way). First the data is fed into the scrambler unit which performs the scrambler operation on the data. The scrambled data is then sent to the 66 to 32 conversion unit which consists of a shift register and a counter. The shift register stores the data and the counter tracks what cycle the data is in to determines when new data should be requested and when the system should pause to let the buffers clear, sending a pulse back to the data handler unit when new data is needed. Next the data passes into another simpler shift register which scales the data blocks down from 32 bits to 8 bits. Finally, this 8-bit value is fed into the Xilinx SERDES block which sends the data out serially.

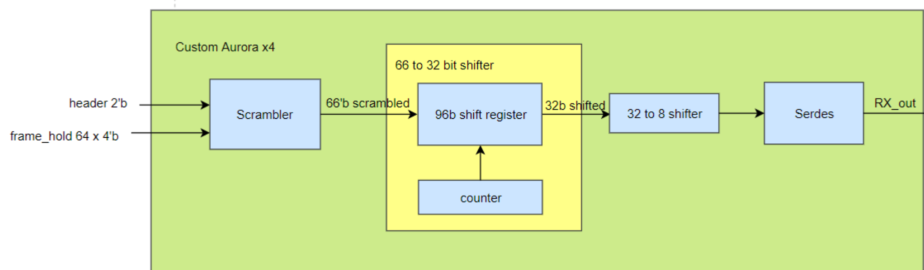


Fig. 21. Data flow through the custom Aurora transmission unit with the header and frame inputs coming from the data framing unit and the RX_out leaving the chip.

As was mentioned in section 2.1 the emulator requires an input clock sent with the data due to the difficulties of doing CDR in a digital project. For the baseline emulator this clock is expected to be 160 MHz. However, in YARR there isn't an exact 160 MHz clock to send to the emulator. In YARR the internal clocks are being generated using a PLL which is driven by a clock received through the PCIe connection which is 250 MHz. While one of the clocks being generated by the PLL is 160 MHz, it is not exactly 160 MHz. The exact value is 156.25 MHz and trying to use this clock to drive the emulator would lead to timing instabilities. So instead of a 160 MHz clock, the PLL in the emulator was changed to use the 250 MHz clock. However, this change alone caused the system to fail to communicate. This reason for this is because of the differences between the requested clocks and the actual clocks in the YARR PLL. Like how the 160 MHz clock was 156.25 MHz, all the clocks generated by the YARR PLL were slightly off. This resulted in the emulator and YARR getting out of sync. In order to fix this issue, the clocks in the emulator were adjusted to match the clocks in YARR.

The final major edits done were on the YARR firmware. One of these was adjusting the clocks feeding into the YARR RX core to receive data at the slower 640 Mbps speed. The second change was updating the control module for the Ohio MMC card. Because the Ohio card can both receive and transmit data across the display port lines, it has to be configured to determine which lines are receiving and which are transmitting. To do this there is a 24-bit shift register that takes in serial data and when a LOC signal is pulsed it will write the inputted data into the display port line control blocks. The bottom 20-bits control the 20 different display port lines (5 lines for each of the 4 lanes) with the top four connected to nothing. To ensure the lines are configured to transmit in the correct direction a simple counter was implemented

that continuously wrote in the correct configuration to the shift register and pulsed the write command every 24 clock cycles. With these changes in place YARR and the emulator were able to communicate. Some of the results can be seen in the figures below.

