University of Toronto
Faculty of Applied Science and Engineering
ECE521
Final Report
March 22nd, 2018

Presented by:	Team	2017737
	Mitchell Kingsley	1000393095
	Ruben Tjhie	1000374757
Presented to:	Khoman Phang	
	Natalie Enright Jerger	
Section:	4	

Table of Contents

1.0 Introduction

This report summarizes the motivation, design, implementation and testing of a system that uses approximate computing to perform error correction in video streams. This report is a component of our final year design project course ECE496. The report concludes with suggestions of improvements and future work to the aforementioned system.

1.1 Background and Motivation (Author: Mitchell)

Today, many options for wirelessly transmitting video streams have emerged in consumer and commercial markets. Protocols such as Miracast, CASTV2 (Google's Chromecast), and AirPlay (Apple's AppleTV) allow users to transmit high definition video over Wi-Fi, as protocols such as WirelessHD and WiGig use a 60Ghz carrier signal to provide wireless HD video between transmitter and receiver. All of these listed options are viable solutions for wireless HD video transmission, but poor system performance (defined by reliability, latency, and power efficiency) become an issue in wireless applications that require low latency.

This project aims to fix such problems by means of creating a system that is able to process incoming video streams and apply an error correcting algorithm to said streams. In order to minimize the latency and power used within the system, approximate computing methods will be used.

Approximate computing describes a method of design within computing that sacrifices the accuracy of the result in order to improve other metrics in the design. This method of computing is not ideal for programs and designs where the accuracy of computed values is paramount, but can be beneficial in designs where a small subjective loss of quality may not be noticed or important. In the case of the aforementioned error correction algorithm, we are trading the accuracy of pixel colour values within the image for a reduction in the power consumed and a reduction in latency of a wireless video stream with erroneous data. Approximate computing is appropriate for this application due to qualitative nature of video quality and the lack of ability in human vision to perceive minute differences in colour over extremely small intervals of time.

The structure of the design is comprised of two main elements: A software component to create a test video stream and inject error into it and a hardware component that takes in the erroneous video stream and applies an error correction algorithm on it. The test video stream software is run on a PC, and the hardware component is stimulated via the Questa's ModelSim.

Over the course of project development since the initial proposal, the project goal has remained largely the same. The intention was to create an error-tolerant video processing system that is appropriate for real-time video applications using unreliable communication channels. The project requirements evolved as we began to make decisions about our solution. From our research of wireless transmission protocols we argue that error is introduced as packet errors according to a packet error rate. Therefore, our system was designed so it could receive these packets and identify erroneous or missing packets to rebuild the video frame. Packets can vary by two parameters, packet size and packet error rate. To exercise the limits of our design we will vary these parameters and measure the results. This will demonstrate the error-tolerance of our system and the effect of smaller or larger packets.

1.2 Project Goals and Requirements

The following sections describe the primary goal of the project and the project requirements which are used to quantify the success of the project.

1.2.1 Project Goal (Author: Ruben)

The goal of the project is to develop an error-tolerant video processing system that is appropriate for real-time video applications using unreliable communication channels. This will be accomplished by implementing approximate computing methods that approximates the representation of frames received in error.

1.2.2 Project Requirements (Author: Ruben)

The following table (Table 1) outlines the updated functional requirements, constraints, and objectives of the project.

Table 1: Functional requirements, constraints, and objectives.

Functions	
1.0	The design shall generate video frames from packets containing pixel data
2.0	The design shall interpolate erroneous or missing pixel data to build video frames
3.0	The design shall make use approximate computational circuits (adders, multipliers) when performing interpolation algorithm
4.0	The design shall process video frames into packets of varying size
5.0	The design shall introduce packet errors into a set of packets belonging to a video frame at a varying packet error rate
6.0	The design shall be able to process video packets with varying packet error rates
Objectives	
7.0	The pixel colour approximated by the algorithms should have a CIELAB Delta E* value under 10 when compared to the correct pixel colour [1]
8.0	Power usage by the implementation should be kept to a minimum
9.0	The resolution and frame rate of the output video should match the input video
Constraints	
10.0	Computational time of the algorithm shall not exceed 20 ms, the minimum time in which input lag becomes perceptible to the user [2]

11.0	Approximations performed to generate output video shall not result in loss of subjective video quality by users
12.0	Interpolation and approximate computational circuits shall be implemented on FPGA on DE1 board

2.0 Final Design

This section describes the technical components of the system in its entirety. It provides a high level overview of the system, module level descriptions, and an assessment of the final design.

2.0.1 Terminology (Author: Mitchell)

The following table (Table 2) defines the vocabulary used within the Final Design portion of the document.

Table 2: Definitions of Vocabulary used in Final Design Portion of Report

Term	**Definition**
Frame Buffer	A buffer that contains a single frame of video data.
Current Frame Buffer	The frame buffer that contains the frame of video data that is undergoing correction.
Previous Frame Buffer	The frame buffer that contains the frame of video data that was previously processed.
Target Pixel	The pixel within the current frame buffer that is undergoing correction.
Surrounding Pixels	The pixels that surround the target pixel.
Target Packet	The packet within the current frame buffer that is undergoing correction.
Target Line	The line within the current frame buffer that is undergoing correction.

2.1 System Level Overview

The system consists of two major components:

1. Video pre-processing system that translates video into frames which are injected with error
2. Pixel estimation hardware which calculates the color value for a target pixel in an erroneous frame.

These two components work in series with one another, where the pixel estimation hardware performs error correction on the errors injected into the current frame buffer by the video pre-processing system. The configuration for the system is as shown below in Figure 1.

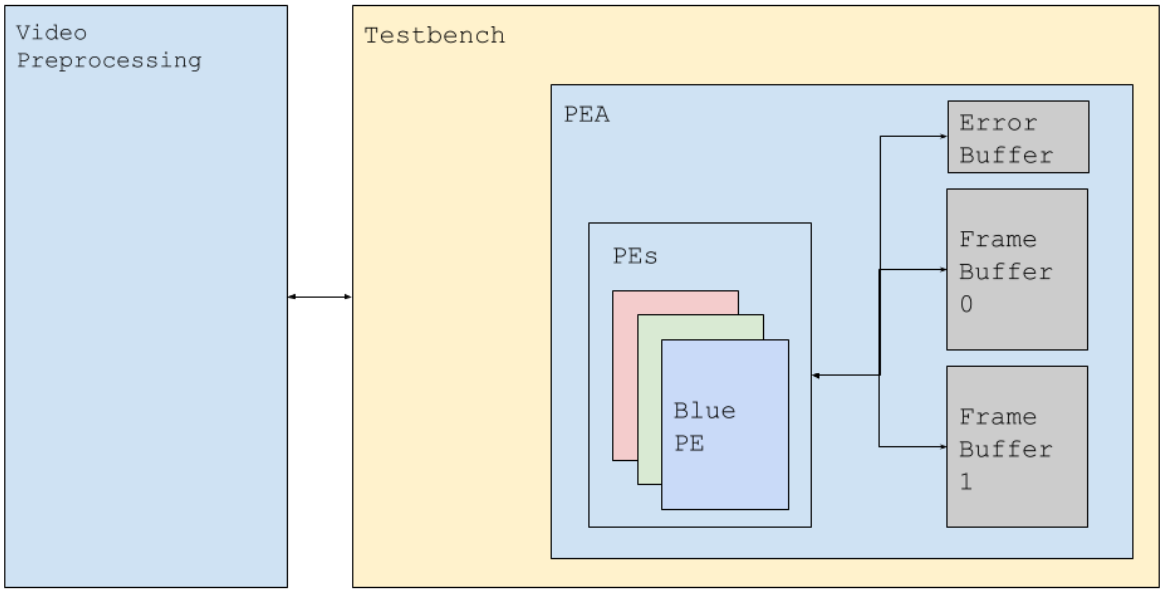


Figure 1: A high level overview of video pre-processing and correction system.

2.2 Pixel Estimation Algorithm (Author: Mitchell)

The pixel estimation algorithm (PEA) is the main component within our design. The purpose of the PEA is to take information from a erroneous video source and calculate colour values for the portions of the video that are in error. The PEA consists of three pixel estimators (or PEs) for each color channel, buffers for storing frame data, and an error buffer. Within each PE there is a control FSM and approximate adders used for colour calculation. An overview of the PEA is shown in Figure 2.

