© Copyright 2021

Donavan Martin Erickson

Development of a High-Speed Hit Decoder for the RD53B Chip

Donavan Martin Erickson

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

University of Washington
2021

Committee:
Scott Hauck
Shih-Chieh Hsu

Program Authorized to Offer Degree:
Electrical and Computer Engineering

University of Washington

Abstract

Development of a High-Speed Hit Decoder for the RD53B Chip

Donavan Martin Erickson

Chair of Supervisory Committee:

Scott Hauck

Department of Electrical and Computer Engineering

The Large Hadron Collider (LHC) is undergoing an upgrade, called the High Luminosity LHC (HL-LHC), that will increase the data rates produced by particle collisions by tenfold from what they currently are. ATLAS and CMS experiment sites are designing the next generation read out chips named RD53 to be able to handle the increase in data rates. To accomplish the task, an encoding system was implemented in the second trial of chip development known as the RD53B to help shrink data streams and reduce the overall bandwidth of the system. An exploratory effort was undertaken to create a hardware decoder for Field Programmable Gate Arrays (FPGA) to cut down on CPU usage from software decoders later in the system. A parallelized hardware decoder was designed to meet the data rates produced by an RD53B chip. Overall, the final product is a base hardware decoder design that can handle the throughput constraints of a single RD53B and is resource efficient. The necessary background, hardware decoder design, and the decisions made throughout the process will be outlined in this thesis.

Table of Contents

1: Background – 1

- 1.1: Large Hadron Collider – 1
- 1.2: Layout – 3

2: Hitmap Encoding – 7

- 2.1: Initial RD53B Encoding – 7
- 2.2: Updated RD53B Encoding – 10

3: RD53B Hardware Decoder Design – 11

- 3.1: ROM Design – 11
- 3.2: Decode Engine – 13
- 3.3: Decode Engine Timing – 17
- 3.4: Decode Engine Parallelization – 18
- 3.5 Centralized ROM – 19
- 3.6 Initial Full Design – 21
- 3.7 Optimizations – 24
- 3.8 Final Results – 25
- 3.9 Interpretation of Results – 26

4: Conclusion – 28

5: Future Work – 29

6: Acknowledgements – 30

7: References – 31

1: Background

1.1: Large Hadron Collider

Currently the largest and most advanced hadron collider in the world is the Large Hadron Collider (LHC) located on the Franco-Swiss border. The LHC is used to accelerate particle beams at near light speed to collide with each other using magnets. This experiment site is owned by the European Council for Nuclear Research (CERN) and has been used to study the physics interactions that take place during these collisions, which has led to discoveries like the first real world findings surrounding the existence of the Higgs Boson [1].

Figure 1: Arial view of LHC location with super imposed tubing and location of the experiment sites. Source [1]

There are four main particle detection points shown in figure 1 around the collider: ALICE, ATLAS, CMS, and LHCb. This thesis will focus on the ATLAS experiment site which consists of a large layered cylindrical detection system that sits 100 meters below the ground [9]. The area of focus for this paper will be the pixel detector layer which is the inner-most layer of the ATLAS experiment.

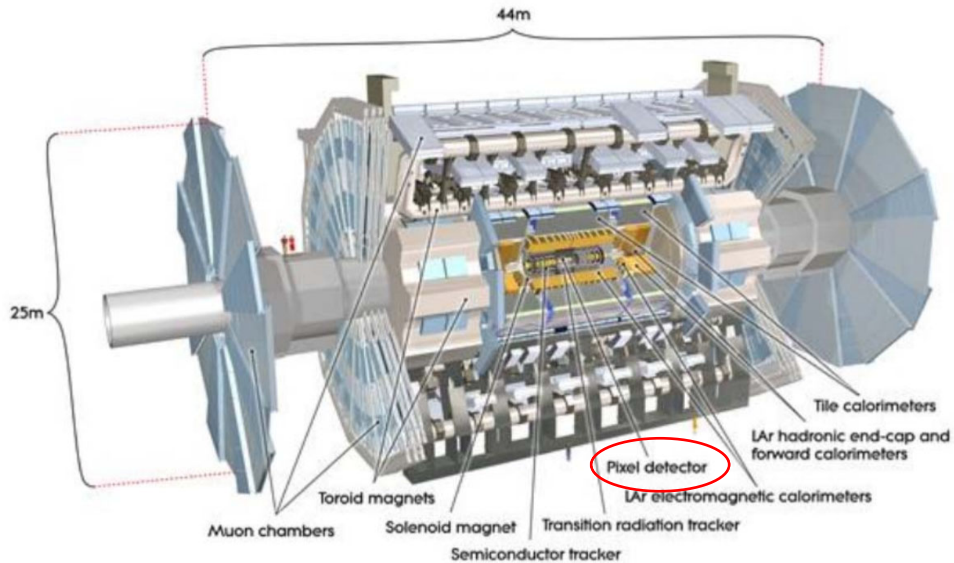

Figure 2: Computer rendering of the ATLAS detector system. Source [1]

The pixel detector, circled in red in figure 2, is the closest region to the particle collisions that tracks the momentum and charge of passing particles [9]. Read out chips that are used for the detection and read out of the collision data inside the pixel detector and are formed into 4 cylindrical layers as seen in figure 3 [8]. Each layer will have a different detection rate with layers closer to the beam capturing more hits than further out layers. One major issue for the ATLAS site is the amount of data that is created by the LHC. Up to 1 billion particle collisions can take place each second, which can create up to 60 terabytes of data per second [1]. This data is composed of "events", which is a snapshot in time for all the particles detected on each read out chip at that instant. To cut down on the amount of data to store, the data is saved using a two-trigger system to filter out less interesting data. The first trigger is a hardware trigger that will decide to save events based data on current detector information that can save up to 100,000 events per second (or only about 0.006% of all events). After the initial triggers, a large set of CPUs will further filter the data sent from the first triggers to keep the most relevant data at a maximum of 1000 events per second [9].

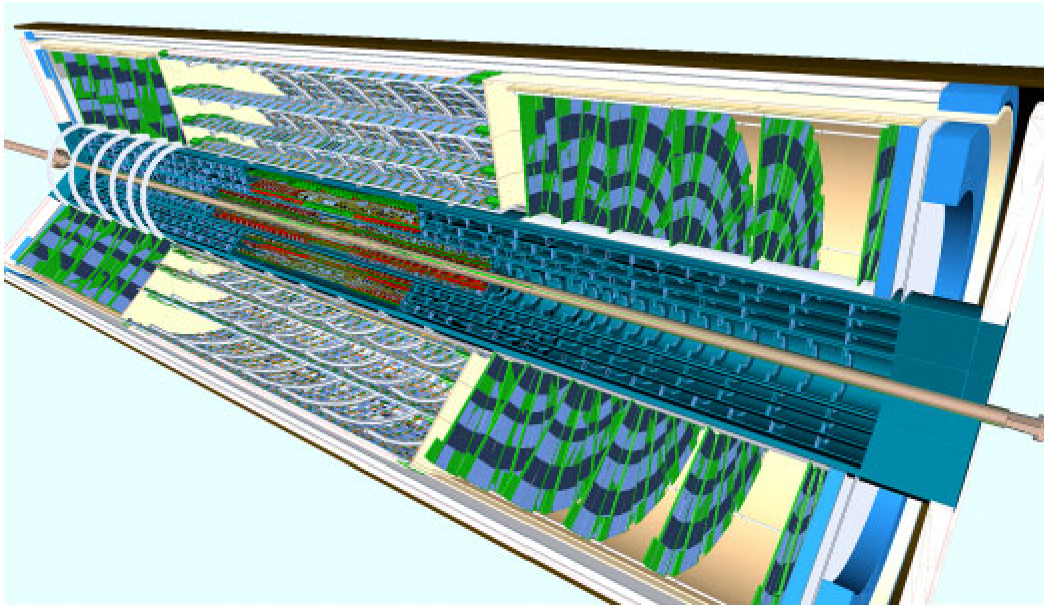

Figure 3: Computer rendering of the pixel detector. Source [8]

The LHC is currently undergoing an upgrade called the High Luminosity LHC (HL-LHC) to increase the luminosity of the LHC which will increase collision rates by a factor of 10 [2]. With increased collisions, the current pixel detector will be replaced by the Inner Tracker (ITk) pixel detector that has 5 cylindrical layers. With the upgrade in the pixel detector, the read out chips needed to be upgraded as well which brought about the RD53 collaboration effort between ATLAS and CMS to create new FE chips that can handle the increased data rate in the ITk pixel detector [10]. The main idea was to use 65 nm technology to create a radiation tolerant and high-speed particle detection device. Although ATLAS and CMS are working together on this project, they have different requirements for their application-specific integrated circuits (ASIC) [3]. This paper will focus specifically on the ATLAS requirements.

1.2: Layout

The RD53B is the second iteration of the RD53 chips. At the base of the design is the pixel sensors which collects charge from passing particles. An analog system is used to determine the charge and compare the charge to a threshold value. When a pixel's charge goes above the threshold value, a comparator on the output of the pixel analog logic goes high and the pixel is considered to have been "hit". Digital logic is used to read and save these hit values from the comparator. With the combination of analog and digital logic the design of the chip resembles an "analog island" in a "sea" of digital logic [4].

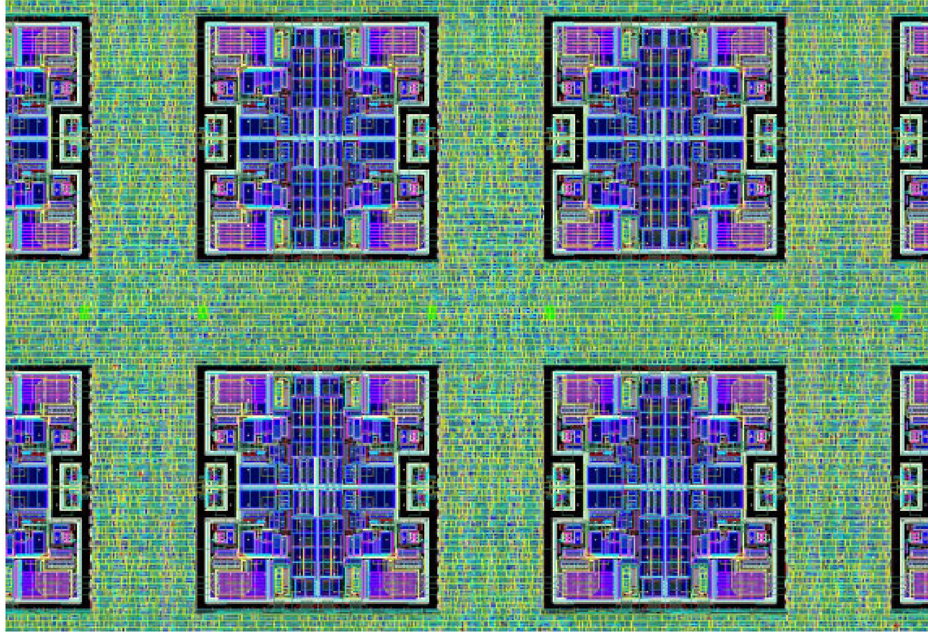

Figure 4: Layout of analog islands with surrounding digital logic. Four complete islands are seen in the middle of the image. Source [4]

Each analog island in figure 4 consists of 4-pixel sensors. The islands are then used as a template and placed in the chip surrounded by digital logic that creates the “sea”. Along with registering the hits from each pixel the digital logic produces the time-over-threshold value. Each pixel will have a corresponding time-over-threshold (ToT) value that denotes how long the pixel has been over the threshold. To create the larger chip, these analog islands are made into cores. A core consists of 64 pixel sensors (or 16 analog islands) put into a square format [4].