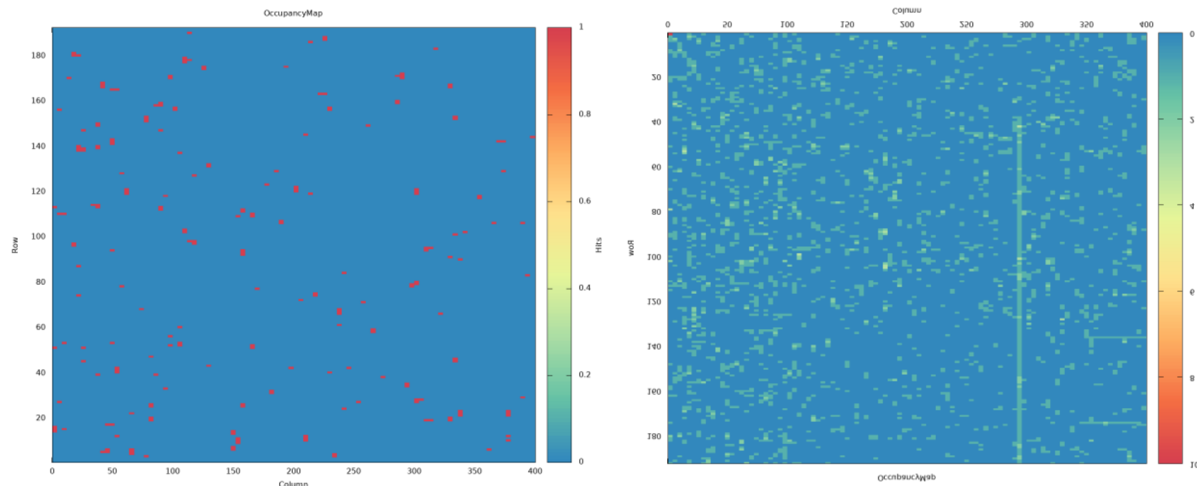


Fig. 22. Data acquired using the YARR hardware setup with the system configured to output a low amount of data on the left and a large amount of data on the right.

## 3.2: RCE

### 3.2.1: What is RCE?

Another readout system being worked on is the RCE (which stands for Reconfigurable Cluster Element) readout system which is being developed by the RCE development Lab at the SLAC National Accelerator Laboratory. The focus of this laboratory is to explore detector readout applications and new DAQ architectures for the coming ATLAS upgrade that is based on their modular building blocks which include RCE. In working towards this goal, the group has begun working on establishing high speed communication with the RD53A chip and transferring the information back to a host computer where it can be processed into useful data [9]. As before the project is split into two primary parts, the firmware and the software. The firmware is again designed in VHDL which allows for testing and rapid reconfigurability of the RCE system and is loaded into the boards of the RCE hardware setup to establish communication between the host PC and the RD53A. On the software side the project has software that allows the user to reconfigure the hardware and more recently has imported the software from the YARR project due to the versatility and functionality it possesses in regards to interacting with the RD53A. As a result, parts of the YARR firmware have also been integrated into the RCE firmware to support the YARR software.

### 3.2.2: RCE Hardware

To set up the full RCE hardware chain the following equipment is required. First an FEB (front end board) is required to host the emulator firmware. The FEB board is a custom chip that the SLAC group have designed for the purpose of front-end readout. This was used in place of a KC705 due to the SLAC group have easier access to additional FEB boards over KC705s. The emulator board is then connected to another FEB board using a display port cable, which then in turn is connected to an HSIO2+DTM board using an QSFP+ cable. This hardware setup hosts the RCE firmware and allows for up to 16 RD53As (real chips or emulators) to be connected to the system as each FEB board can connect to 4 RD53As and each HSIO2

board can connect to four FEB boards. Finally, the HSIO2 board is connected to a host PC with a 10G ethernet card using an ethernet cable where the software can process the data. Since a fair amount of this equipment was lacking at the UW a modified version of the hardware chain was implemented for testing. The FEB board was still used to host the emulator firmware, but the rest of the chain was constructed using the YARR hardware setup. While not an exact copy of the full RCE hardware chain this setup was determined to be an adequate representation of the RCE setup and good enough for comparative testing and helping the group at SLAC debug their emulator setup as some of the critical components of the RCE firmware, especially those that are directly interfacing with the incoming data from the emulator, uses YARR firmware.

Fig. 23. The full RCE hardware setup.

### 3.2.3: RCE Modifications for the Emulator

Due to the similarities between the YARR firmware and the RCE firmware many of the changes needed to communicate with YARR translated over to the changes that would be needed for RCE. There are a few notable differences though. One of these changes involves the sent 160 MHz clock and its impact on the main PLL. Unlike in the YARR system where a 250 MHz clock needed to be sent, the RCE firmware had a “clean” 160 MHz clock to send. So, the system was set back to expecting a 160 MHz input. Also unlike in YARR the outputs of the PLL were exact so the clock difference issues were no longer a problem. A quick side note is that while this difference applies to the full RCE hardware setup, for the system setup at the UW the YARR clock setup was still required. The other major change was the output speed of the emulator. The RCE setup is currently receiving data from the RD53A at 160 Mbps and not at the higher speeds of 640 Mbps or 1.28 Gbps. Because of this the output speed of the emulator had to be slowed down by a factor of four to send data at the lower rate. This required the addition of a new 20 MHz clock and made it necessary to update the FIFOs in the data framing unit from how they were originally implemented to a version where the write side and the read side of the FIFOs were in separate clock domains. This made it easier to transfer data from the higher internal speeds to the now slower data output speed of 20 MHz without having to worry about timing issues. With these changes in place the RCE/YARR system was able to communicate with the emulator. Some of the results can be seen in the figures below.