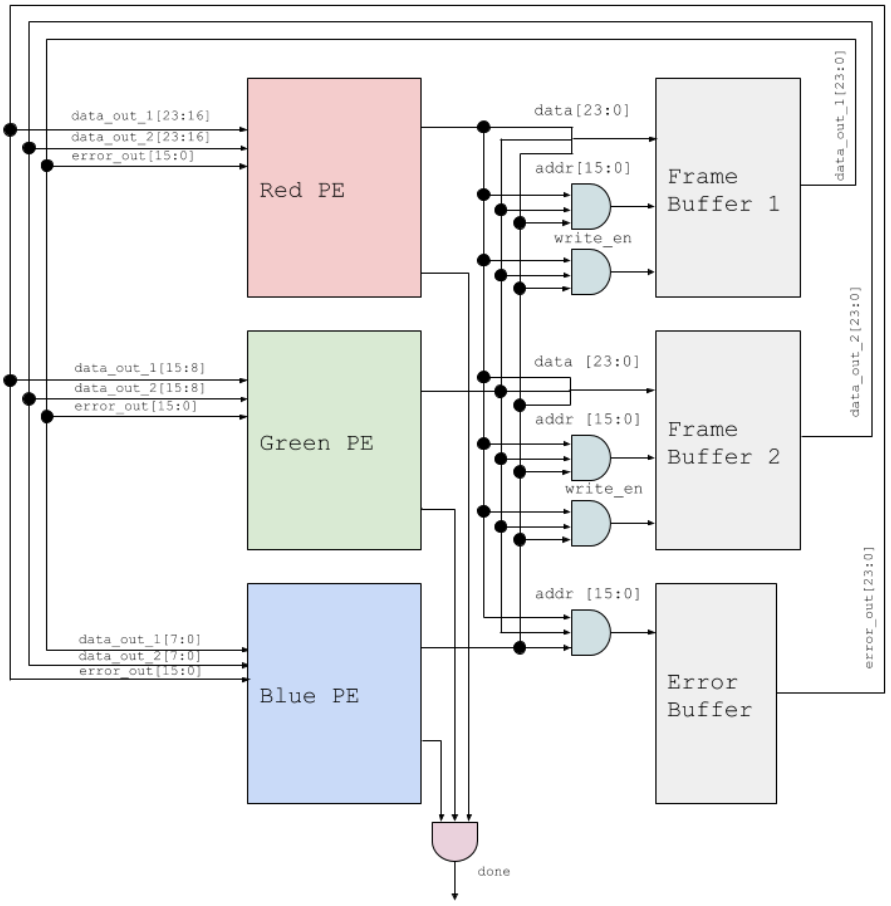


Figure 2: Top level view of the PEA. Each line exiting the red, green, and blue PE contains data[23:0], addr[15:0], and write_en. These signals were consolidated for visual clarity.

2.2.1 Control FSM (Author: Mitchell)

The control FSM is the brain of each PE. It controls the frame buffers to read and write from, keeps track of the target pixel and surrounding pixels, and initiates the calculations of the approximate result of erroneous packets.

The control flow for the FSM can be broken down into two main components: The pixel tracking logic and the pixel loading logic. The pixel tracking logic is a series of counters that keeps track of the target pixel, target packet, target line, and frame that we are in as well as determine when error correction needs to occur. The pixel loading logic is responsible for collecting the relevant pixel data from the current or previous frame. This data is subsequently used to calculate the estimated color value for the target pixel. A state diagram for the pixel tracking logic can be seen in Figure 3. A state diagram for the pixel loading logic can be seen in Figure 4. Descriptions for each state can be found in Table 3.

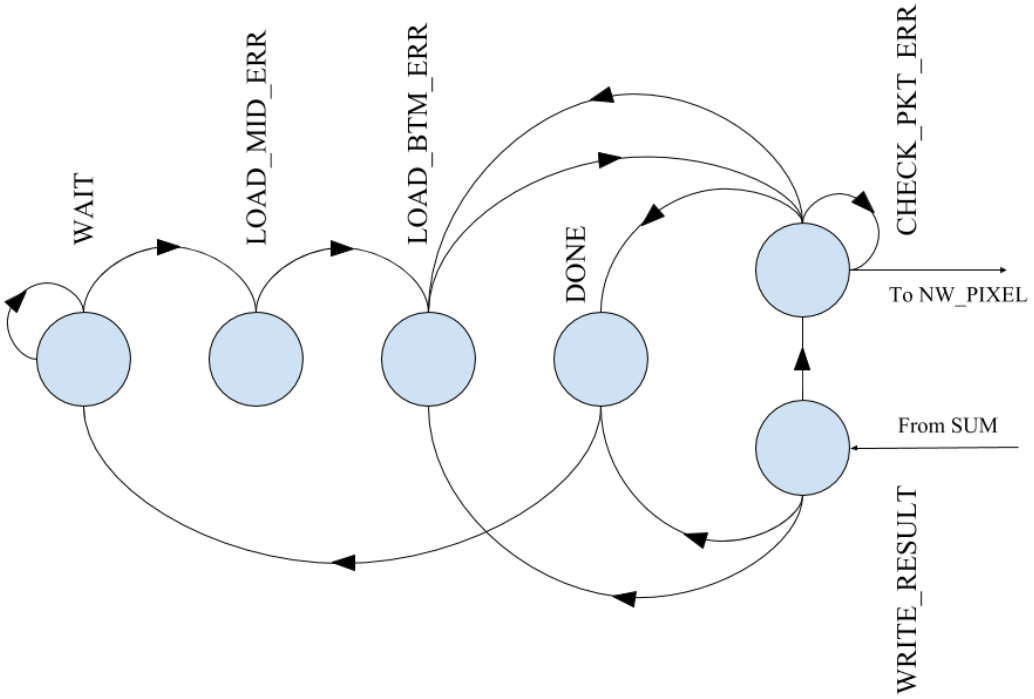


Figure 3: FSM for Pixel Tracking Logic

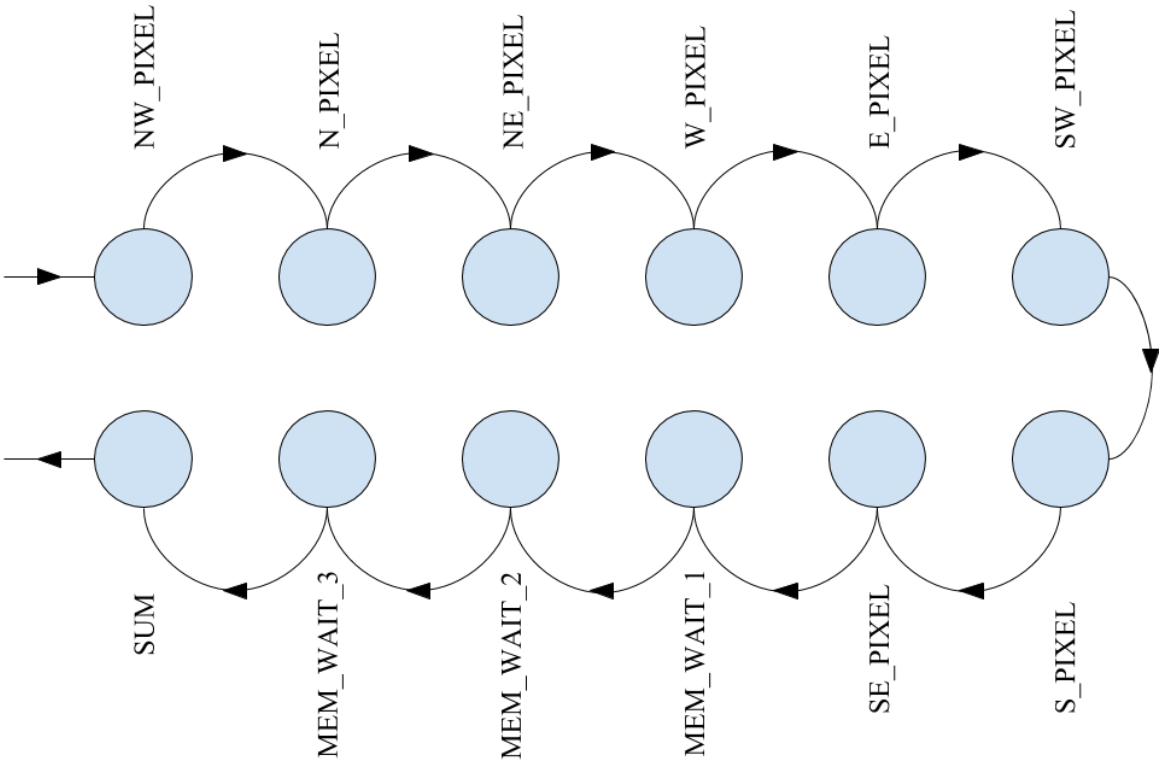


Figure 3: FSM for Pixel Loading Logic

Table 3: Description of States in Control FSM

State	**Description**
WAIT	Wait for the start_signal to go high. Initialize values used in estimation.
LOAD_MID_ERR	Load packet error information from memory into the error buffer for the current line.

LOAD_BTM_ERR	Load packet error information from memory into the error buffer for the line below current. If not on initial iteration, shift error buffers such that:
	error_buf_top <= error_buf_mid;
	error_buf_mid <= error_buf_btm;
	error_buf_btm <= error_mem_read;

CHECK_PKT_ERR	Check if the current packet is in error. If it is, begin correction process. If not, then jump to the next packet. If no erroneous packets left in current frame go to DONE state.
(*)_PIXEL	For pixel (*) (where (*) is the pixel to the NE, N, NW, W, E, SW, S, or SE of the target pixel), fetch the color value from the appropriate frame buffer. If pixel (*) is also in error, then fetch from the previous frame buffer, otherwise fetch from the current frame buffer. Read memory from previous pixel memory accesses
MEM_WAIT_(1/2/3)	Delay for 3 states while we wait for the last reads from the S and SE pixel values to become valid.
SUM	Sum and divide the pixel values fetched from the *_PIXEL states. Depending on the target pixel location, sum all pixels or a subset of each.
WRITE_RESULT	Write the result of sum to the target pixel location in the frame buffer.
DONE	If all erroneous pixels in the current frame buffer have been corrected, indicate that correction is complete.