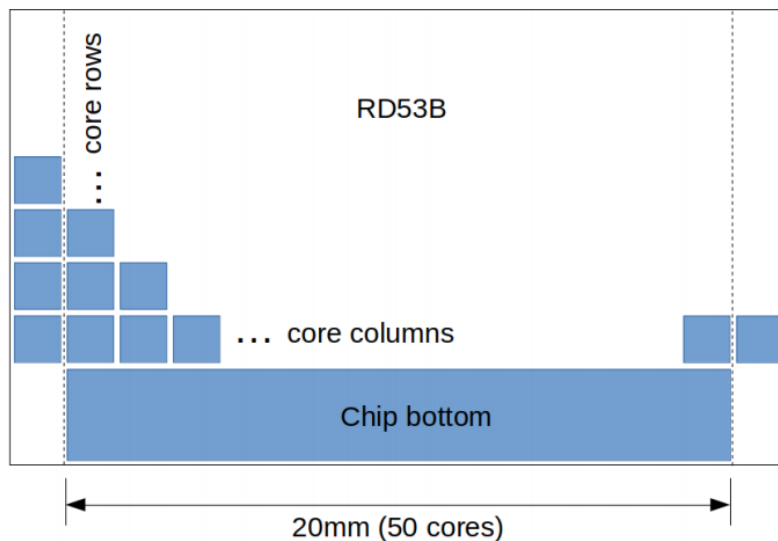

Figure 5: Diagram of core layout of the RD53B chip. Source [4]

Figure 5 illustrates how the cores are used as a template and placed on a chip in a grid format which allows for configurability of the chip. The chip bottom handles the system functionality and is a standard feature

across all the configurations of the RD53B which creates a standard minimum width for the chip which is 50 cores. The amount of core columns and core rows are variable depending on the use case [4].

Table 1: Chip requirements for ATLAS and CMS versions of RD53B. Source [4]

Parameter	**ATLAS**	**CMS**
Pixel bump pitch	$50\,\mu$m $\times$ $50\,\mu$m	
pixel rows (H)	384	336
pixel columns (W)	400	432
core rows	48	42
core columns	50	54
Chip width (including seal ring)	20.054 mm	21.654 mm
Chip height (including seal ring)	21.022 mm	18.622 mm

This paper will focus on the ATLAS version of the RD53B, so the chip will have 50 core columns and 48 core rows as shown in table 1. Each core column will consist of eight pixels across, and each core row will consist of eight pixels down. Information sent out from the chip consists of pixel hit data which is saved and stored for later research use. For data output, the pixel hits are broken up into quarter rows which consists of a two by eight block of pixels. Data will only be sent out for quarter rows that contain at least one hit. An illustration of a quarter row selection is shown in figure 6.

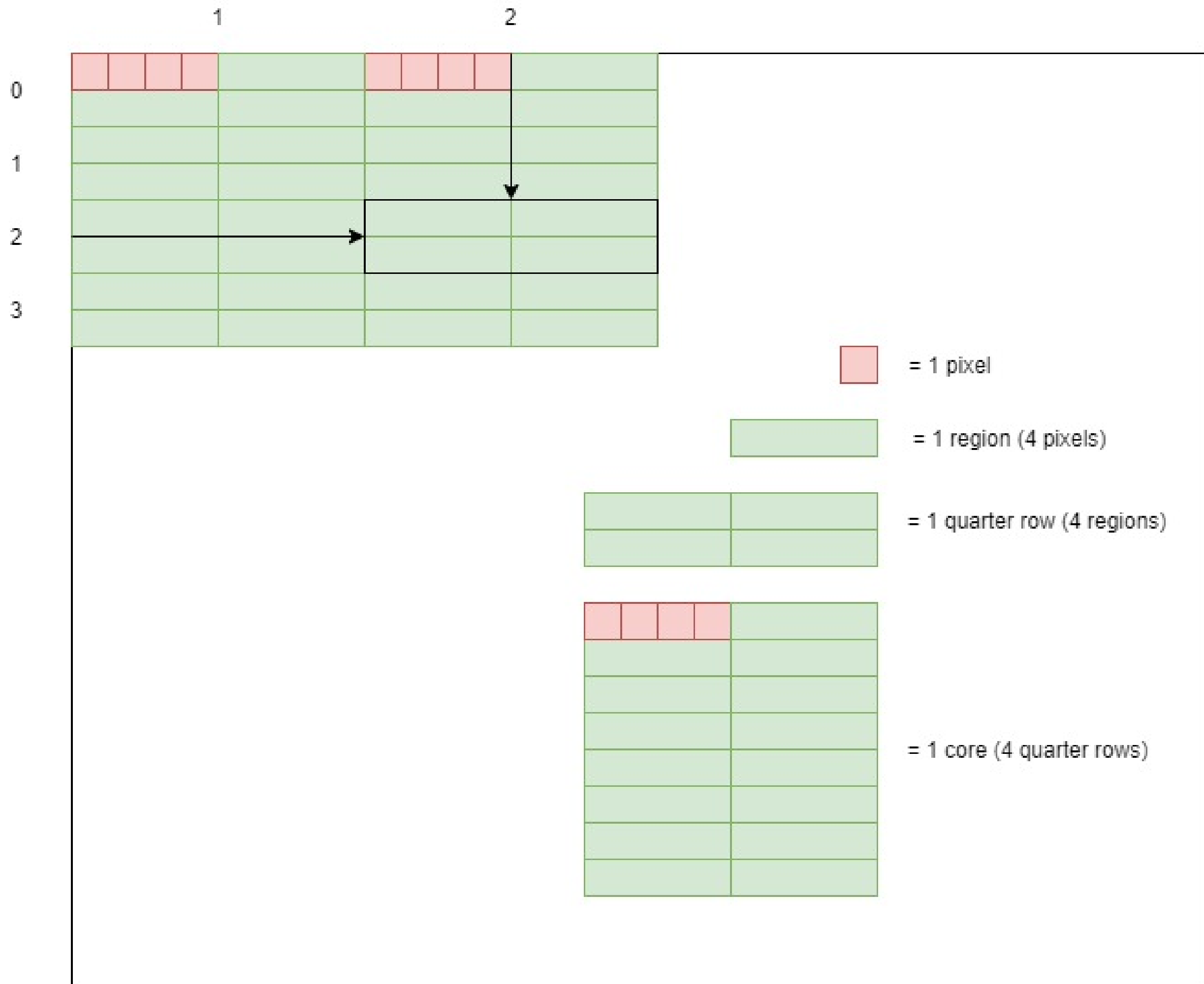

Figure 6: Visualization of RD53B quarter row selection process. If there is at least one hit in core column 2 and quarter row 2, then the entire quarter row will be selected.

During a bunch crossing, the entire chip is read and all the quarter rows with hits will be assembled into an event. A bunch crossing is the time at which two beams in the LHC will intersect and have particle collisions, which happens in 40 ns intervals [4]. The format for each quarter row with hits will look like figure 7.

6 bits	1 bit	1 bit	8 bit	5-30 bits	4-64 bits
Core Column	Is Last	Is Neighbor	Quarter Row	Hitmap	ToT

Figure 7: Quarter row data format in context of a stream.

Streams are used to pack and send the data out from the RD53B chip. A stream will pack data into 64-bit packets, as shown in figure 8, which consist of all the quarter row hit data sorted numerically by Core Column Address (Ccol) and then by Quarter Row Address (Qrow). In the RD53B manual, there are sections that describe having multiple events per stream, but recently there has been a shift to having one event per stream. With the change becoming a strong possibility, the RD53B hardware decoder was built using the assumption that there will be one event per stream [7].

Figure 8: Example RD53B stream data. Source [4]

Each packet in a stream will have a start-of-stream bit. This will signify if the packet is the start of a new stream or the continuation of a previous stream. When a new stream happens, the following eight bits will be tag data. This tag data is used to create an identifier for the stream's data. A stream is terminated when there are six consecutive zeros in a stream [4].

The Ccol and the Qrow are used to specify the address of the specific quarter row of data in the chip. Since hit data can be clustered, the encoding scheme has mechanisms to avoid sending the Ccol and Qrow bits for neighboring locations. The islast bit is used to give data about the next quarter row in the stream. When the is last bit is set to one, this signifies that the current quarter row data is the last one in that core column and the next quarter row will have a different Ccol. When the is last bit is set to zero, that means that the next quarter row is in the same column and the next quarter row data will not have a Ccol value. The isneighbor bit specifies if the current Qrow is one more than the previous Qrow, in which case there will not be a Qrow value for the current quarter row data. The Qrow value will be assigned to one plus the last Qrow value [4].

The hitmap contains data that specifies which pixels were hit in the specified quarter row. Hitmap encoding was introduced in the RD53B to help cut down on data bandwidth. The encoding causes the hitmap to vary in size between 4 and 30 bits. The specifics of the hitmap encoding will be covered in section 2: Hitmap Encoding. There will be four ToT bits for each pixel with a hit, which can be anywhere from 4 to 64 bits [4].

2: Hitmap Encoding

2.1: Initial RD53B Encoding

To encode a quarter row of data a system of "cuts" is used on the 16-pixel quarter row to determine the encoded hitmap. The cuts are used to determine the position of any hits in a quarter row. Each cut narrows down the position of a hit [5].

In figure 9 there is one hit in the entire quarter row. The cutting process is used to denote the location of the hit in the quarter row. In the first cut, the only valid area is to the left of the vertical cut because the left half of the quarter row has a hit. As sequential cuts continue, the area continues to be halved until the location of the hit is found in the fourth cut.

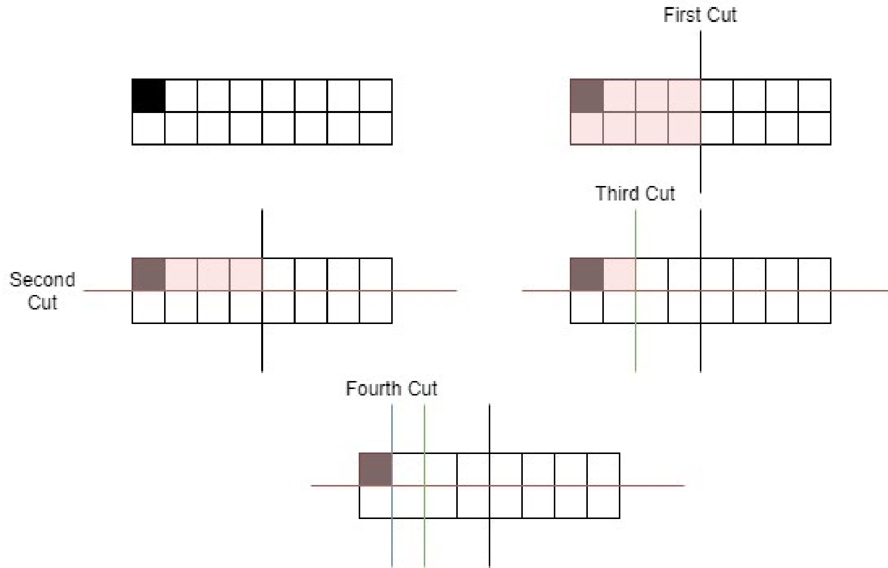

Figure 9: Initial hitmap encoding cut process.

This format for doing the cuts is accompanied by corresponding bits in the encoding. For a vertical cut if there is a hit on the left, then the first bit will be a one and if there is a hit on the right, then the second bit will be a one. The same is true of horizontal cuts, where the top bit is first, and the bottom bit is second. Only areas that have a hit in them will continue with the cutting pattern, and this will result in a binary tree [4].

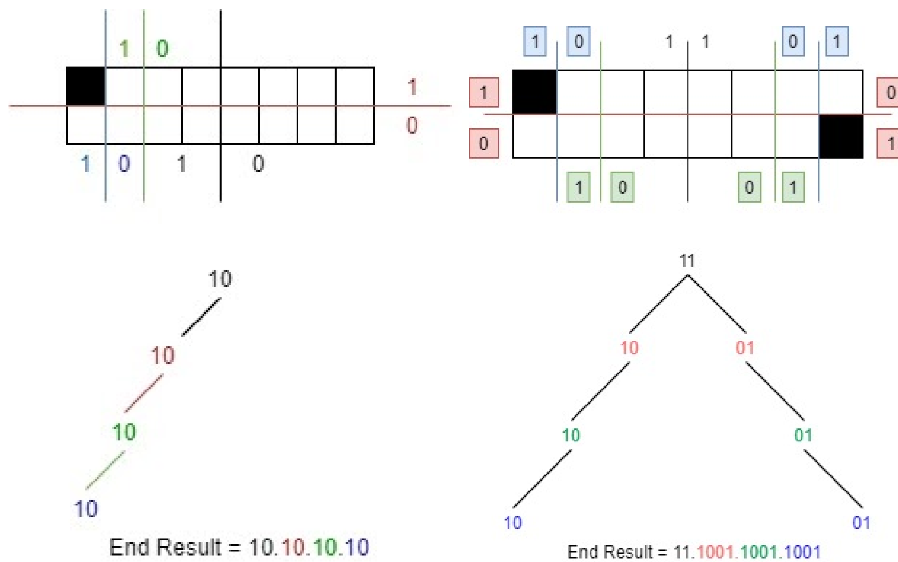

Figure 10: Initial hitmap encoding binary tree representation.