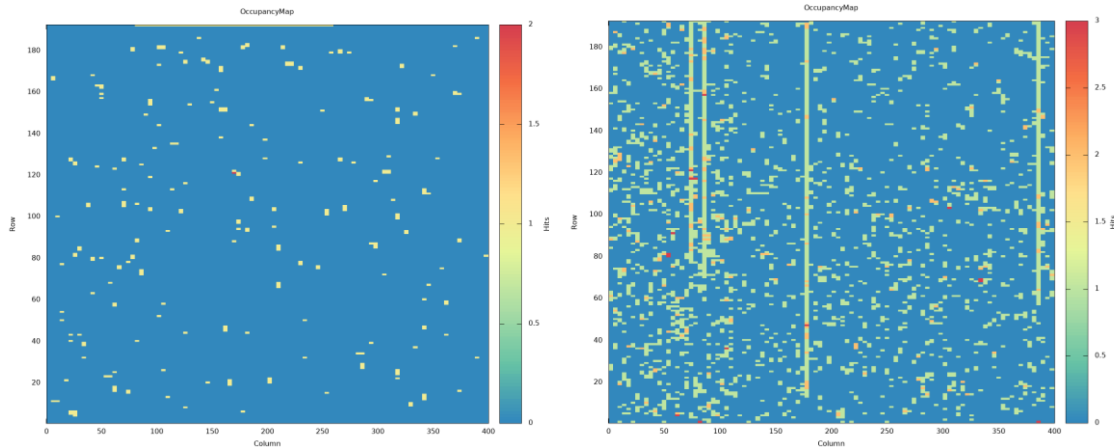


Fig. 24. Data acquired using the RCE hardware setup with the system configured to output a low amount of data on the left and a large amount of data on the right.

# 3.3: FELIX

## 3.3.1: What is FELIX?

The final readout system that will be discussed is the FELIX readout system which has been developed by ANL (Argonne National Lab). It has been designed as part of the Atlas upgrade effort and is a data router for sending data from front end chips to programmable peers. Unlike other efforts which are tailored towards performing readout of a specific hardware platform (i.e. the RD53A chip for YARR and RCE), FELIX has been designed to serve as a general-purpose readout system [10]. While like YARR and RCE it can perform readout of an RD53A chip, FELIX can also perform readout for other ATLAS projects as well as several non-ATLAS projects. It accomplishes this by doing the data processing in software while at the same time having flexibility in both the firmware and the software to support multiple technologies and readout configurations.

In terms of firmware, FELIX has two different link protocols to facilitate the transfer of data from the front end to the host PC. The first is the Gigabit Transceiver (GBT) and the Versatile Link. The GBT version of the firmware was designed from the reference of the coming ITk upgrade which requires a radiation hard bi-directional link due to the increased radiation bombardment inherent to the environment of the LHC post upgrade. The GBT protocol combines multiple lower bandwidth links from the front ends into a single data link (running at speeds up to 5 Gbps). This high-speed data link is then interfaced with optical connectivity technology called the Versatile link which provides the radiation hard transport required. The second protocol was the full mode protocol. As interest from other sections of ATLAS as well as external parties in using FELIX grew, the need for a higher bandwidth data link than the GBT could provide arose. The speeds GBT can run at are limited due to the need for radiation hardness, but since these external parties did not require radiation hardness higher transfer speeds could be achieved. The full mode protocol implements a single wide data stream and does not require any special handshaking, so it can achieve speeds up to 9.6 Gbps.

In terms of software, FELIX has a large collection of it. Part of this software is used to initialize and program the firmware. Unlike in YARR/RCE where once the boards are flashed the firmware is static (RCE offers some control post flashing but not as much as FELIX), in FELIX the behavior of the firmware can be modified to suit the needs of the user. One example of this is the elinkconfig program. This program allows

the user to configure the setup of elinks (which manage data transfer to and from the front ends), changing which elinks are active, what data protocol they use, etc. This is just one example and there are lots of tweaks and edits a user can make to customize FELIX to their needs. For the purpose of communicating with an RD53A specifically, the YARR software suite has been integrated into the FELIX software, allowing a user to make use of the YARR's functionality for configuring and probing an RD53A.