Simulation results for the PEA can be seen in Appendix G, Appendix H for an RTL view of the PEA top level, and for a full description of the next state logic see Appendix I.

2.2.2 Approximate Adders (Author: Mitchell)

The approximate adders are located within the PEs and are responsible for computing the estimated value for the target pixel. Since the target pixel is calculated by taking the average of the 8 surrounding pixels, the superstructure of the adders must be able to sum 8 pixel values that are each 8-bits wide. To achieve this a series of approximate adders are aligned in a 3 layer chain, with the first layer taking in 8-bit values, the second 9-bit, and the third a 10-bit. The superstructure is shown in Figure 4.

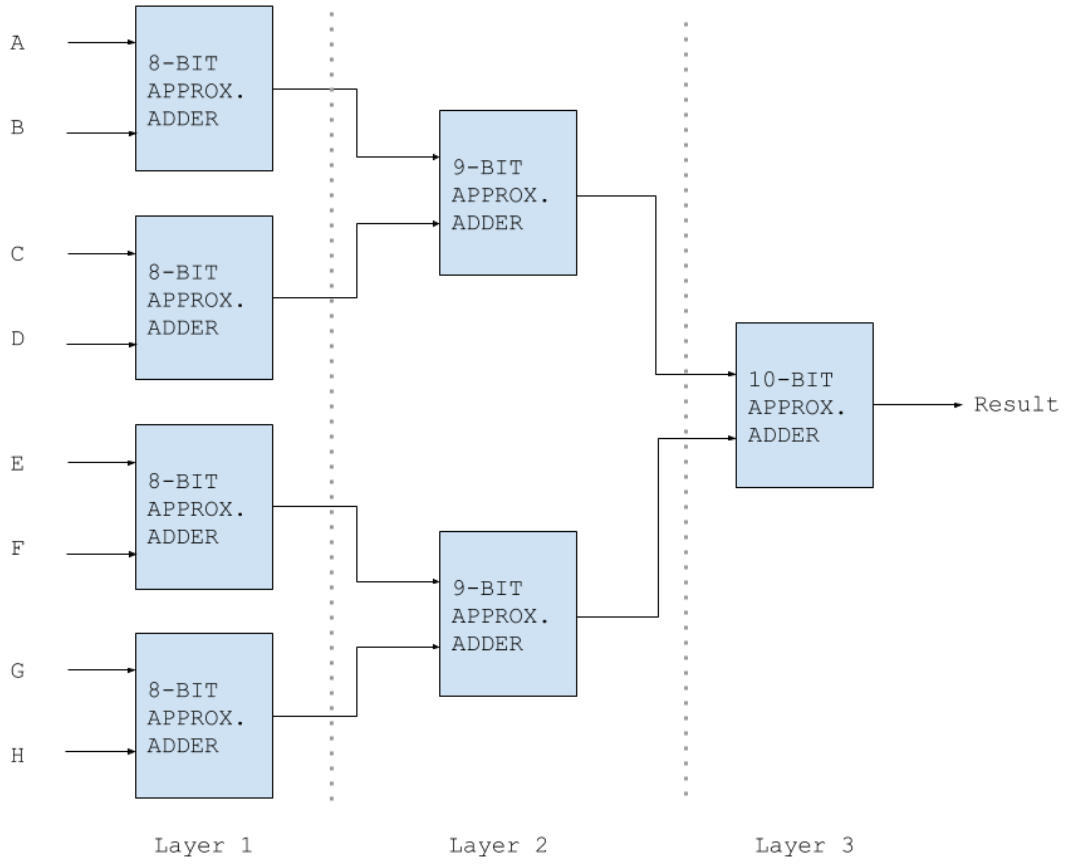


Figure 4: Approximate adder superstructure.

Each layer holds an adder that is one bit larger than the previous layer to accommodate for the cases where the sum of the previous layer exceeds the width of the previous layer's inputs.

The approximate adders themselves are designed in a way which minimizes the length of the carry chain through each adder, resulting in a structure that produces sums faster and uses less power [3]. When two values are passed into the adder to sum, the bits are immediately divided between three 4-bit full adders. Bits [3:0], [5:2], and [7:4] are each sent to their own 4-bit sub-adders and the sum from each sub-adder is calculated normally. Selections of each sub-adder's result is then taken in order to compile the final result of the approximate adder. The structure of the 8-bit approximate adder and the order of sub-adder results that compose the final result is shown in Figure 5. An RTL view of the 8-bit approximate adder and the adder superstructure can be seen in Appendix J.

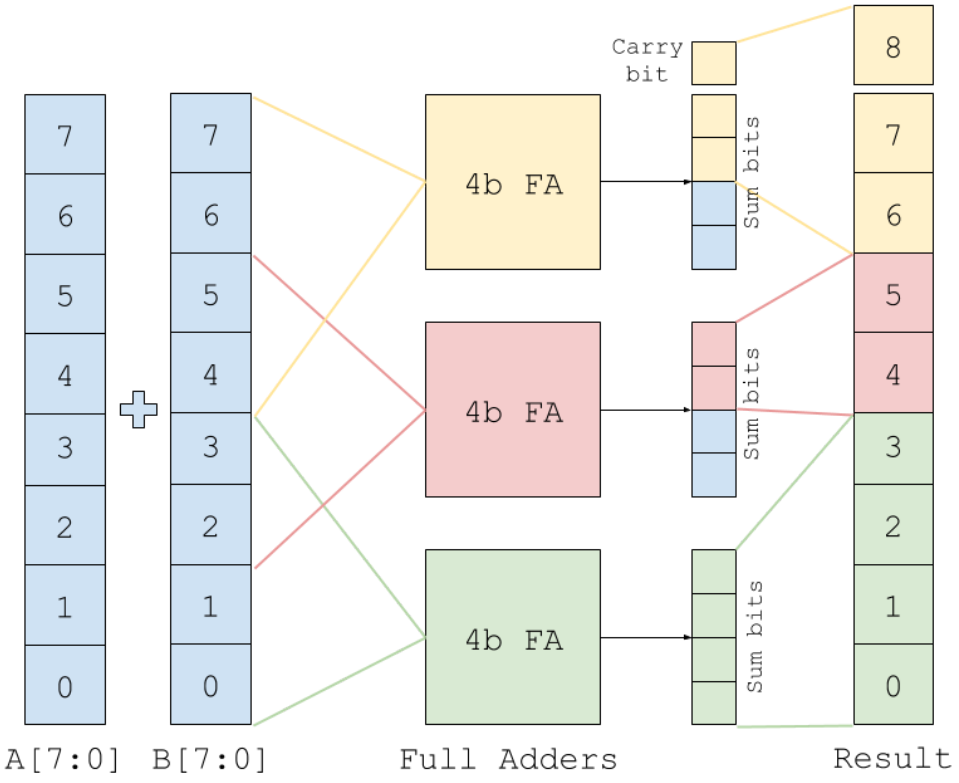


Figure 5: Structure of 8-bit approximate adder. Notice on the yellow and red 4-bit adders the 2 most LSB are dropped when compiling the final sum.

2.3 Video Preprocessing and Error Generation (Author: Ruben)

Before the PEA can approximate an image, error must first be injected into it. The error generation program takes in a 24-bit color depth RGB video stream and injects errors into the stream. These errors are in the form of discrete packets which can be configured to be different lengths and frequencies. The injections emulate errors that would be encountered in the wireless transmission of video signals.

In order to accomplish this the following python modules were developed:

1. vid_cap.py
2. gen_frame_buffer_mif.py
3. gen_error_table_mif.py
4. gen_jpg.py
5. comparison.py

Key features of each module are discussed in the following sections. The flow of files through the python modules are shown in Figure 6.

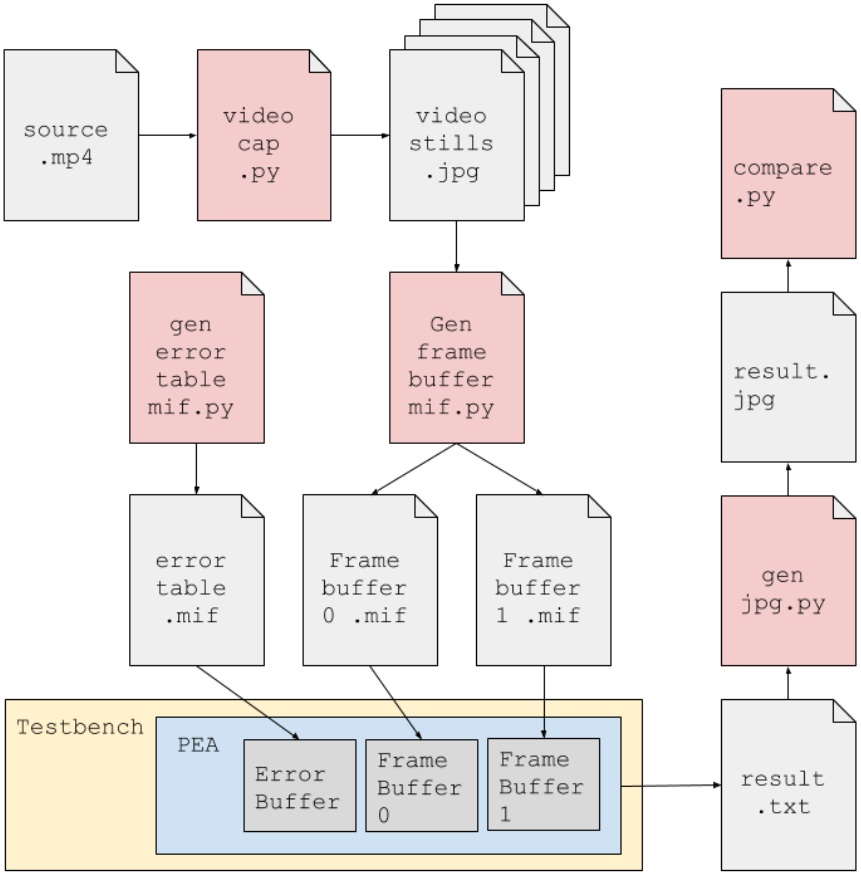


Figure 6: The flow of a source video through the developed python modules.