In the right side of figure 10, there are a series of four cuts done to represent the one hit. This follows the same cut pattern as shown in figure 9**.** In the left side of figure 10, there are now two hits that need to be accounted for. The first cut will yield that both the left and right regions are valid regions because both contain hits. A separate cut pattern must be done for the left region and the right region to locate each hit separately.

On the left is an example of a basic one hit encoding. We assemble the values into the binary tree formation, where each cut (first, second, third, fourth) is represented by a level on the tree. The diagram on the right shows what happens when we have multiple pixels with hits. To determine the end result, shown at the bottom of each tree, we group together the values at each level of the tree going from left to right. This will give us a value for which pixels have hits in them.

After hitmap encoding, a simple Huffman compression is performed to further decrease the size of the data. Encoded hitmaps will only report pixels that have a hit, so there will never be a case of 00 in our encoding map; a 00 would mean that there were no hits on either side of a cut [5]. This allows for bit replacement with the value of 01 becoming 0 to shrink the encoded hitmap even more. I will use the example from figure 10 on the right to do the encoding.

Swapping 01 with 0

11.100.100.100

From this we have created a compressed version of the data that we can now send. This data will need to be decoded to be usable again.

This is where the idea for a hardware decoder for a field programmable gate array (FPGA) began. Since there is a compression to cut down on the bandwidth usage, the LHC will need a fast and efficient way to decode the data to become usable again. Based on the above version of the encoding system, we will detail our trials in creating a hardware decoder.

The major challenge in decoding the encoded hitmap is that the meaning of individual bit depends on the preceding bits in the encoding. That is, despite the examples having levels of data separated by periods in the end result, the decoder needs to recover this information from just the 0's and 1's. For example, in the encoding 11100100100 the first 11 means there is a hit in both the left and right sides of the first cut, and thus the next 2-4 bits will represent the next level. The next 10 means a hit on the top row of the left side, while the following 0 is a Huffman compressed 01, which means a hit on the bottom row of the right side. Only by figuring out that the 100 is the 2nd level of cuts, can we then begin to figure out what the following 100100 mean.

The first thought was to go sequentially through the bits performing decoding. The idea is to look at each individual bit pair, understand what it means, and use that to determine where the hitmap ends and ToT data begins. This method would work, but the amount of time to decode an encoded hitmap would take too long because most bit pairs would have to be decoded sequentially.

A second idea was to use read only memories (ROM) tables to decode the hitmaps by creating a lookup table that took encoded hitmaps as input and output decoded hitmaps. The issue is that encoded hitmaps can be anywhere from 4 bits to 30 bits. In this case, a ROM would have to be able to take a 30-bit input to encompass all possible encoded hitmaps and have a 16-bit output where each bit corresponds to one pixel in a quarter row. A 30X16 ROM will take up 2GB of memory, which is too large to fit into any FPGA. We considered breaking the encoded hitmap into multiple parts where smaller ROMs could handle the decoding; however, the cutting of the encoded hitmap into usable pieces was not reasonable because of the sequential nature of the encoding.

After looking into a hardware decoder for the above encoding method, the process was not deemed reasonable and was abandoned.

2.2: Updated RD53B Encoding

The hardware decoder project was revitalized when a new RD53B encoding structure was implemented. As shown in figure 10, each row of the hitmap tree would be placed next to each other in the end result. The new encoding scheme starts with the first two bits which give information about which rows have hits, followed by all the first row hit data, and then all the second row hit data.

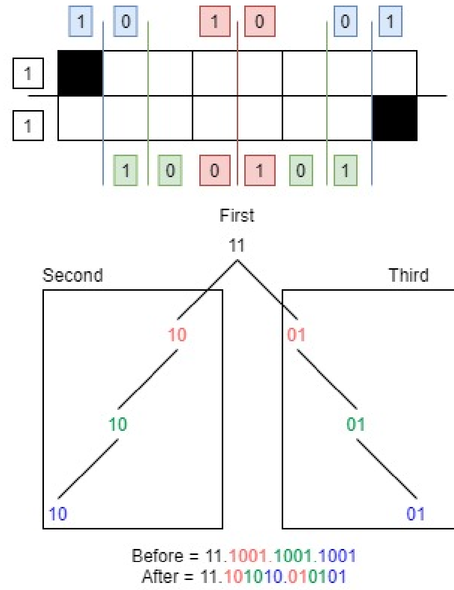

Figure 11: Updated encoding pattern with binary tree.

Figure 11 contrasts with the example from the left of Figure 10. The first cut is now a horizontal cut which separates the areas into top row and bottom row. The subsequent cuts will be vertical and divide either the top row or bottom row pixels. With the updated encoding the end result will become 11.101010.010101 from 11.1001.1001.1001. Instead of the first quarter row divide being a vertical cut, the first two bits will be represented by a horizontal cut. This allows the first two bits to signify if there is a hit in the first row, second row, or both. With the new encoding, the next set of bits will encompass the left half of the tree followed by the right half. The left half of the tree will contain bits from the top row cuts while the right half contains bits from the bottom row cuts [5].

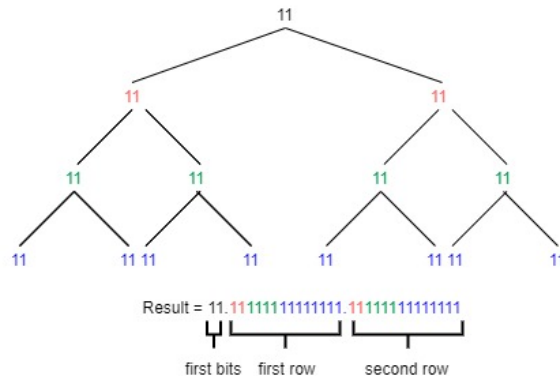

Figure 12: Encoded hitmap with maximum number of values.

The binary tree in figure 12 shows an encoded hitmap with the maximum number of values which corresponds to a quarter row where every pixel has a hit. The maximum amount of cut data that each row can hold is 14 bits as shown by the left and right part of the tree. With this new encoding, the first sixteen bits are always guaranteed to contain the first two bits which indicate the rows that have hits, followed by at most 14 bits worth of first row data. The second guarantee is that each row will never have more than 14 bits. These two guarantees allow us to revisit the idea of using ROMs to do the decoding in an efficient fashion. It is now feasible to use a 16-bit ROM to decode the first 16 bits because the first 16 bits will always contain the information about which rows have hits and the top row data. Since we are decoding at least the first 16 bits, the top row will always be decoded by the 16-bit ROM. This ROM can also tell us the starting position, in the encoded data, for the 14 bits for the bottom row. This leaves a maximum of 14 bits for the bottom row, which enables the ability to use a 14-bit ROM to decode the bottom row. Using the two ROMs in combination will allow us to decode any hitmap with some additional meta data.

3: RD53B Hardware Decoder Design

3.1: ROM Design

The two ROM design consists of one 16x32 ROM and one 14x16 ROM as shown in figure 13. The total size of these two ROMs in combination is 288 kB which is a much more convenient size than the 2GB ROM mentioned in the initial RD53B encoding section. This ROM design is borrowed from Hongtao Yang, a postdoc working at LBNL ATLAS group, who created the design as look up tables in software which we modified to create ROMs instead of look up tables. The original code can be found here: https://gitlab.cern.ch/yanght/rd53bDataEmulator/-/blob/master/tool/generateBinaryTreeLUT.cpp.

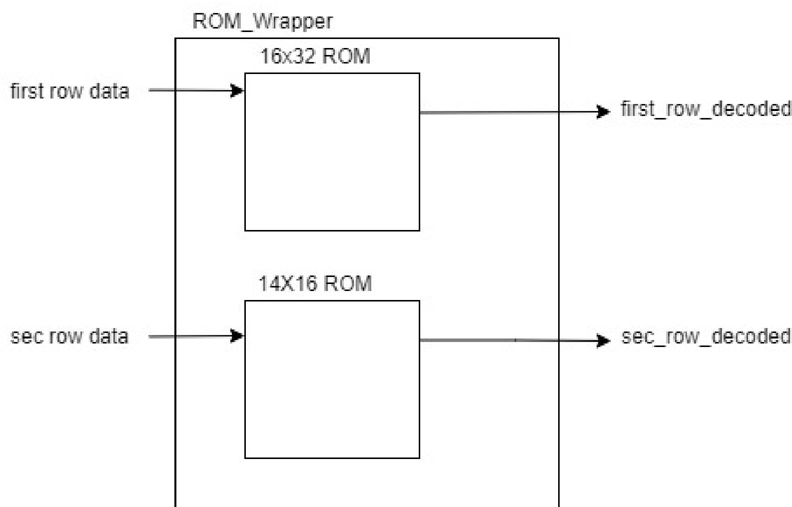

Figure 13: High level block diagram of split ROM system.

The encoded hitmap data will be received as part of a stream as discussed in the layout section. When the stream data comes in, the initial position of the encoded hitmap is located as shown in figure 14, but since the encoded hitmap can be anywhere from 4 to 30 bits, the two ROMs contain extra data on the outputs to figure out the size of the encoded hitmap. The rest of the stream cannot continue to be processed until the size of the encoded hitmaps are found because without that information, whatever is processing the data does not know what is hitmap data and what is general stream data.

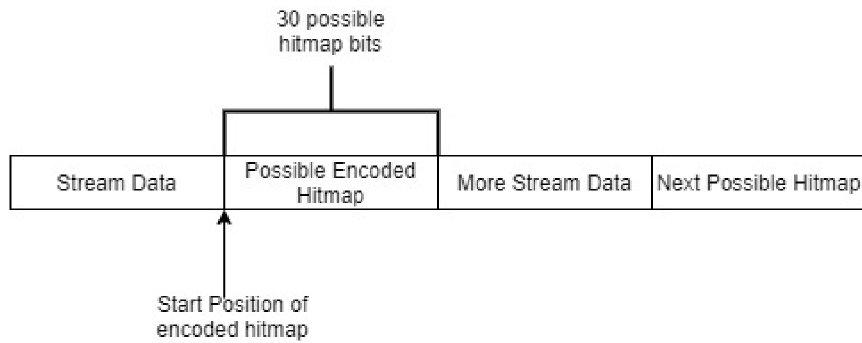

Figure 14: Example showing the possible position of a hitmap within the context of a stream.

Once the initial position is found, the first 16 bits of the encoded hitmap is sent to the 16x32 ROM. The ROM will then send back a decoded hitmap along with a rollback value which will specify how many of the initial bits are not part of the top row in the format shown in figure 15. There are two rollback values, one to specify the rollback if the encoded hitmap is less than 16 bits and another rollback value if the encoded hitmap is more than 16 bits.

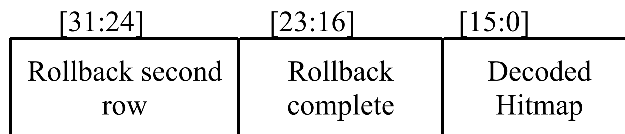

Figure 15: Format for 16x32 ROM output.