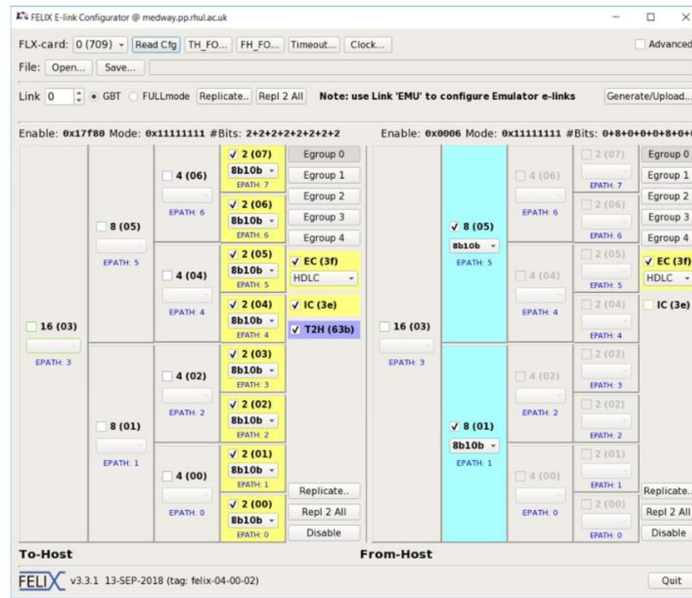


Fig. 25. The FELIX elinkconfig program [10].

## 3.2.2: FELIX Hardware

To host the software the recommended setup consists of a Supermicro X10SRA-F motherboard, with 32 GB of DDR4 RAM and an Intel® Xeon™ E5 family CPU. However, for the setup at the UW the same PC used for YARR/RCE was used as it has a PCIe slot which FELIX requires. There were several reasons for using our PC instead of the recommended one. The first reason is that the new PC is expensive and would be time consuming to acquire. The second reason is there is doubt as to whether or not the higher end PC is required to run FELIX or if it will work just fine on a simpler PC. If it does work on a simpler PC this will expand the number of interest groups who can make use of FELIX as they won't have to purchase a new PC to run it.

To host the FELIX firmware several different boards are required. The first is a VC709. This along with the BNL-712 board are the only two boards FELIX firmware is targeted to so one or the other is required to set up FELIX. Since VC709s were easier for the lab to acquire that board was selected. The necessity of using a VC709 as opposed to something like a KC705 is the four SFP+ cages present on the board. The FELIX setup uses GTX cores and optical cables to transmit data to and from the RD53A instead of display port cables so a board with 4 SFP+ cages is required. The reason for using optical cables over display port is that optical cables can support much higher transfer speeds, up to 10 Gbps, which FELIX requires. Attached to the FMC of the VC709 is a TTCfx mezzanine card. The purpose of this card is to loop a 160 MHz clock back into the system through the GTX SMA ports of the VC709. This is required for the GTX cores to function properly. The second piece of the FELIX firmware is the VLDB board. The VLDB board is connected to the VC709 using an SFP cable and takes in a 4.8 Gbps signal. The VLDB board scales this down to 160 Mbps and then sends the data along with a 160 MHz clock out of G0-link0 using a mini HDMI

cable. Finally, to host the emulator firmware a second VC709 was used. Much like with the first VC709, the emulator needs to have access to 4 SFP cages to transmit data back to the FELIX firmware. To connect the VLDB board to the emulator VC709 a cable chain converting mini-HDMI to mini-display port is used. The mini display port cable is then connected to the Ohio card, which is attached to the FMC of the VC709. To connect the two VC709s four optical cables are used to connect the two sets of 4 SFP cages. Finally, SMA cables are connected from the USER SMA ports to the GTX SMA ports of the emulator VC709 to complete the flow of the clock.