2.3.1 Video Capture (Author: Ruben)

The first step in the system is to generate individual JPEG frames from video files. The `vid_cap.py` module uses the opencv2 library function `cv2.VideoCapture()` to read individual frames, then uses `cv2.resize()` to resize the frames into the resolution required. The frames are then written as JPEG images using `cv2.imwrite()`.

2.3.2 Initialize Frame Buffers (Author: Ruben)

In order to initialize frame buffers in the design with the individual frames, JPEG images must be translated into MIF format using the `gen_frame_buffer_mif.py` module. Again, using the opencv2 library, JPEG images are read in with `cv2.imread()` and parsed into 24-bit words. These words are written into MIF files which have a specific format that indicates what data exists at each address.

2.3.3 Initialize Error Table (Author: Ruben)

The design makes use of an error table to determine which packets were received in error or never received. In order to initialize this table a MIF is required - similar to the frame buffers. The key feature in the `gen_error_table_mif.py` module is the use of

`numpy.random.binomial()` to generate the distribution of packet errors according to a specific packet error rate (PER). With this distribution 16-bit words are generated and written into MIF format.

2.3.4 Generate JPEG from Testbench Output (Author: Ruben)

The Verilog testbench outputs a text file that contains the error-corrected frame, this text file is input into the `gen_jpg.py` module which uses the opencv2 library to process the text file into a JPEG.

2.4.5 Compare Output with Original Frame (Author: Ruben)

The generated output JPEGs are fed through `comparison.py` to get the resulting colour differences.

2.5 Verilog Verification Environment (Author: Ruben)

The Verilog testbench instantiates the top-level module connecting memory modules to each R, G, and B channel PE instantiations.

The testbench has three phases:

1. **Initialize:** reset entire system, and initialize registers in DUT with appropriate packet size
2. **Process:** sends start pulse to algorithm to direct it to begin processing data in buffers
3. **Read:** when algorithm asserts done signal read what was written into frame buffer and output into text file

The DUT includes three sets of interfaces to the testbench:

1. **System interface:** clock, reset
2. **Initialization:** packet_size, and start
3. **Output:** done, buf_sel
   Tells testbench that the algorithm has finished
4. **Verification:** verif_sel, read_addr, read_data
   Allows testbench to access frame buffers to read results

This testbench is critical in verifying the functional correctness of our system. Resulting simulations can be seen in Appendix G.

3.0 Testing and Verification (Author: Mitchell, Ruben)

This section describes the requirements for the design as outlined in Table 1, as well as the methodology used to test them.

Table 4: Verification Matrix

Change	ID	Requirement	Verification Result and Proof		Requirement Verification Method		
				Similarity	Review	Analysis	Test
	1	The design shall generate video frames from packets	Pass. See Section 3.1.1				X

		containing pixel data					
	2	The design shall interpolate erroneous or missing pixel data to build video frames	Pass. See Section 3.1.1				X
	3	The design shall make use approximate computational circuits (adders) when performing interpolation algorithm	Pass. See section 2.2.2		X		
	4	The design shall process video frames into packets of varying size	Pass. See Appendix X				X
	5	The design shall introduce packet errors into a set of packets belonging to a video frame at a varying packet error rate	Pass. See Appendix E				X
	6	The design shall process video packets with varying packet error rates	Pass. See Appendix E				X
Modified	7	The pixel colour generated by approximate added should have a CIELAB Delta E* value under 10 when compared to the average produced by an accurate adder[1]	Pass. See section 3.2.1				X
	8	Power usage by the implementation should be kept to a minimum	Pass. See 3.2.3	X			
	9	The resolution and frame rate of the output video should match the input video	Pass. See Appendix F			X	
	10	Computational time of the algorithm shall not exceed 20 ms, the minimum time in which input lag becomes perceptible to the user [2]	Pass. See section 3.1.2			X	

	11	Approximations performed to generate output video shall not result in loss of subjective video quality by users	Untested. See section 3.1.1 for explanation.				X
	12	Interpolation and approximate computational circuits shall be implemented on FPGA on DE1 board	Passed. See Appendix G and Appendix H.		X		

3.1 Pixel Estimation Algorithm (Author: Mitchell)

In designing the PEA care was taken to meet as many of the requirements as possible. All of the design requirements pertaining to the format of the output video have been met. Requirements 1, 4, 5, 6, and 9 are demonstrated in Appendix E and F, where figures pertaining to input and output resolutions, error rates, and packet size variance are shown. All remaining requirements which pertain to the PEA are explained in this section.

3.1.1 PEA Performance (Author: Mitchell)

The design is able to interpolate erroneous or missing pixel data to build video frames (Requirement 2). This is demonstrated in Figures 7 to 12. These figures show the effects of error correction at various error rates and cases of high and low movement within the frame. In the cases with lower error rates, there noticeably less striations and smudging within the frame especially around moving subjects (Figure 7).

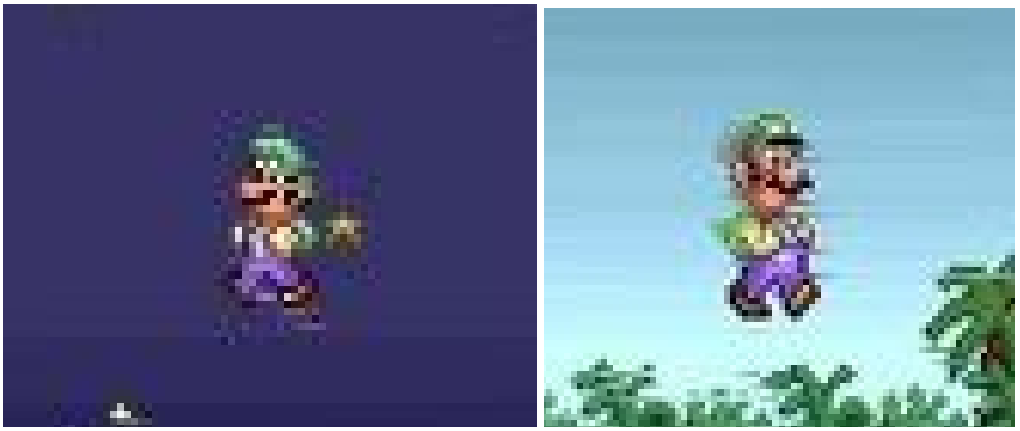


Figure 7: Striations and smudging around the high movement and high contrast subject (right) are more numerous than the low movement and low contrast subject (left).

Figure 8: Error correction performed on a low movement stream at error rate of 20%.
Top: Previous frame
Middle: Current error corrected frame
Bottom: Current error free frame

Figure 9: Error correction performed on a high movement stream at error rate of 20%.
Top: Previous frame
Middle: Current error corrected frame
Bottom: Current error free frame

Figure 10: Error correction performed on a high movement stream at various error rates.
Top: 10% error rate
Middle: 20% error rate
Bottom: 50% error rate

Figure 11: Error correction performed on a high movement stream at various error rates.
Top: 10% error rate
Middle: 20% error rate
Bottom: 50% error rate

Figure 12: Error correction performed on a low contrast stream at various error rates.
Top: 10% error rate
Middle: 20% error rate
Bottom: 50% error rate

From the results above it remains ambiguous as to whether the approximations performed generate output video with a loss of subjective video quality (Requirement 11). The ambiguity lies in the nature of the error itself, as the resultant approximation quality is heavily related to the error rate, the nature of the video itself (i.e. high movement vs. low movement or high contrast vs. low contrast), and for how many frames the error rate remains prominent for. When displaying video at 30 frames per second (fps) or 60fps an error rate of 10% may be completely imperceptible to a user, while a error rate of 20% in a video stream with high movement may be extremely obvious. In order to test this requirement a large survey spanning many varying test cases and many users would be necessary.