Bits [15:0] corresponds to one bit per pixel from the quarter row where a 1 signifies a hit. If the encoded hitmap is smaller than 16 bits, then the 16x32 ROM can process the entire encoded hitmap and will return a full 16-bit decoded hitmap. When an encoded hitmap is larger than 16 bits, the data will need to be split across the 16x32 ROM and the 14x16 ROM. If the encoded hitmap is longer than 16 bits only the first-row data will be decoded by the 16x32 ROM. In this scenario, the upper 8 bits of the decoded hitmap will be zero and bottom 8 bits will return data which signifies the first-row data. The upper 8 bits which represents the second row will later be filled in by the 14x16 ROM.

Bits [23:16] of the output corresponds to the rollback if the encoded hitmap is less than 16 bits and can be decoded entirely by the 16x32 ROM. If the encoded hitmap is larger than 16 bits, then this value will be 0. For example, if the encoded hitmap is 6 bits long, then the rollback value will be 10, so 16 – 10 = 6. With this information, the size of the hitmap is now known, which also signifies where the next part of the stream data begins.

Bits [31:24] is the rollback amount if there is a second row of data and the encoded hitmap is over 16 bits, which specifies the start position of the second-row data. For example, if the encoded hitmap is 20 bits, then the 16x32 ROM will decode the first-row data. If the first-row data plus the two initial row bits is 13 bits long total, then the rollback value will be 3. This allows the user to find the end of the first-row data via 16 – 3 = 13, so the start of the second-row data will start where the first-row data ends. To decode the second row, the next 14 bits after the second rows starting position are sent to the 14x16 ROM because this is the maximum possible number of bits. A diagram of how this would look in a stream context is shown in figure 16.

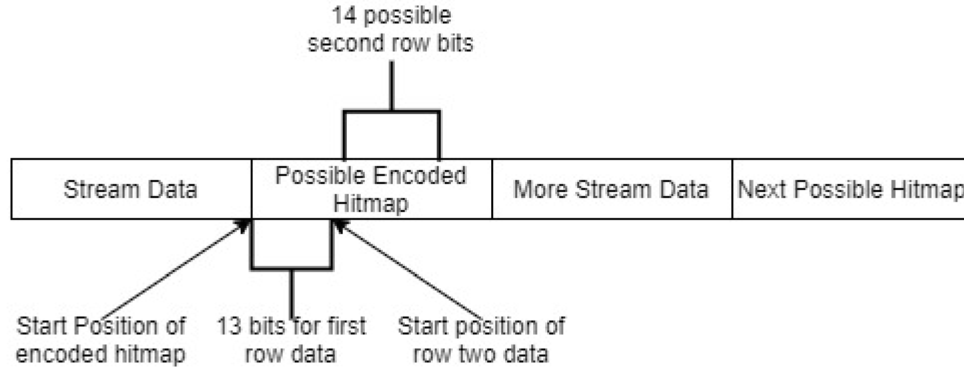

Figure 16: Example showing possible position of second row data in context of a stream.

With the start of the second-row data found, as shown in figure 16, the next 14 bits of data is sent to the 14x16 ROM. This ROM will send back an 8-bit decoded hitmap that represents the second-row data and 8 bits of rollback data to determine the size of the second-row encoded hitmap with the format shown in figure 17.

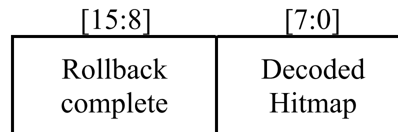

Figure 17: Format for 14x16 ROM output.

Bits [7:0] gives the second row of the quarter row where each bit corresponds to one pixel in the row. By doing an OR operation of these 8 bits of second row decoded hitmap with the upper 8 bits of the 16x32 ROM decoded hitmap, a complete decoded hitmap is formed.

Bits [15:8] of the output will be the rollback value of the second-row data, which determines where the second-row data ends. Since the second-row data can be up to 14 bits, then the rollback is subtracted from 14 instead of 16. To finish off the example above with a 20-bit encoded hitmap, the rollback value needs to be calculated to find the end of the second-row data. In the previous section, it was stated that the first-row data was 13 bits, so the second-row data will be 7 bits. After decoding, the rollback value of 7 will be sent back and we find that the end position of the second-row data is 14 – 7 = 7. With all the information from the 16x32 ROM and 14x16 ROM, the final size of the encoded hitmap can be calculated to be 20 bits by using both rollback values (16 – 3 + 14 – 7 = 20). Now that the size of the encoded hitmap is known, the start of the next piece of stream data can be determined and can continue processing.

3.2: Decode Engine

With the issue of hitmap decoding and size taken care of by the split ROM system, the challenge becomes the data stream's inconsistent length of data from quarter row to quarter row. To solve this problem the decode engine needs the ability to buffer data, parse the data, and format the outputs. Here, we will discuss the structural explanation behind the decode engine.

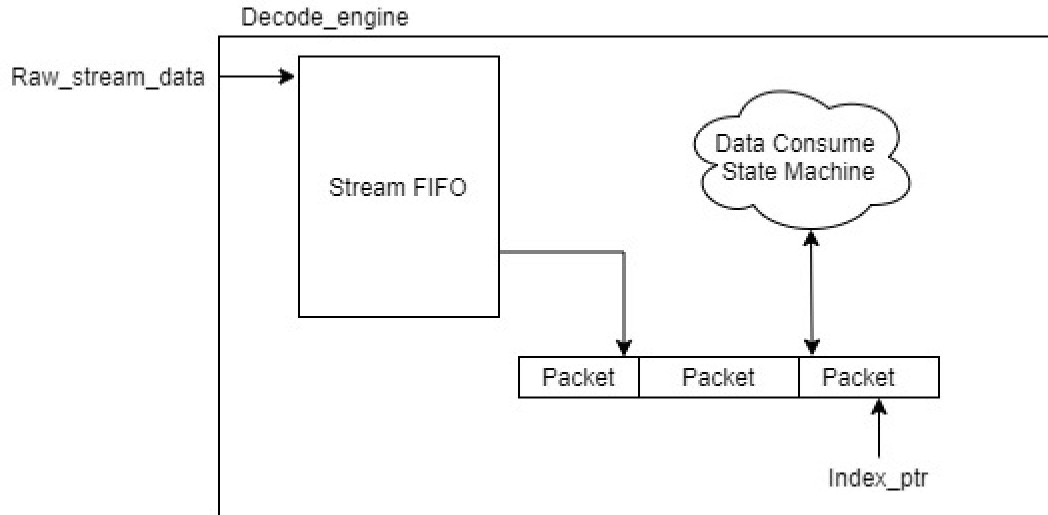

Figure 18: Decode engine high level input diagram.

To start, we will discuss the ingress and parsing of the data as shown in figure 18. A raw data stream is accepted by the engine and stored in a FIFO. Raw data currently has the start-of-stream bit stripped off the data packet before entering the FIFO. This creates a continuous stream of data, which changes the size of the incoming data from 64 bits to 63 bits for each packet. After the FIFO is a three-packet buffering system in which the data will be read from the FIFO. A three-packet buffer is used to allow access to more than one 63-bit amount of data at a time. As data continues across packet boundaries, it is important to create a buffer that allows for continuous data parsing without having to wait for the FIFO to complete a piece of data.

An index pointer signals to the DCSM (data consume state machine) the location of parsed data in the buffer. As the DCSM intakes data, it will continually update the pointer to provide the start position of the next piece of data to be consumed. For example, to register the tag value, the DCSM would check the position of the index pointer and read the next eight bits. The DCSM would then update the index pointer to be the original position plus eight.

When the index pointer becomes greater than one packet worth of data, the packet that has been processed will be dropped. The following packets will be shifted over, and a new packet brought in from the raw data FIFO. The index pointer updates by subtracting the length of one packet from the index pointer's position, allowing for the continual refreshing of data without process stoppage.

The size of three packets of data gives a buffer size of 189 bits or three 63-bit packets. This length was chosen so that the data engine could calculate the start of the next hitmap from the end of the previous hitmap. Using the worst-case scenario, a hitmap would end on bit 63 of the buffer and have a ToT value of 64 bits and the next quarter row would have an address value of 16 bits. This indicates that the index pointer needs to be able to process data from index 63 to index 140 leading to a necessary buffer size of at least 140 bits. Since the buffer needs to be packet size aligned to grab data easily from the FIFO, the closest value is three packets worth of data at 189 bits.

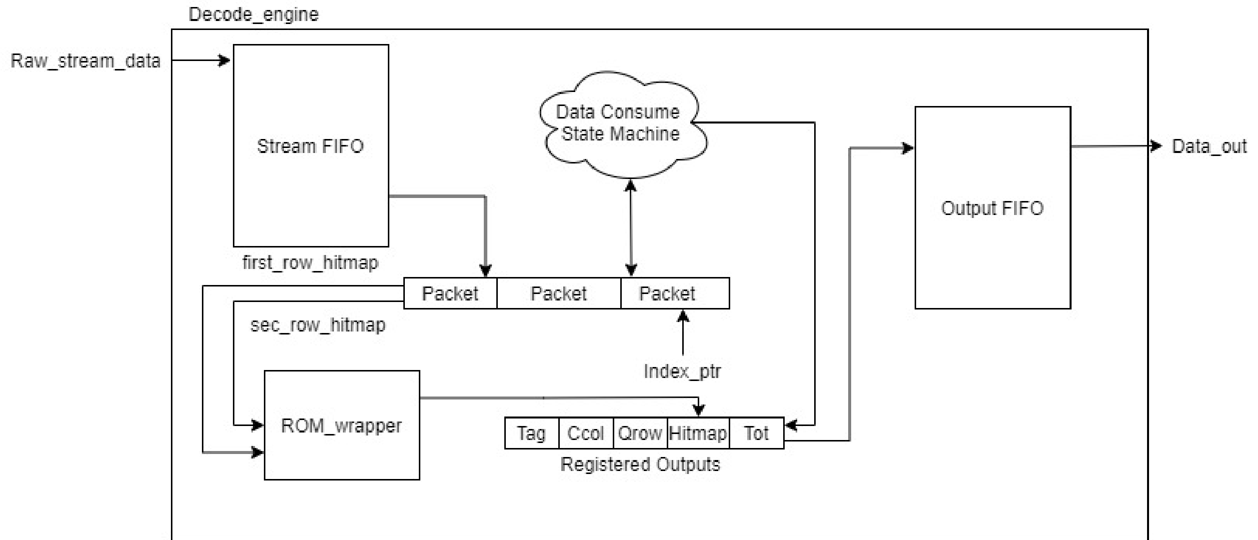

Figure 19: Decode engine high level full diagram.

Next, the split ROM system and output registering process is evaluated as shown in figure 19. As previously stated, the DCSM controls the index pointer allowing the DCSM to manage the data flow. As the state machine processes the data, it will take each section and put them into a set of registers to hold until a complete quarter row has been decoded. The DCSM functions as shown in the following diagram.