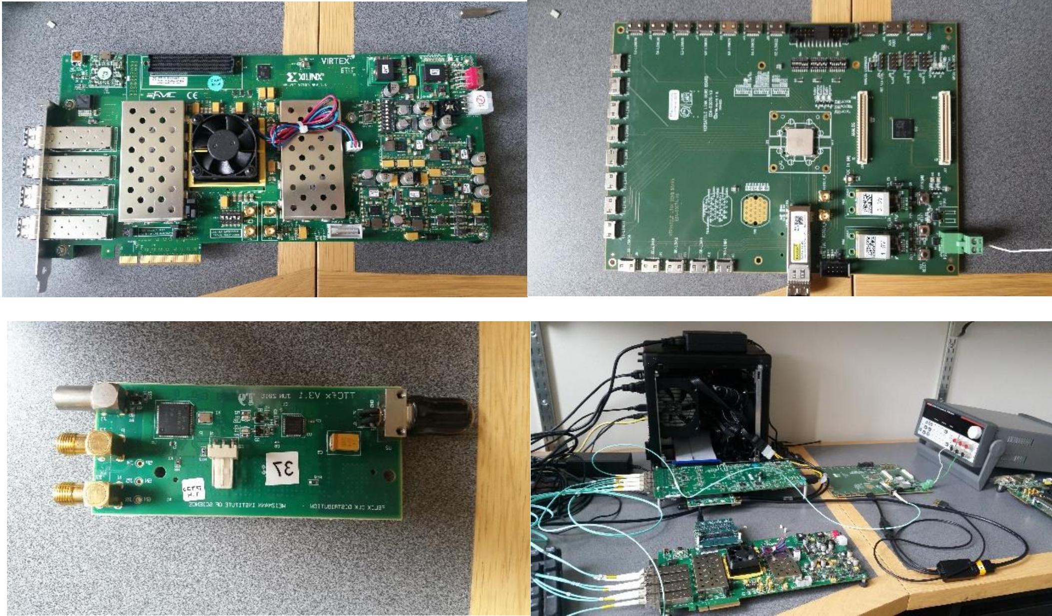

Fig. 26. The FELIX hardware setup. Top left VC709, top right VLDB board, bottom left TTCfx mezzanine card, bottom right full system

## 3.2.3: FELIX Modifications for the Emulator

The main modification made to the RD53A emulator was that the data output module was primarily replaced with a Xilinx IP. Because FELIX uses the SFP cages and optical cables for communication, in order to connect the system to those cages the system output must be routed through a GTX core. These are not the same as the SERDES units used for YARR communication and the only way to connect to them is by using the Xilinx Gigabit Transceiver Wizard. However, using the GT wizard and the SFP cages for output communication provides the benefit that the emulator can now transmit data at 1.28 Gbps and is no longer restricted to 640 Mbps. The GT wizard and the example files provided by Xilinx sets up the four lanes of data communication and automatically handles many of the pieces that were implemented in the custom Aurora implementation such as data scrambling and the conversion process from 66-bit to 32-bit. But there are a few quirks with using the GT wizard that will be discussed.

The first difference between the custom Aurora implementation and the GT wizard is that the GT wizard does not provide a pulse that signals when the system wants new data. Instead it provides a user clock that has a speed of 20 MHz, eight times slower than the speed data blocks are transferred at, as well as a signal that indicates on which cycles of the 20 MHz clock new data should come in. The idea behind this is that in every clock cycle of the 20 MHz clock the GT wizard can process 64 bits of the current 66-bit block so if the system is producing new data in sync with the 20 MHz clock things will work smoothly. To connect the GT wizard with the rest of the system a pulse was created by and-ing the 20 MHz user clock and the

new data signal together. The data framing unit was then adjusted to detect when a positive edge is seen and only input the positive edge as a pulse to the rest of the data framing unit. This change created the needed pulse but introduced another problem. Because the system needs to take a cycle to detect the positive edge this means that the data framing unit is one 160 MHz cycle slow in responding, thus causing problems. To fix this in the "and" statement the 20 MHz clock was replaced with a 20 MHz clock that was phase shifted by 7/8 of a clock cycle. This caused the positive edge to arrive at the data framing unit a 160 MHz earlier and synced up the timing.