3.1.2 PEA Delay (Author: Ruben)

The computational time of the algorithm does not exceed 20 ms which is the minimum time where input lag becomes perceptible to the user (Requirement 10). The latency of the algorithm on a frame by frame basis is simple to calculate given the number of states in the PEA (Figure 3 and Figure 4) and the speed of the clock. For the following calculations three assumptions will be made:

1. Within the DE1-SOC the base clock speed is 50 Mhz, which will be the frequency used.
2. All values used will represent the steady state case of error correction, which discludes initialization states.
3. The size of the frame will be fixed to the size of frame used in testing: 256 by 224

The delay is calculated as follows:

$$Let\ error\ rate\ be\ defined\ as\ \varepsilon$$

$$Delay\ Per\ State\ = \frac{1}{50\times10^{6}Mhz} = 2\times10^{-8}sec\ = \Delta$$
$$Number\ of\ states\ for\ erroneous\ packet = 15\times\frac{pixels}{packet}$$
$$Number\ of\ states\ for\ error\ free\ packet\ = 2$$

$$Total\ Delay\ = \Delta(\ 15\frac{pixels}{packet}\times\frac{packet}{frame}\ \varepsilon\ + 2\ \frac{packet}{frame}\ (1-\varepsilon))$$

With the equation for delay formulated, Table 5 shows the delays for a selection of error rates and packet sizes.

Table 5: Total Delay in Seconds Given Packet size and Error Rate

Error Rate	**16 pixels/packet**	**32 pixels/packet**	**64 pixels/packet**
10%	1.84e-3	1.78e-3	1.65e-3
20%	3.55e-3	3.49e-3	3.27e-3

30%	5.26e-3	5.21e-3	4.89e-3

Given these results, for a reasonable amount of error the delay will never exceed 20ms per frame. This complies with Requirement 10.

3.2 Approximate Adder

The following section describes the performance of the approximate adder on the metrics of accuracy, speed, and power consumption.

3.2.1 Approximate Adder Accuracy (Author: Mitchell)

When testing the accuracy of the approximate adder three different cases were used in order to simulate the different regions of an image that the PEA would have to calculate color values within. They are as follows.

1. The **Specific Case:** A single color value was selected to surround the target pixel. This case acts as an analog to a solid color being displayed on the video stream. A representation of this case is shown in Figure 13.
2. The **Random Case:** Random color values were selected to surround the target pixel. This case acts as an analog to edges of colour or vibrant images being displayed in the video stream. A representation of this case is shown in Figure 14.
3. The **Constrained Delta Case:** Colour values within a delta of 20 were selected to surround the target pixel. This case acts as an analogue to gradients of color being displayed in the video stream. A representation of this case is shown in Figure 15.

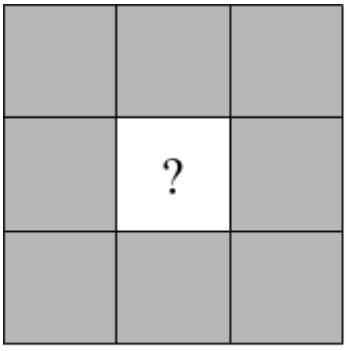


Figure 13: The Single Value Case

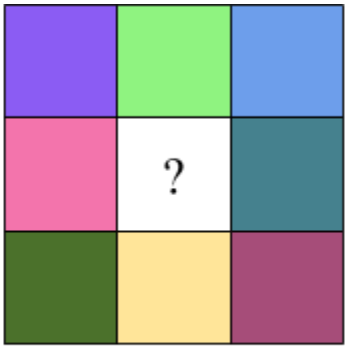


Figure 14: The Random Value Case

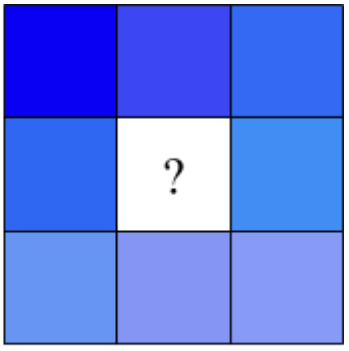


Figure 15: The Constrained Delta Case

The performance of the approximate adder in these three cases can be found in Table 6. For comparison, the averages produced by the approximate adder shown against the averages produced by an accurate full adder. The delta between these color values as well as their percent difference is also shown in Table 6.

Table 6: Results of Approximate Adder Averages vs. Full Adder Averages

Input	Approximate Adder Result	Full Adder Result	Avg Pixel Delta	Avg. % Difference	CIE76 Deta E
Specific Case: All surrounding pixels the same colour (value of 146 on each channel).	Total: 1076 Pixel Avg: 134	Total 1168 Pixel Avg: 146	12	4.7%	4.62
Random Case: All surrounding pixels are assigned random color values.	Overestimates pixel values by a small margin.	N/A	17.1	6.7%	6.73
Constrained Delta Case: All surrounding pixels are of a value with delta within 20 (~7.8%).	Overestimates and underestimates near equally.	N/A	12	4.7%	4.62

Note: For full details on the values used for the Random and Constrained Delta cases see Appendix C and D.

From the results in Table 6 (above) the delta is lower for cases of similar colors (Specific Case and Constrained Delta Case) than it is for colors that are different from one another (Random Case). This bodes well for the approximate adder, as most video consists of areas of similar color rather than many regions of drastically changing color. A representation of the differences in the colors calculated by the approximate adder and the full adder can be seen in Table 7 (below).

Table 7: Colors Calculated by Approximate Adder vs. Full Adder

Input	Full Adder Color 1	Approximate Adder Color 1	Full Adder Color 2	Approximate Adder Color 2
Specific Case			N/A	N/A
Random Case				
Constrained Pixel Delta				

Note: Colors were created from values in Appendix C and D that were centered about the average delta. The cases with constrained pixel deltas produces a more subtle color differences.

3.2.2 Approximate Adder Delay (Author: Ruben)

In order to calculate the relative speed of the approximate adder, we used a comparison between the critical paths of the approximate adder and a full adder. A view of the arrangement of approximate adders used in the PEA can be seen in Section 2.2.2, Figure 4. The analogous structure of full adders used for comparison can be seen in Figure 16.

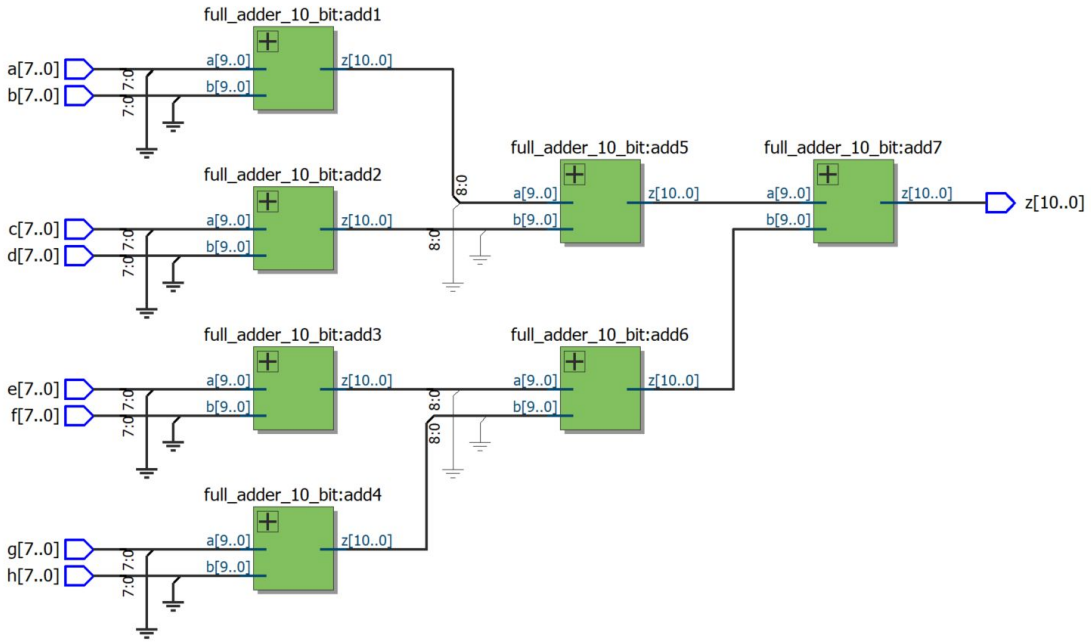


Figure 16: Structure used for full adder delay comparison.
Taken from a screen capture of Questa's RTL viewer

In order to calculate the critical path, the number of gates per adder must be known. The critical path through a chain of single bit adders is 1 XOR, 1 AND, and 1 OR gate for the first adder in the chain, while successive adders have 1 AND and 1 OR gate.

Approximate Adder	Full Adder
The approximate adder uses multiple 4 or 5 bit adders per layer in order to calculate it's approximate result. This means that per layer the critical path passes through 4 to 5 single bit adders.	The full adder uses 10 bit adders per layer in order to calculate it's exact result. This means that per layer the critical path passes through 10 single bit adders.
Length of critical path: $Approx.\ Adder\ = 3XOR\ +\ 14AND\ +\ 14OR$ Delay through critical path: $XOR_{delay}\ =\ 0.478$ $OR_{delay}\ =\ 0.313$ $AND_{delay}\ =\ 0.236$ $Total_{delay}\ = 3(0.478)\ +\ 14(0.236)\ +\ 14(0.313)$ $Total_{delay}\ = 9.12$	Length of Critical Path: $Full\ Adder\ =\ 3XOR\ +\ 33AND\ +33OR$ Delay through critical path: $XOR_{delay}\ =\ 0.478$ $OR_{delay}\ =\ 0.313$ $AND_{delay}\ =\ 0.236$ $Total_{delay}\ = 3(0.478)\ +\ 33(0.236)\ +\ 33(0.313)$ $Total_{delay}\ = 19.55$

* Delays of individual gates acquired through timing analysis in Quartus

3.2.3 Approximate Adder Power (Author: Mitchell, Ruben)

In order to verify the power usage of the approximate adder, verification through similarity was used. Borrowing from the work at [3] we created an approximate adder that was analogous in design. Given the findings for the 16-bit adder that the paper describes it follows that the 8-bit, 9-bit, and 10-bit version of the adder we developed should follow the same pattern. The low power usage described by [3] fulfills requirement 8.