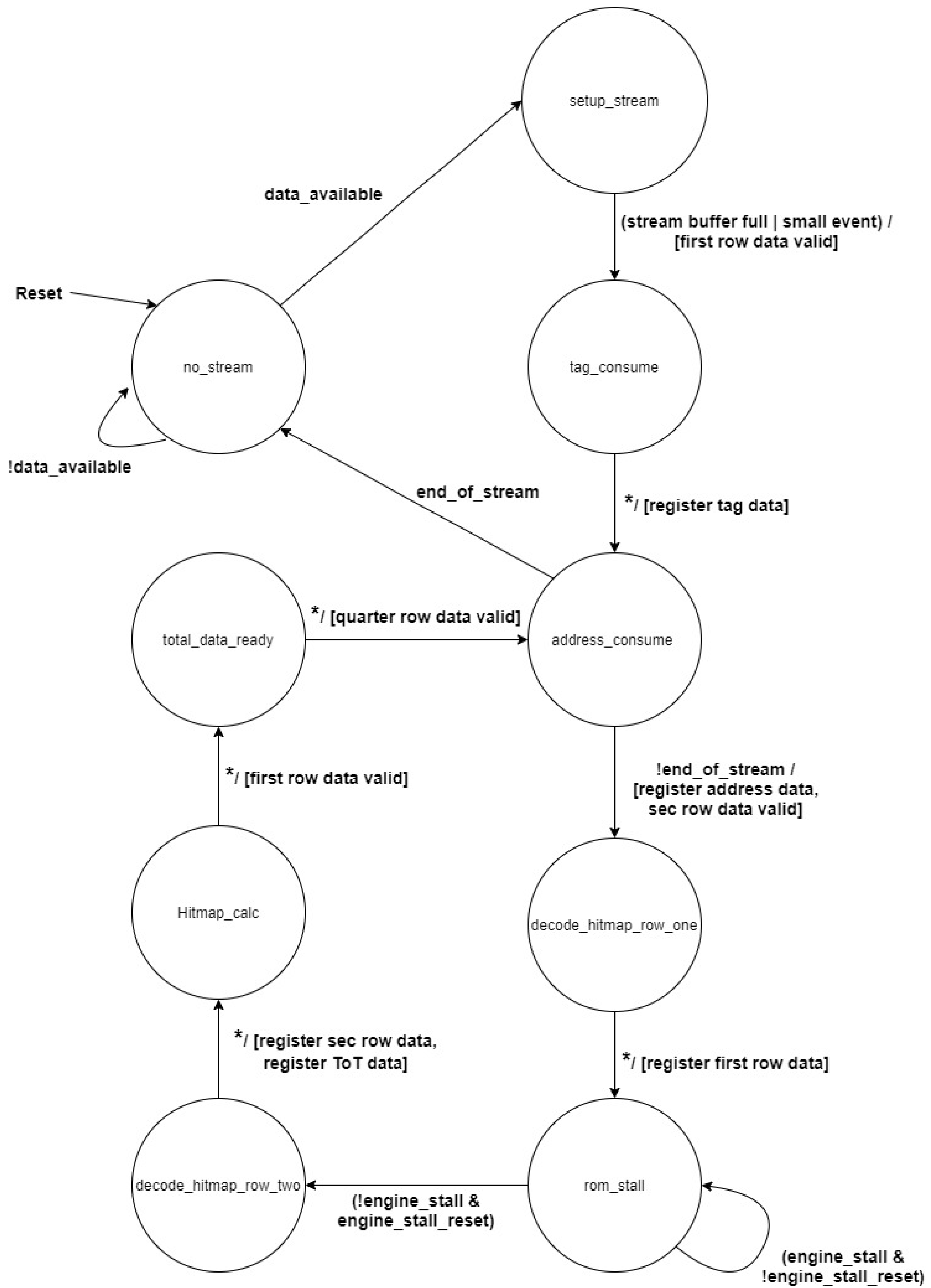

Figure 20: Data consume state machine diagram.

Figure 20 shows the different states in the DCSM starting with setup stream and tag consume states that only run at the beginning of a new stream. The setup stream is used to initially fill the buffer one packet at a time. Tag consume is used to grab the tag data of the new stream and calculate where the first hitmap starts to send to the ROMs. After these initial states, the main body of the data consume state machine takes six cycles which are used to calculate one quarter row worth of data. This six-cycle limitation is due to the ROM processing times. For the main body of the state machine, the first 16 bits hitmap is calculated in the hit calc state, and the processed data is received in the decode hitmap row one state. The second row hitmap data is calculated in the decode hitmap row one state and processed data is received in the decode hitmap row two state. The delay between calculation and results is due to ROM registering times.

Not all encoded hitmaps will be longer than 16 bits, so the second row ROM may not be used. The decision was made to standardize the hitmap decode process so that in future implementations of the design pipelining and c-slowing may be possible. This decision will be explained more thoroughly in the ROM centralization section.

Once all outputs for one quarter row are registered, the data is sent and stored in an output FIFO which preserves the order that the data comes in. The FIFO works on a ready-valid handshake that advertises when the quarter row data is ready and holds this signal high until a valid signal is received from the downstream module.

The output format for the data is currently {26'b0, 8'tag, 6'ccol, 8'qrow, 16'decoded hitmap, 64'ToT}. ToT bits will always be 64 bits long to save hardware resources. A downstream module takes the number of bits in the decoded hitmap and selects which parts of the ToT are valid. For example, if there are four ones in the decoded hitmap, then the 16 most significant bits in the ToT are valid. This bit ordering was chosen to have all the data easily and readily available, and totals 128 bits per quarter row. When building this engine, the assumption was made that there is infinite bandwidth on the output; however, when an output bandwidth is specified, the output format will likely change.

3.3: Decode Engine Timing

After the completion of the decode engine design, a timing analysis was performed to figure out the throughput of a single engine. The desired throughput is 5.12 Gb/s which comes from the output bandwidth of the RD53B chip. The RD53B uses a 4-lane system with 1.28 Gb/s of throughput for each lane which is the maximum possible data rate for the RD53B [5]. To investigate the worst-case scenario, the value of 5.12 Gb/s was used as the target value.

The best clock rate that the decode engine was able to achieve was 160 MHz, which is one clock cycle every 6.25 ns. After setup delay when initially reading in a stream, the DCSM takes six cycles to decode one quarter row's worth of data. To decode one quarter row worth of data it takes 6.25 ns * 6 cycles which equals 37.5 ns per quarter row.

Notably, an issue with consistent throughput determination is that the number of bits in a single quarter row of data can vary drastically. A quarter row can be anywhere from 10 bits to 110 bits, leading to a range of throughputs for a decode engine.

Table 2: Timing estimates for decode engines based off of the number of bits per quarter row.

Ccol + Qrow size	Hitmap	TOT	Total	Speed Per Engine	# Engine Needed
2 bits (islast and isneighbor active)	4 bits (1 hit, most compressed)	4 bits (1 hit)	10 bits	10bits/37.5ns = 266.66Mb/s	5.128/0.26666 = 19.23 (20)
16 bits	8 bits (Arbitrary number)	4 bits (1 hit)	28 bits	28bits/37.5ns = 746.666Mb/s	5.128/0.7466666 = 6.867 (7)
16 bits	20 bits (Arbitrary number)	16 bits (4 hits)	52 bits	52bits/37.5ns = 1.38666Gb/s	5.128/1.3866 = 3.698 (4)

Looking at the worst-case scenario from table 2 of a 10-bit hitmap, the throughput of a single engine is roughly 267 Mb/s. To achieve a throughput of 5.12 Gb/s while only processing the smallest sized quarter rows, 20 decode engines working in total are necessary. The main takeaway from this calculation is that more than one decode engine is necessary. To achieve better throughput, the project moved to engine parallelization.

3.4: Decode Engine Parallelization

Since the throughput of one decode engine would be too low, the decision was made to make parallel decode engines. Each engine will operate independently, processing one stream at a time. This parallelization was achieved by creating a parameter to allow for any number of engines, which helps with the throughput needs while also allowing for customization for different applications. If there is a more resource sensitive application with a lower bandwidth, then the number of engines can be reduced to fit that need.

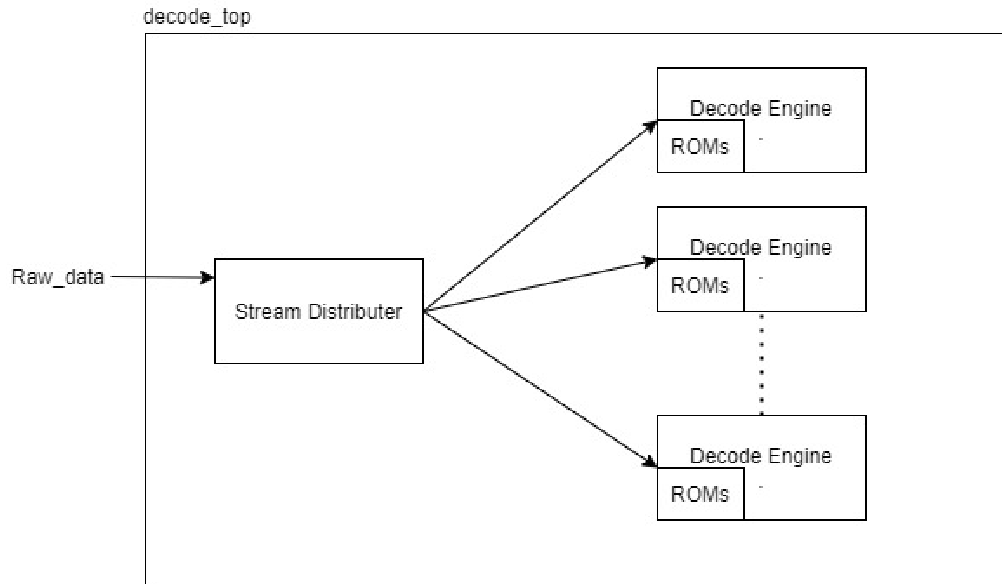

Figure 21: High level diagram of stream distributer with multiple decode engines.

To accommodate the decode engine parallelism, a stream distributer module, shown in figure 21, is added to send streams to multiple decode engines. The function of the module is to take in raw data and then distribute the data in chunks of one stream per engine. Additionally, the stream distributor module strips off the start-of-stream bits from each packet received.

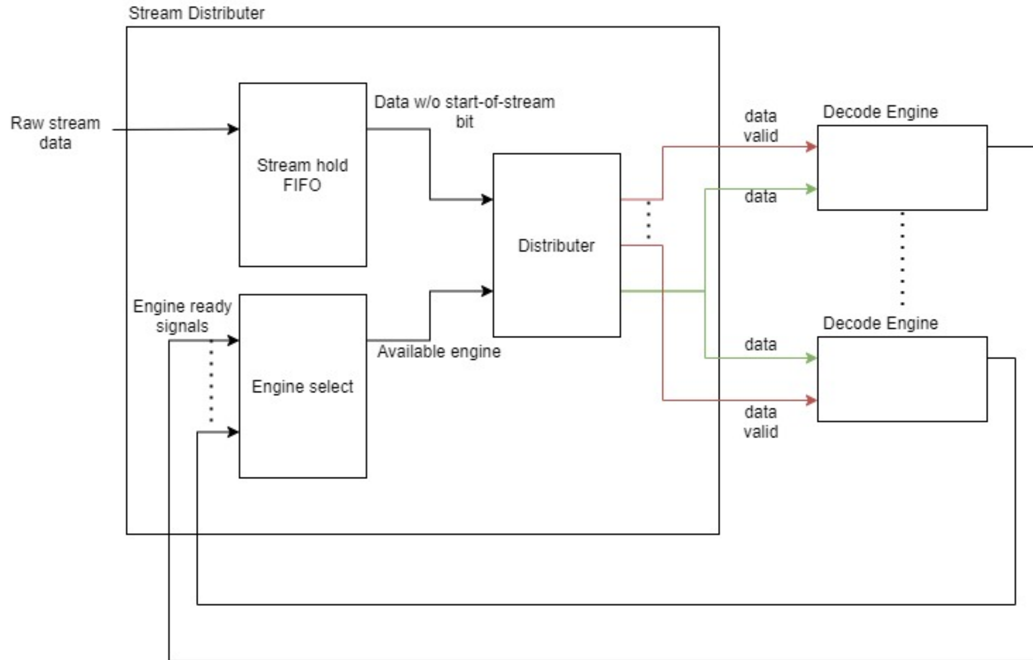

Figure 22: High level block diagram for stream distributer.

The stream distributer shown in figure 22 has an incoming data FIFO to hold the raw data. The engine select module will receive a ready signal from each engine. If an engine's ready signal is high, then it means that the engine receives a new stream of data. Once an engine is selected, the stream distributor continues to send data to the engine until a new stream is sent from the FIFO. At this point, the stream distributor selects the next engine with a ready signal and begins sending data to that engine. There is also an end-of-stream bit that goes out to each engine. This is to inform the engine when the distributor has found the end-of-stream and no more data is sent to the engine until it advertises that the engine is ready again.

After implementing the engine parallelization there was an issue with the ROMs resource usage. The size of the 16x32 ROM is 58 BRAMs, while the 14x16 ROM is 7.5 giving a total of 65.5 BRAMs per ROM set instantiation. Parallelizing the decode engines becomes costly if each engine has its own ROM system. To fix this issue, a centralized shared ROM between all decode engines was created.