The other major change that needed to be made was to the clocking arrangement in the system. Because of how the GT wizard is set up it requires two clocks, a system clock and a GTX clock which must come in from the outside through the GTX SMA's. To meet these clock requirements the 160 MHz clock signal that came with the incoming data was routed back out through the USER SMA ports and then back into the system. The fabric clock generated by the GT wizard was then used as the system clock for the GT wizard. The other major clock change was in the location of the system's PLL. As part of the GT wizard it instantiates its own PLL which is the source of the 20 MHz user clock. To prevent timing issues and reduce the number of clocks sources the additional clocks that the emulator needed were sourced from this PLL and the original PLL was removed. At the time of writing this thesis the FELIX/emulator communication project is still in development and while in theory it should work, more development and time will be required to work out all the bugs.

# 4: Next Steps

At the time of writing this thesis most of the internal work on the RD53A emulator has been completed. The hit data generator is almost completed and with it the flow from the input to the output of the emulator will be finished, though the emulator will continue to be supported in the future. Likely minor bug fixes and tweaks will continue to be made to ensure that the emulator behaves as close to the real chip as possible as well as support the features desired from outside groups. For the big next steps, the first is to continue developing the integration of the RD53A emulator with the FELIX readout system. Currently the back end designed for FELIX should be able to communicate with FELIX, but proper communication has not yet been established. Establishing this communication will require further development of the back-end code and possible new hardware, specially a new host PC.

The second and more prominent next step is the development of an RD53B emulator. Due to the overall success of the RD53A, the RD53 team is moving forward with the development of the RD53B chip which will fix the mistakes and bugs of the RD53A as well as bring in new features to the chip. There have also been significant changes to the data protocols compared to what was used in the RD53A. Since the focus of the community is now shifted towards the RD53B, the emulator project will shift with it to begin work on an RD53B emulator. While there are numerous changes being made, there is also a lot of features and functionality that will remain the same between RD53A and RD53B which means large portions of the RD53A emulator code can be reused and will greatly speed up development time compared to the original RD53A. Because of this the RD53B emulator should be available before the release of the real RD53B chip and will allow readout groups to test their data acquisition system early.

# 5: Acknowledgements

First and foremost, I would like to thank Scott Hauck and Shih-Chieh Hsu, my PIs. They have been immensely helpful as mentors providing advice on both my graduate work as well as general life advice.

I would like to thank all the students in the lab, both graduate and undergraduate, who have worked on the project. This includes Joseph Mayer, Logan Adams, Dustin Werran, Lev Kurilenko, Tony Faubert, Jessica Lan, Michael Walsh, and Niharika Mittal.

I would also like to thank Timon Heim, Alexander Paramonov, and Zijun Xu who have helped me in developing the modification needed to support YARR, FELIX, and RCE.

Finally, I would like to thank my family, who have always supported me through my academic career.

# 6: References

[1]. “CERN website”, CERN, [Online], Available: https://home.cern/

[2]. The Henryk Niewodniczanski Institute of Nuclear Physics Polish Academy of Sciences, “'New physics' charmingly escapes us,” August 3, 2018.

[3]. “ATLAS website”, ATLAS, [Online], Available: https://atlas.cern/

[4]. G Aad et. al., “ATLAS pixel detector electronics and sensors,” Journal of Instrumentation, JINSTI 3 P07007, 2008.

[5]. “LHC commissioning website”, CERN, [Online], Available: https://lhc-commissioning.web.cern.ch/lhc-commissioning/

[6]. “RD53 collaboration home”, RD53, [Online], Available: http://rd53.web.cern.ch/rd53/

[7]. “RD53A Manual v3.5”, RD53 group, Available: https://cds.cern.ch/record/2287593/files/RD53A_Manual_V3-5.pdf

[8]. “YARR Documentation”, LBNL, [Online], Available: https://yarr.readthedocs.io/en/latest/

[9]. “RCE Documentation”, RCE Development Lab, [Online], Available: https://twiki.cern.ch/twiki/bin/view/Atlas/RCEDevelopmentLab

[10]. “FELIX Documentation”, Argonne National Lab, [Online], Available: https://atlas-project-felix.web.cern.ch/atlas-project-felix/

[11]. “RD53A Testing”, [Online], Available: https://twiki.cern.ch/twiki/bin/viewauth/RD53/RD53ATesting

[12]. L. Kurilenko, S. C. Hsu, S. Hauck, “FPGA development of an emulator framework and a high speed I/O for the ITk Pixel upgrade,” June 6, 2018.

[13]. D. Werran, S. C. Hsu, S. Hauck, “Development of an FPGA Emulator for the RD53A Test Chip”, March 22, 2019