4.0 Summary and Conclusions (Author: Ruben)

The design team produced an error-tolerant video processing system that is appropriate for real-time video applications using unreliable communication channels as a synthesizable hardware module that performs error correction on video frames using approximate adders. The module was verified against the requirements of the project and was largely successful. The major success of the project was in obtaining quantitative requirements pertaining to performance, such as, functional correctness, colour accuracy and timing constraints.

All requirements regarding output video format, generation of errors, and timing were achieved. However, aspects of the design that require further verification are subjective video quality as perceived by users. In terms of subjective video quality, in order to perform user testing with people, the developed system must be extended and integrated into an end-to-end system that will display the corrected video stream in its native format. In order to extend the system, the pre-processing system should be further automated to generate a steady video stream, the PEA hardware module should be integrated as an Quartus IP that might communicate with an ARM processor, and the frame buffers could easily be read into a VGA adapter to be displayed on a monitor. In testing, various error rates, packet sizes, and types of video should be shown to a wide range of subjects.

In conclusion, the final design serves as a proof-of-concept of an appropriate video processing system that might be used in applications such as wireless VR systems, where the system might exhibit error-tolerance, unreliable communication channels, and hard timing and power constraints. While these systems might not make use of an FPGA the final design might be implemented as an ASIC.

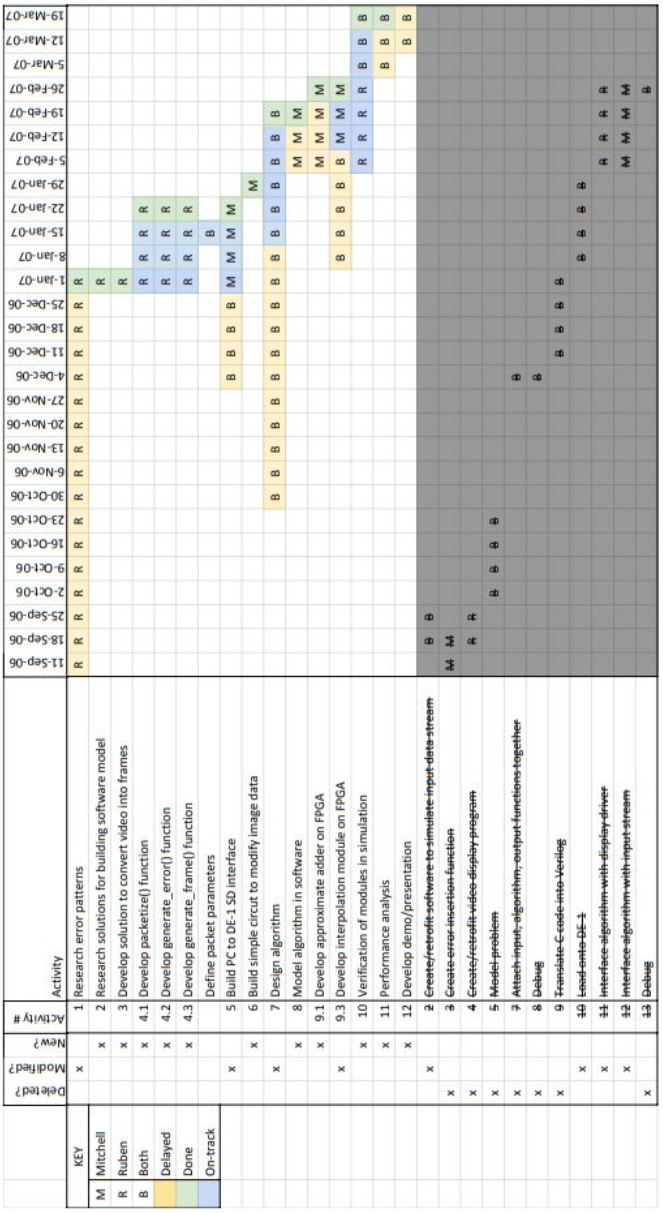
5.0 References

[1]"Color difference", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Color_difference. [Accessed: 25- Oct- 2017]

[2] Qualcomm. March 2016. Making Immersive Virtual Reality in Mobile Available: https://www.qualcomm.com/documents/whitepaper-making-immersive-virtual-reality-possible-mobile

[3] Kahng, Andrew B. and Seokhyeong Kang. “Accuracy-configurable adder for approximate arithmetic designs.” DAC Design Automation Conference 2012 (2012): 820-825.

6.0 Appendices



Appendix A: Gantt Chart History

Appendix B: Validation and Acceptance Tests (Old)

Requirement	Requirement Verification Method			
	Similarity	Review	Analysis	Test
Frame rate shall not be reduced from source video				X
Users do not detect a loss in subjective video quality				X
Time added by algorithm shall not				X

exceed 20ms				
Algorithm shall execute on an Altera DE1 Board		X		
Approximation of pixel colour should have CIELAB Delta E* value under 1.0			X	

Many more requirements were added, as shown in Table 4. The last entry in the above table was modified due to overconstraint. A CIELAB Delta E* of 1 is extremely small and was not reasonable for this project. The value was updated to 10 to provide a more reasonable benchmark.

Appendix C: Raw Adder Results For Random Distribution:

Full Adder Total	Full Adder Pixel Value	Approximate Adder Total	Approximate Adder Pixel Value	Approx vs. Full Adder Pixel Δ	Approx vs. Full Adder Pixel % Difference
1280	160	1075	134	26	10.2
622	77	379	47	30	11.8
958	119	827	103	16	6.3
939	117	946	118	1	0.4
848	106	631	79	27	10.6
603	75	526	66	9	3.5
1414	176	1492	187	11	4.3
1051	131	922	115	16	6.3
1224	153	1170	146	7	2.7
881	110	951	119	9	3.5
1165	145	875	109	36	14.1
			Average:	17.1	6.7

The above table was created using the following pixel values. Note that each row is comprised of uniformly distributed values between 0-255.

Row 1: 14, 145, 98, 200, 250, 180, 200, 193
Row 2: 100, 0, 22, 55, 200, 180, 37, 28
Row 3: 134, 240, 120, 90, 73, 48, 52, 201
Row 4: 159, 65, 126, 63, 188, 131, 34, 173
Row 5: 142, 154, 173, 141, 4, 17, 41, 176
Row 6: 4, 90, 36, 55, 143, 77, 70, 128
Row 7: 140, 227, 85, 88, 245, 196, 184, 249
Row 8: 45, 173, 173, 252, 120, 13, 194, 81
Row 9: 152, 232, 248, 234, 27, 153, 102, 76
Row 10: 147, 216, 149, 146, 9, 68, 33, 113
Row 11: 160, 81, 208, 228, 148, 110, 28, 202

Appendix D: Raw Adder Results for Constrained Delta:

Full Adder Total	Full Adder Pixel Value	Approximate Adder Total	Approximate Adder Pixel Value	Approx vs. Full Adder Pixel **Δ**	Approx vs. Full Adder Pixel % Difference
1181	147	1079	135	12	4.7
889	111	973	122	11	4.3
308	38	460	58	20	7.8
1937	242	2036	255	13	5.1
1302	162	1173	147	15	5.9
1044	130	888	111	19	7.5
82	10	8	1	9	3.5
95	11	7	1	10	3.9
1342	167	1411	176	9	3.5
1329	166	1347	168	2	0.8
			Average:	12	4.7

The above table was created using following pixel values. Note that each row is comprised of values with a delta no greater than 20 and each row is centered around a uniformly distributed value between 0-255.