3.5: Centralized ROM

With each decode engine receiving their own ROM system, the BRAMs will become the limiting factor in implementing this design. For each decode engine, the design will linearly increase at a rate of 65.5 BRAMs per decode engine. If we instead use a single centralized ROM shared amongst many engines, as shown in figure 23, the amount of decode engines will become limited to how many engines one ROM system can service reliably.

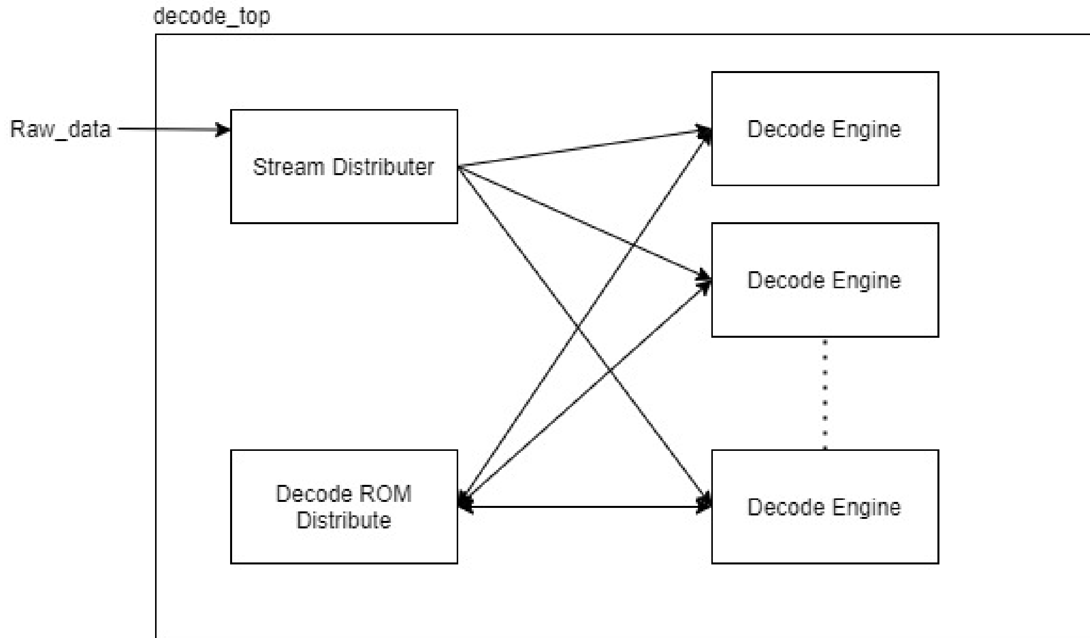

Figure 23: High level diagram of stream distributer, centralized ROM, and multiple decode engines.

To be able to service multiple decode engines with the same ROM set, the access requests need to be offset. Since the throughput time for a decode engine to process a quarter row is six cycles, each engine will send one hitmap decode request every six cycles. If each engines request is offset from each other, then there can be up to 6 decode engines being used by one ROM set. This would limit the system to having a maximum of 6 decode engines total, but by changing out the ROMs to dual port ROMs the system can have 12 engines. Dual port ROMs would allow for 2 engine accesses each clock cycle, thus allowing for 12 access requests every 6 cycles.

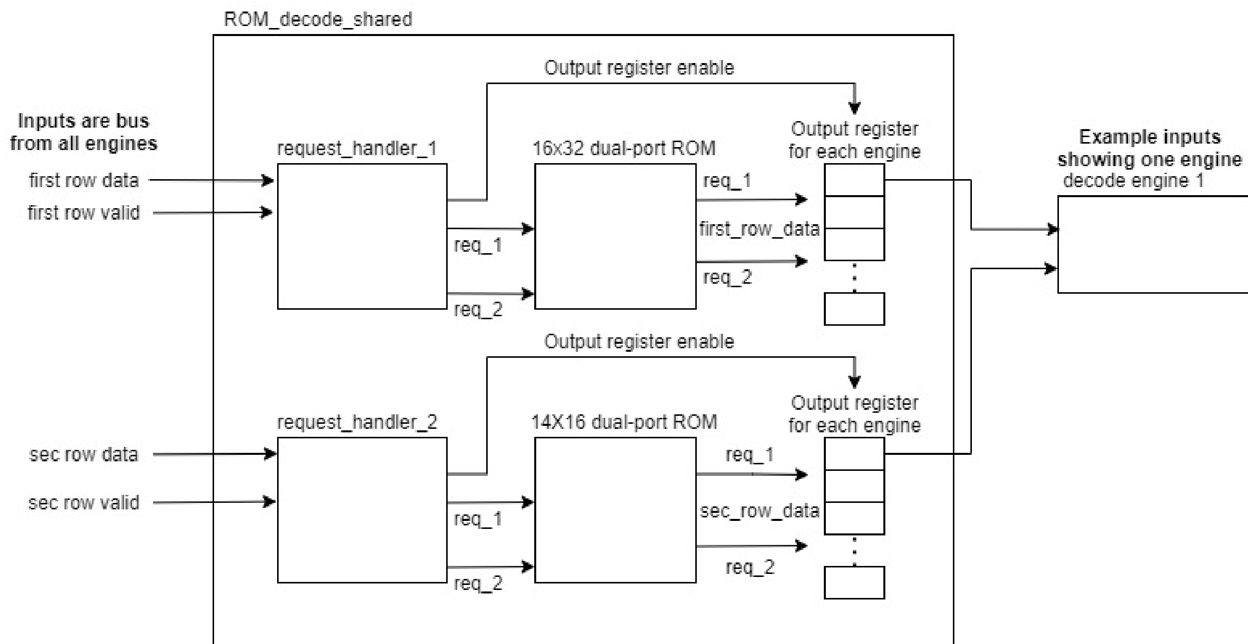

Figure 24: High level diagram of the ROM decode shared module.

To handle the shared ROM system, extra hardware was needed for the ROMs to handle request management as presented in figure 24. On the input side a request handler is used for each ROM to specify which engine is making a request. After the request goes through, there is a register bank on the output that saves each engine's decoded hitmap for that engine to use later. The request handler controls the inputs to the ROMs as well as the final registering. Once a decode engine sends a request, there is a two-cycle delay before the decoded output is ready to be used. The diagram below gives a more detailed view of how the requests and outputs correspond.

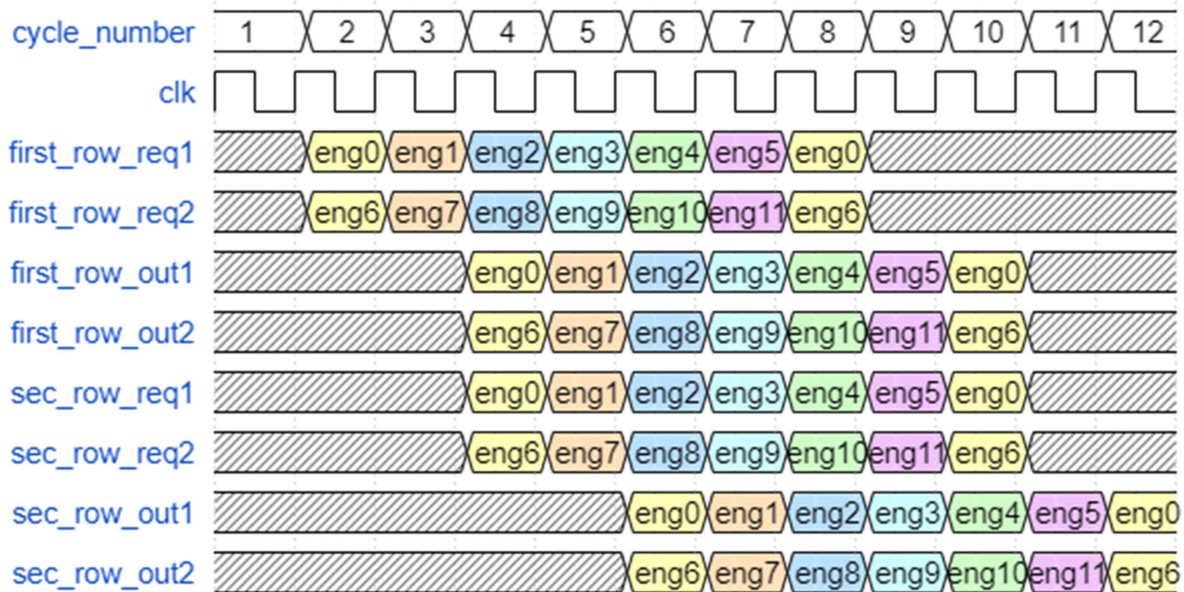

Figure 25: Timing diagram for the engine offsets for accessing centralized ROM system.

To explain this example further, we can trace the eng0 request through the system in figure 25. In cycle 2 eng0 makes the first request to decode row one. The ROM takes two cycles to process the data and on cycle 4, the decoded hitmap is ready to use. With this output, decode engine zero can calculate where the second-row data starts and send the data to the second row decode ROM for processing in the same cycle. After the second-row request is made in cycle 4, there is a two-cycle delay for the ROM to process the data and the decoded second row hitmap is ready for use on cycle 6. There is a one cycle delay in cycle 7 where the engine calculates the start of the next hitmap to send another request. On cycle 8, eng0 sends another request for hitmap decoding. This six-cycle pattern happens for each engine but offset to start at different cycles so that there is no overlap between engine requests.

3.6: Initial Full Design

With all the pieces of the design assembled, the first version of the hardware was completed. For testing and debugging, a data generator and test block were added to the design. In the beginning, a ROM was used for data generation with a few selected stream patterns to do basic testing. The test block is used to check the accuracy of the decode engines versus what the generation source produced.

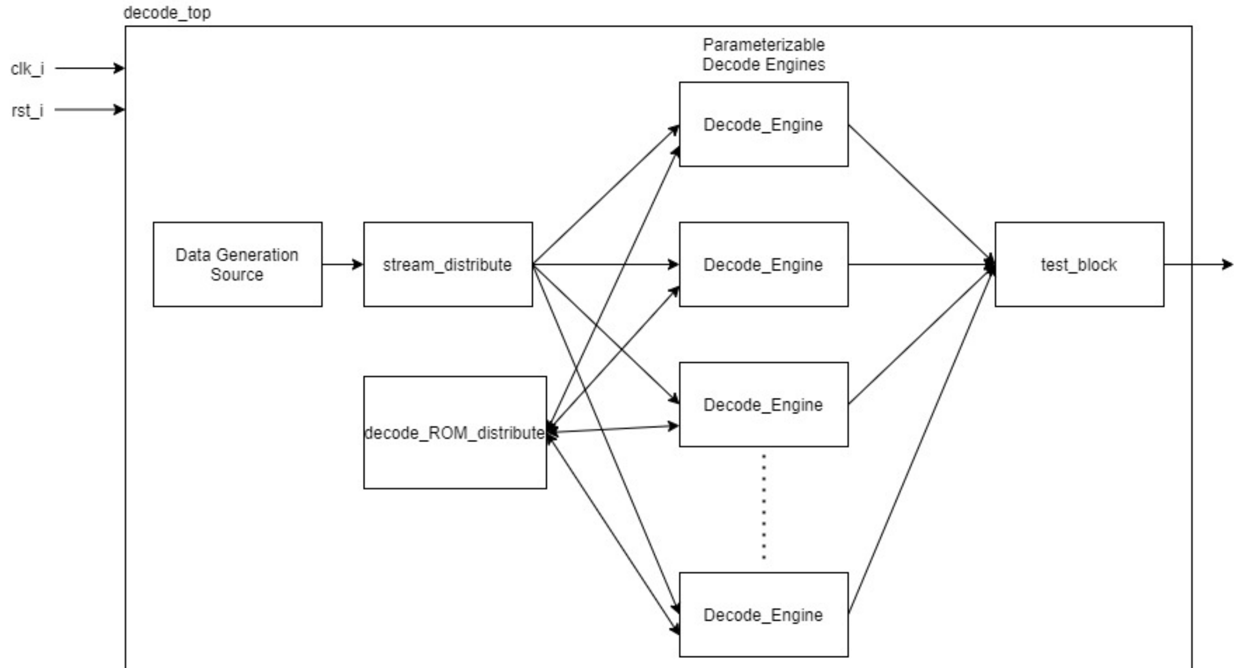

Figure 26: Top level diagram of initial full design for hardware decoder.

A larger testing system was developed for testing on an FPGA (see figure 27). At ACME labs, we have developed an RD53B emulator which includes a hit generator. Geoff Jones, Collin May, and An Nguyen have reworked the hit generator to be put back into the emulator project, which has been utilized for testing the RD53B decoder. The system consists of a hit maker that can generate streams of data both randomly and based off of simulated data. Encoded data streams are fed by the hitmaker into the hit decoder, and raw decoded data is sent from the hit maker to the hit checker block. The hit checker will then verify if the hit decoder is producing accurate data by checking against the decoded data sent by the hit maker. For now, the design only consists of the top half of the diagram and the errors can be checked through internal logic analyzers. The lower half of the diagram has not been created and checked yet, but consists of a CPU that is used to send configuration data to the system and report errors seen by the hit checker. This expanded testing system is used to test larger sets of data, while also having the capability to run for long periods of time.

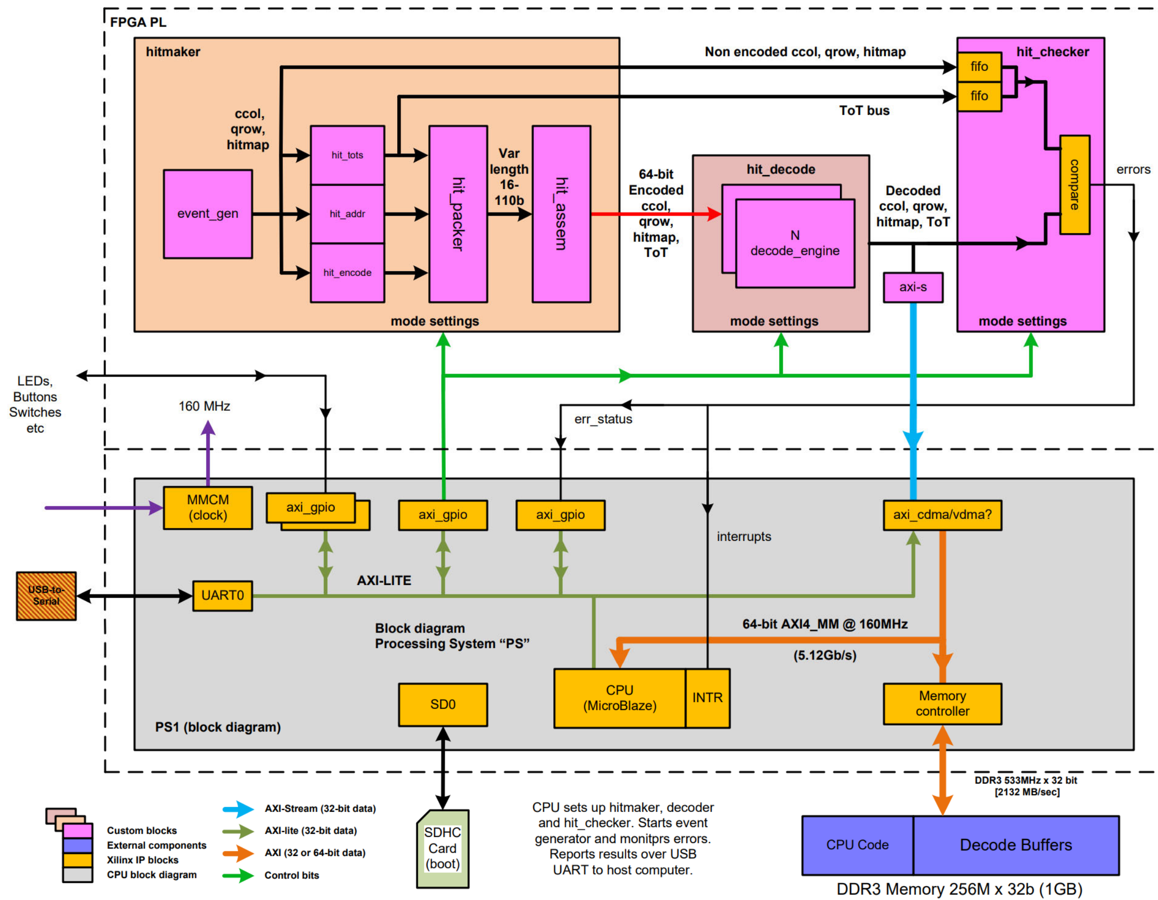

Figure 27: Top level architecture for hardware testing of the hardware decoder. Source Geoff Jones.

To report the utilization of the hardware decoding system, the setup in figure 26 is used since this configuration mainly contains the hardware decoder system. This design is slightly bloated due to the testing hardware but is a good estimation of the size. More precise resource numbers will be discussed in section 3.8 with the final design. The board that this design was compiled on is the Kintex-7 FPGA (KC705) board. Resource usage numbers shown in table 3 are specifically for a 12 decode engine configuration running at 160 MHz. Although a singular instance of the 12 decode engine configuration fits well onto a KC705 board with the hardware decoder alone, this design will likely be integrated into a larger project. To accommodate easier integration of the hardware decoder into other designs, multiple optimizations made to make the design more resource efficient while maintaining the current clock rate.

Table 3: Implementation resource usage for a 12 engine hardware decoder configuration.

Resource	Utilization	Available	Utilization %
LUT	66434	203800	32.60
FF	10112	407600	2.48
BRAM	140	445	31.46
IO	4	500	0.80

3.7: Optimizations

After design assembly, there was an effort to optimize the resource usage. A primary objective was to decrease overall BRAM usage because the ROM system is so large, and by replacing the 16x32 ROM with a 14x16 ROM the BRAM usage can be reduced by 50.5 BRAMs. The advantage of having the 16x32 ROM is that the 16x32 ROM can decode any 16 bit or less encoded hitmap on its own, but to create the shared ROM system, the decode engines need to have a standardized cycle pattern. Since we always take six cycles to decode a hitmap and always use cycles for both ROMs, this means there is no advantage to using the 16x32 ROM. To create a smaller resource footprint for the ROM system, the decode engine can decode the first two bits in logic gates to see which rows have hits in them, and then two 14x16 ROMs can be used to decode the top row and the bottom row individually. This reduces the BRAM count from 65.5 BRAMs per ROM system to 15 BRAMs per ROM system.

Once the BRAM reduction for the ROM system was in place, the main contributor to the remaining BRAM count was the FIFOs inside of the decode engines. The key to sizing these FIFOs is to create a FIFO big enough to hold one stream's worth of data on the ingress and egress of the decode engine to prevent FIFO overflows. To find the estimated size of a stream Hongtao Yang created the simulation data that allowed for a conservative estimate of event sizes. Since the assumption was made at the beginning of this project that there will be one event per stream, this estimated event sizes can be used as an estimate for stream sizes.

Figure 28: Conservative estimated event sizes for RD53B.

For the FIFO sizing, the value of 4244 bits was used which is the upper 3rd sigma quartile of the data from figure 28. In the first draft of the design, the input and output FIFOs were sized at 1024 entries as an estimate on what would be needed. Initial size of the input FIFO was 63 Kb (63 bits * 1024 depth = 63 Kb) at 2 BRAMs, and the output FIFO was 128 Kb (128 bits * 1024 depth = 128 Kb) at 4 BRAMs. By resizing the depth of these FIFOs to 512, the new sizes become 31.5 Kb at 1 BRAM, and 64 Kb at 2 BRAMs. This is still sized much higher than the recommended 4244 bits because reducing the FIFO depth any lower will not decrease the BRAM usage.

For the KC705 board that is being used, the size of each BRAM can be 18 Kb or 36 Kb where 36 Kb is equal to one BRAM. The shallowest FIFO configurations that use BRAM are 512 entries deep, anything less than that is wasted BRAM space which means input data width is now the limiting factor on BRAM reduction [6]. While the input size of the input FIFO to the decode engines can not be changed due to the packet size of the streams being fixed, the input size of the output FIFO to the decode engines can be changed since the output format is currently {26'b0, 8'tag, 6'ccol, 8'qrow, 16'decoded hitmap, 64'ToT}. In case meta data or byte alignment was needed, the upper 26 bits of the format was left as 0's, but as of right now they are not serving a purpose. By switching the output FIFOs data format to {8'tag, 2'b0, 6'ccol, 8'qrow, 16'decoded hitmap, 64'ToT}, all the data is still byte aligned and the output FIFOs would now be sized at 1.5 BRAMs each which would give a total savings of 6 BRAMs in a 12 decode engine system. Even more BRAM savings could be achieved if this new format were implemented, but the old format is currently still in the design. As mentioned previously, the output data format is not fixed and is subject to change.

The last optimization was for the LUT usage which consisted of optimizing the System Verilog code. The initial version of the design was built for functionality, so an effort was undertaken to use more resource efficient code to cut down on LUT usage. Redundant code was deleted and similar logic operations were combined in a more resource efficient manner.

3.8: Final Results

With the optimized design, the resource count for both BRAMs and LUTs significantly reduced. The LUTs and BRAMs both reduced by a factor of 2 for the 12 decode engine configuration while maintaining the same clock rate of 160 MHz.

Table 4: Resource usage based on implementation of resource optimized 12 decode engine configuration.

Resource	Utilization	Available	Utilization %
LUT	27132	203800	13.31
FF	9464	407600	2.32
BRAM	53.50	445	12.02
IO	4	500	0.80

By examining the hierarchical resource usage in table 5, the resource usage of a single decode engine is shown to average about 2100 LUTs and include 3 BRAMs for I/O purposes. In the 12 decode engine model, this contributes to 36 of the BRAMs in the overall design. If the decode engine output FIFO optimization stated in the optimizations section is considered, then the number of BRAMs from decode engines would be reduced to 30. The number of BRAMs from the optimized ROM system is 15, and the stream distributer

FIFO takes 2 BRAMs, so the real amount of BRAMs used by the decode engine is 53. Bloating from the test system makes up 0.5 BRAMs.

Table 5: Hierarchical breakdown of resource usage for 12 decode engine configuration.

Name	Slice LUTs (203800)	Block RAM Tile (445)	Bonded IOB (500)	IBUFDS (480)	BUFGCTRL (32)
decode_top	27132	53.5	4	1	1
decode_engines[0].u_decode_engine (decode_engine)	2224	3	0	0	0
decode_engines[1].u_decode_engine (decode_engine__parameterized0)	2090	3	0	0	0
decode_engines[2].u_decode_engine (decode_engine__parameterized1)	2092	3	0	0	0
decode_engines[3].u_decode_engine (decode_engine__parameterized2)	2146	3	0	0	0

The final throughput of the engine is a range based on the number of bits per quarter row as stated in the timing section. To get a more accurate picture of what the throughput would be, simulated ATLAS hit data was used to determine the average amount of hits for any given quarter-row that contains at least one hit. Quarter rows that do not include a hit during each event were excluded because empty quarter rows were excluded from data streams. Using a script for each to get data for different layers in the ITk pixel detector, it was determined that the average amount of hits per quarter row can vary between 1.78 and 2.34.

Table 6: Average amount of hits per active quarter row by layer. Layer 3 ATLAS hit data was not provided because layer 3 data is similar enough to data in layer 4 that the same estimates can be used for both.

Layer #	Average Hits Per Active Quarter Row
Layer 0	2.34
Layer 1	1.78
Layer 2	1.98
Layer 4	1.97

3.9: Interpretation of Results

For the sake of taking worst-case numbers, the throughput range will be for 1 hit per encoded hitmap. This would yield between 4 and 8 bits of encoded hitmap and 4 ToT bits per quarter row. The address data can range from 2 to 16 bits, which would give a final average number of bits per quarter row between 10 bits (2 address bits + 4 encoded hitmap bits + 4 ToT bits) and 28 bits (16 address bits + 8 encoded hitmap bits + 4 ToT bits). With a range of 10 to 28 bits per quarter row, the throughput for a single decode engine would range from 0.267 Gb/s to 0.747 Gb/s and the throughput for a 12 decode engine would range 3.2 Gb/s to 8.96 Gb/s.