Row 1: 155, 141, 145 , 141 , 158 , 144 , 150 , 147
Row 2: 116, 100, 120 , 105 , 120 , 103 , 109 , 116
Row 3: 41, 32, 44, 32, 39, 45, 34, 41
Row 4: 236, 245, 247 , 241 , 230 , 250 , 248 , 240
Row 5: 164, 169, 163 , 168 , 163 , 155 , 157 , 163
Row 6: 127, 135, 137 , 133 , 140 , 121 , 123 , 128
Row 7: 14, 19, 2 , 8 , 13, 7 , 12, 7
Row 8: 10, 9, 14, 18, 16, 13, 11, 4
Row 9: 167, 159, 175 , 166 , 175 , 169 , 161 , 170
Row 10: 165, 166, 170 , 156 , 173 , 174 , 167 , 158

Appendix E: Error Rates and Packet size Variance

Figure X: Error packet sizes of 16 pixels, 32 pixels, and 64 pixels (Requirement 1, 4)

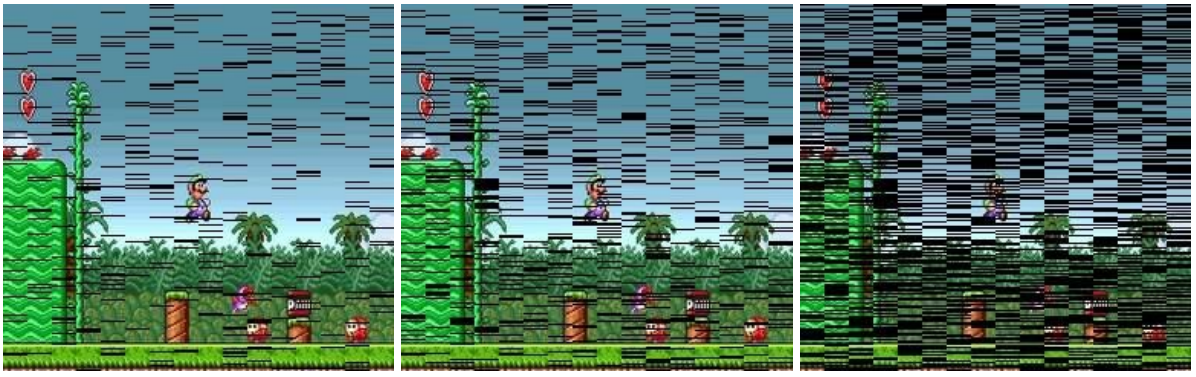
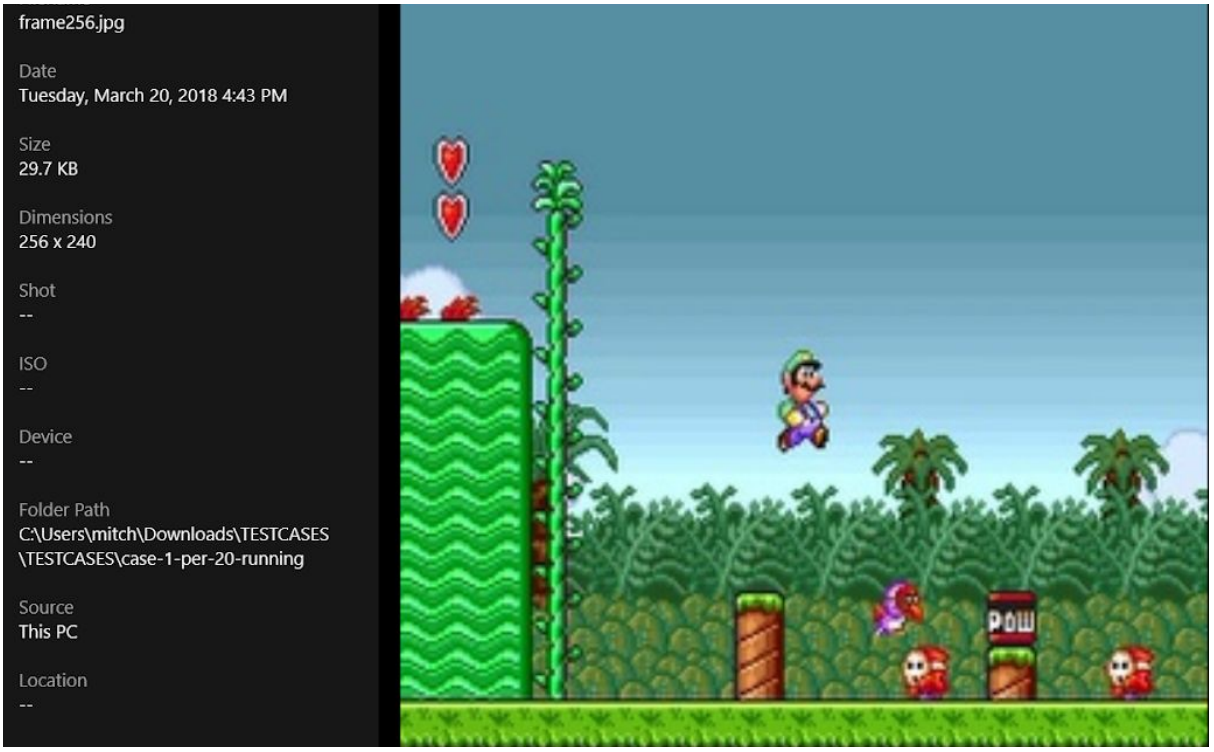
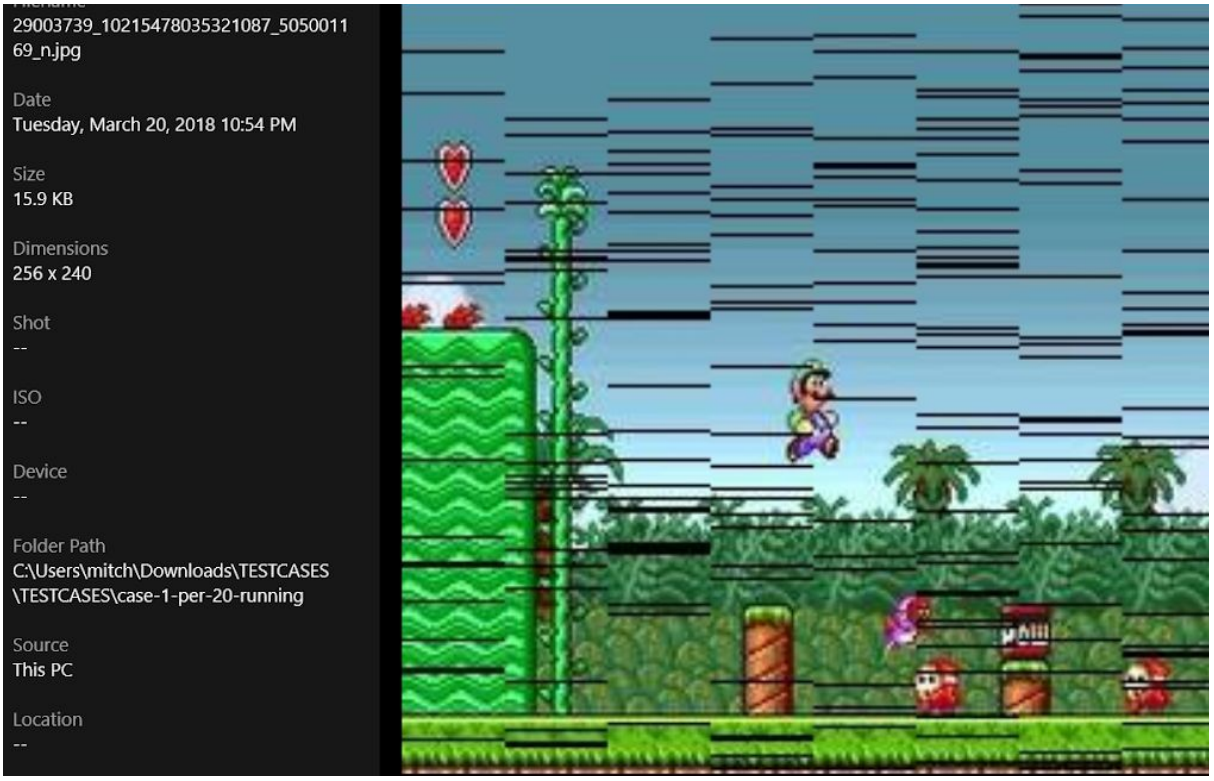


Figure X: Error rates of 10%, 20%, and 50% (Requirement 5, 6)

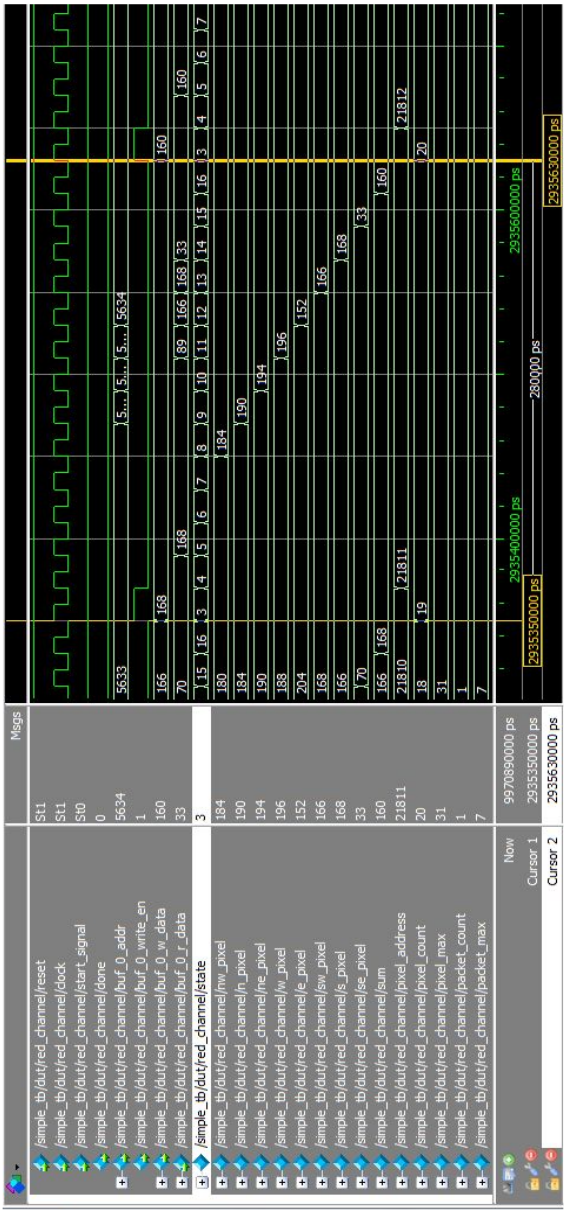
Appendix F: Input and Output Resolution



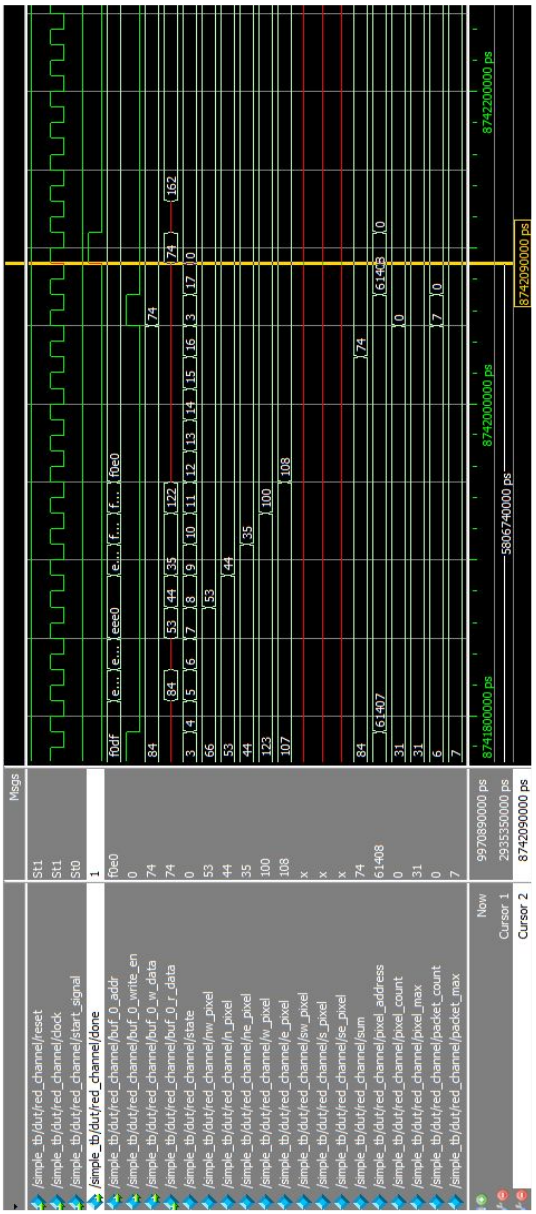


Original image (top) and image with injected error (bottom) are the same resolution. This satisfies Requirement 9.

Appendix G: Raw PEA Simulation Results

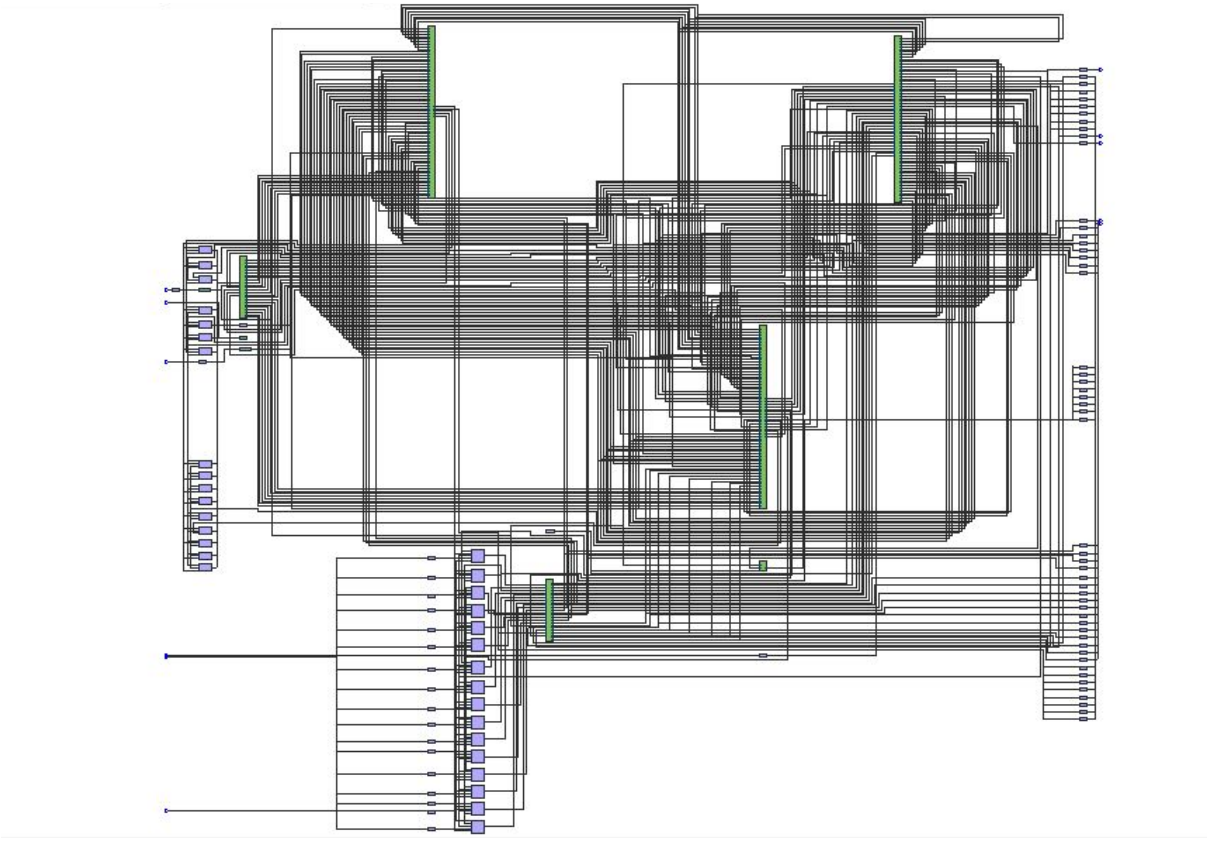


The above results show the PEA fetching the relevant colour values needed for approximation and the calculation of the approximate sum.



The above results show the PEA fetching the relevant colour values needed for approximation and the calculation of the approximate sum in the case where we are at the bottom edge of an image. Note that the values of SW, S, and SE pixel are all in a don't care state.

Appendix H: RTL Generated view of the top level design



Appendix I: FSM Next State Logic Pseudocode

```
always @ (posedge clock, negedge reset)
    begin
        if(!reset)
        begin
            state <= WAIT;
        end
        else
        begin
            case(state)
                WAIT: begin
                    if(start_signal) begin
                        state <= LOAD_MID_ERR;
                    end
                    else begin
                        state <= WAIT;
                    end
                end

                LOAD_MID_ERR: begin
                    state <= LOAD_BTM_ERR;
                end

                LOAD_BTM_ERR: begin
                    state <= CHECK_PACKET_ERR;
                end

                CHECK_PACKET_ERR: begin
                    // If error bit is high we need to correct.
                    if (err_buf_mid[15-packet_count] == 1'b1)
begin
                        state <= NW_PIXEL;
                    end
                    // No errors. Proceed to next packet.
                    else begin
                        if (packet_count == packet_max) begin
                            if (line_count == LINE_MAX) begin
                                state <= DONE;
                            end
                            else begin
                                state <= LOAD_BTM_ERR;
                            end
                        end
                        else begin
                            state <= CHECK_PACKET_ERR;
```

```
        end
    end
end

NW_PIXEL: begin
    state <= NORTH_PIXEL;
end

NORTH_PIXEL: begin
    state <= NE_PIXEL;
end

NE_PIXEL: begin
    state <= WEST_PIXEL;
end


WEST_PIXEL: begin
    state <= EAST_PIXEL;
end

EAST_PIXEL: begin
    state <= SW_PIXEL;
end

SW_PIXEL: begin
    state <= SOUTH_PIXEL;
end

SOUTH_PIXEL: begin
    state <= SE_PIXEL;
end

SE_PIXEL: begin
    state <= MEM_WAIT_1;
end

MEM_WAIT_1: begin
    state <= MEM_WAIT_2;
end

MEM_WAIT_2: begin
    state <= MEM_WAIT_3;
end
MEM_WAIT_3: begin
    state <= SUM;
```

```
            end
            SUM: begin
                state <= WRITE_RESULT;
            end

            WRITE_RESULT: begin
                if (pixel_count == pixel_max) begin
                    if (packet_count == packet_max) begin
                        if(line_count == LINE_MAX) begin
                            state <= DONE;
                        end
                        else begin
                            state <= LOAD_BTM_ERR;
                        end
                    end
                    else begin
                        state <= CHECK_PACKET_ERR;
                    end
                end
                else begin
                    state <= CHECK_PACKET_ERR;
                end
            end

            DONE: begin
                state <= WAIT;
            end
        endcase
    end
end
```

Appendix J: RTL view of 8-bit Approximate Adder and Approximate Adder Superstructure