The target that has been used up to this point is 5.12 Gb/s which is the maximum data output of an RD53B chip. Data rates will vary between different levels of the ITk pixel detector with layer 0 having more hits per event and layer 4 having less due to their relative proximities to the particle collisions. The reason there can be a steady assumption of 5.12 Gb/s across all layers is due to multi chip chaining which consists of aggregating data from multiple chips into one chip [5]. If a group of chips are running at 1/4th capacity, then the four chips can be chained together and aggregate their data into one chip which will now still be outputting at 5.12 Gb/s.

Since the target throughput was 5.12 Gb/s (the maximum data rate for one RD53B chip), the lower end of the 12-engine hardware decoder configuration throughput is roughly half the target throughput while the

higher end of the range has more than 1.5 times the throughput needs. With the current limit of 12 engines, to account for the worst-case throughput with maximum compression and 1 hit per quarter row, two hardware decoder instantiations with at least 20 engines to achieve a throughput of 5.2 Gb/s to keep up with the throughput needs one RD53B chip. As discussed in the future work section, multi pumping of the number of engines would be able to go up to 24 per instantiation. If this is implemented, then one hardware decoder instantiation would be able to handle this case with 20 engines. With the non-compressed address and largest hitmap encoding, an instantiation as low as 7 engines would have a throughput of 5.2 Gb/s and be able to handle the throughput needs of one RD53B chip.

Analyzing the more realistic case, the average number of hits comes closer to 2 hits per quarter row although the number of hits on average is slightly lower for layers like layer 1. Running on the assumption of an average of 2 hits per quarter row, the number of bits per quarter row now ranges from 15 (2 address bits + 5 encoded hitmap bits + 8 ToT bits) to 38 (16 address bits + 14 encoded hitmap bits + 8 ToT bits). With a range of 15 to 38 bits, the throughput per engine would range from 0.4 Gb/s to 1.01 Gb/s and for a 12-engine configuration would range from 4.8 Gb/s to 12.16 Gb/s.

Looking at the 2 hit per quarter row throughput, the lower end of the 12-engine throughput range is slightly below the 5.12 Gb/s target while the higher end of the range more than doubles the throughput needs. To meet the full needs with the maximum address and hitmap compression, there would need to be 13 total engines which gives a throughput of 5.2 Gb/s. Like the discussion of a 1 hit quarter row, this could be done in two instantiations of the hardware decoder or with multi pumping in future work. If non compressed address and encoded hitmap bits is used, then the throughput of one RD53B chip could be handled with 6 engines giving a throughput of 6.06 Gb/s.

With the number of engines known, estimates can be made for the number of resources needed for each configuration. Each instantiation of a hardware decoder takes 15 BRAMs for the split ROM system, 2 BRAMs for the stream distributor module, and 3 BRAMs per decode engine. The LUT count is heavily dependent on the amount of decode engines and an estimate of roughly 2100 LUTs per decode engine is a good estimate. The LUT usage for the rest of the system is minimal in comparison, so for these estimates they will be ignored. The final estimates for each configuration will be given in table 7, but as an example of how the resources are calculated the resources for the 20-engine configuration is as follows. For 20 engines, there will have to be two instantiations of the hardware decoder, so the BRAM usage is 15 * 2 + 2 * 2 = 34 BRAMs for the non-decode engine portion of the design. The decoder engines will contribute 20 * 3 = 60 BRAMs and 2100 * 20 = 42000 LUTs for the whole design. Overall, this configuration has an estimated 94 BRAMs and 42000 LUTs.

To know what kind of instantiation would be optimal, more research would need to be done into the breakdown of the encoding bit ratios and the address bit compression ratios. This may depend on the use case of the hardware decoder as well. The numbers presented are for very generic cases of how many hardware encoders are needed to support one RD53B chip. A summary of the discussion of this section is presented in table 7.

Table 7: The summary of how many engines would be needed for different numbers of bits per quarter row with estimated resource usages. For any configuration that has over 12 engines, the resource estimation is for two hardware decoder instantiations.

Hits per quarter row w/ compression	Bits per quarter row	Single engine throughput (Gb/s)	12 engine throughput (Gb/s)	Engines needed for one RD53B	Estimated BRAM usage	Estimated LUT usage
1 max comp	10	0.267	3.2	20	94	42,000
2 max comp	15	0.4	4.8	13	73	27,300
1 min comp	28	0.747	8.96	7	38	14,700
2 min comp	38	1.01	12.16	6	35	12,600

The code for the hardware decoder can be found in the following link in the decoder_plus_emulator branch: https://gitlab.com/scotthauck/largehadroncollider

4: Conclusion

The HL-LHC upgrade will increase particle collision rates which has led to the overhaul of the ATLAS pixel detector to become the new ITk pixel detector. In the process of upgrading to the ITk pixel detector, the read-out chips that are installed inside of the pixel detector needed to be upgraded as well. This led to the collaboration of CMS and ATLAS to create the new read out chips named RD53. In the second iteration of these readout chips, RD53B, an encoding process was created for the hitmaps to cut down on the amount of data that had to be read out of the RD53B chips. With an encoding process in place, the hardware decoder project started to find a fast, efficient, and low resource way to decode the data from the RD53B chips. This thesis covered the process that was taken to create a base version of the hardware decoder.

Initially, a ROM decoding system was created to decode the encoded hitmaps sent by the RD53B. The ROM decode system was the basis of the design and allowed for quick decoding of hitmaps. With the ROM system in place, a decode engine was built around the ROM system. The decode engine was built to parse through the data streams and package the decoded data into a format that could be used by downstream modules.

When the initial decode engine was built, a timing analysis was done that led to the conclusion that a single decode engine would not have the throughput capacity to service the 5.12 Gb/s maximum data output of a single RD53B chip. A stream distribution system was created to allow for multiple decode engines to service data streams in parallel. With multiple decode engines, having one ROM decode system per engine became too costly in terms of BRAM resources. The ROM decoding system became centralized to save on BRAM resources and the decode engines would now access the centralized ROM on offsetting clock cycles.

With the centralized ROM system in place, the design was debugged and tested until the decode engines covered all base functionality. Once the base functionality was achieved, the design was optimized to further reduce both BRAM and LUT resources by a factor of 2. With the optimizations in place, ATLAS simulation data was used to create a base resource estimation for multiple configurations and use cases of the hardware decoder. The different use cases presented from the simulation data showed a maximum resource usage of 94 BRAMs and 42,000 LUTs and a minimum resource usage of 35

BRAMs and 12,600 LUTs to meet the maximum throughput needs of 5.12 Gb/s of the RD53B chip. These estimations serve as a way for potential users of the hardware decoder to see how this design would fit into their system.

5: Future Work

The current implementation of the RD53B hardware decoder is meant to be a base model. The initial groundwork has been laid, but there are still many improvements to be made.

Firstly, recovery features for the hardware decoder to deal with bad input data will be useful for real implementation of the design. Basic debugging has been done for the hardware decoder with a reliable input data set, but features have not been implemented to account for misaligned packets or having bits flip within a packet that may cause the engine to get stuck. The ability to recover from bad data and not stall a decode engine will be a key feature for this project moving forward. Features like a timeout system if the engine gets into a bad state or stream truncation system for overly large streams should be investigated for better reliability.

Next is to include more modes of operation for the decode engine. Currently, the only supported RD53B operating mode is single chip and ToT on. The RD53B can support modes where multiple different chips have their data chained together and then identified with a chip ID in the stream packets which consists of two bits. Implementing this design would involve expanding the stream distributer module to process the chip IDs and send specific chip data to specific engines. This change will increase the throughput of the hardware decoder because the two bits of every packet with chip ID data will be processed by the stream distributor.

The ToT off mode will drop the ToT data from the stream completely. Implement this mode would involve a minor change to be made inside of the decode engine to not register ToT data or move the index pointer for ToT data. Not including ToT data will decrease the throughput of the hardware decoder because there will not be less bits processed every six cycles.

In terms of optimizations, there is still a few methods that can be employed to decrease the resource usage. One is multi-pumping the decode ROMs and the other is c-slowing the decode engines. Multi-pumping would entail running the decode ROMs at double the speed to allow them to service twice as many engines allowing for up to a 24 decode engine configuration. This would save on the initial cost of 15 BRAMs per decode ROM system instantiation by doubling the maximum amount of decode engines per hardware decoder instantiation. Currently, the hardware decoder is targeting 5.12Gb/s because that is the max speed for a single chip, but with 24 engines, multiple chips could be combined and used by the same hardware decoder.

C-slowing is an idea to reduce the decode engine LUT usage by turning the decode engine into a pipelined processor. Each state from the DCSM could be working on a different data stream allowing for up to 6 data streams to share the same LUT resources. A factor of 6X LUT decrease could be achieved through c-slowing the design but would not allow for less than 6 decode engines per hardware decoder instantiation.

Now that there is a base design for the hardware decoder, a good way to benchmark the hardware decoders performance would be to compare it to the current software decoding implementation. This could be done by comparing the time it takes for both implementations to decode a quarter row, a stream, or multiple streams. As the hardware decoder project moves forward, being able to compare the hardware decoder against other decoder implementations would be valuable data in helping to find the best decoding method.

6: Acknowledgements

From the start, I want to extend a general thank you to everyone who I have worked with during the Covid-19 pandemic. These times were rough for everyone, but I am thankful for the helpful friendly faces.

I would like to thank Scott Hauck and Shih-Chieh Hsu for their help and mentorship through this process. Scott was always there to help guide me down the right path without giving me too many answers. Shih-Chieh was always there to guide me with physics knowledge that was beyond me and to keep me driving towards a goal. I remember the first time I heard about ACME labs, I was on a trip in Portland when I got an email from Scott about joining the lab. Scott was my professor the previous quarter where I first learned about FPGA design. FPGAs absolutely fascinated me, but I was sad at the fact that I only got introduced to digital logic in my Junior year of college. This opportunity meant the world to me and even though there were ups and downs, I could not have been happier that I accepted this research position.

Timon Heim has been a huge driver in the hardware decoder project. He was the one who originally reignited the idea after the first attempt seemed to be a bust. I know that he is a very busy man, but he always takes time out of his schedule to answer any questions and I can't thank you enough for that.

Hongtao Yang's work is what helped to kick start this whole project. Without his initial code and guidance, I would probably still be debugging a ROM system for the hardware decoder. Thank you for taking the time to work with me on this project.

I would like to thank Geoff Jones, Collin May, and An Nguyen for their help in creating the hit maker design to help test the decoder. Without their help, this project would have taken another 6 months to get to the point it is at. Geoff has been there to help me with any questions, bounce ideas off, and is generally a nice person to talk ideas through with. Collin and An, I know I have not known you very long, but your work is very much appreciated, and I wish you the best luck.

Finally, to friends and family, I must thank you for all the love and support. To Agnes Song, my girlfriend of four and a half years, thank you for putting up with me during this hectic period and I will finally have time to take you on dates again. You have always been there through thick and thin and I cannot wait to see what the future holds for us. Thank you to my mom Sue Maciuszek, brother Dylan Erickson, and sister Alyssa Erickson, for pushing me and supporting me every step of the way. To all of my friends, thank you for helping me to remember to kick back and relax every once in a while.

7: References

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

[2]. "High Luminosity LHC Project", CERN, [Online], Available: https://hilumilhc.web.cern.ch/

[3]. Chistiansen, Jorgen, Loddo, Flavio, "RD53 Collaboration Proposal: Extension of RD53", RD53, September 6, 2018, [Online], Available: https://cds.cern.ch/record/2637453/files/LHCC-SR-008.pdf

[4]. "The RD53B Pixel Readout Chip Manual v1.31", RD53, September, 2020

[5]. "The RD53B Pixel Readout Chip Manual v0.38", RD53, April 14, 2020

[6]. "7 Series FPGAs Memory Resources v1.14", Xilinx, July 3, 2019, [Online], Available: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[7]. Timon Heim, private communication, 2021

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

[9]. "CERN atlas website", CERN, [Online], Available: https://atlas.cern/discover/detector