

# Pipelined Hardware Video Compressor & Decompressor

**Team:** sdmay 24-12

**Members:** Kareem Eljaam, Caleb Rock, Benjamin Meinders,  
Colsen Selk, and Logan McDermott

**Client:** John Deere (Nathan Francque)

**Faculty Advisor:** Dr. Joseph Zambreno

**Website:** <https://sdmay24-12.sd.ece.iastate.edu/>

# Table of Contents

- Introduction/Background..... 3
  - Problem Statement..... 3
  - Intended Users/Uses..... 3
  - Related Context..... 3
- Revised Design..... 4
  - Requirements..... 4
  - Engineering Standards..... 4
  - Security Concerns and Countermeasures..... 5
  - Design Evolution..... 5
- Implementation Details..... 5
  - Detailed Design..... 5
- Testing..... 9
  - Process..... 9
  - Results..... 10
- Broader Context..... 13
- Conclusions..... 15
  - Progress Review..... 15
  - Discussion of Value..... 15
  - Future Steps..... 16
- Appendices..... 16**
  - Appendix 1 - Operation Manual..... 16
  - Appendix 2 - Design Iterations..... 20
  - Appendix 3 - Other Considerations..... 21
  - Appendix 4 - Code..... 21

## Introduction/Background

### Problem Statement

This project's goal is to increase the amount of computer vision workload that can be handled by an FPGA while reducing the on-chip RAM usage by using pipelineable compression and decompression cores.

### Intended Users/Uses

This project is intended to be used by John Deere for video processing and computer vision applications. Computer vision applications often require a large amount of convolutions to be performed as fast as possible. An FPGA can perform these operations at near-zero latency. However, it is currently limited by the amount of RAM available to store line buffers. The intended use of our project is to place a compression core at the input of each convolutional line buffer and a decompression core at the output of each convolutional line buffer.

### Related Context

As agricultural and construction machinery increases its use of cameras and video, there is a tremendous need to process and store that data efficiently. John Deere uses computer vision for many new technologies, such as "See & Spray," a technology on John Deere's new sprayers that utilizes machine learning to differentiate between crops and weeds. Each sprayer nozzle comes with a camera and sprays the weeds with weed killer while leaving the crops alone. These advancements in farming technology are helping to feed the world, and improving the performance of this new technology advances the previous progress that John Deere has made.

# Revised Design

## Requirements

- Provide a standard image set for testing the algorithm via software to quantify the loss.
- Create lightweight compression and decompression cores that are mappable to FPGA or ASIC to minimize on-chip RAM usage.
- Compression and decompression are to be performed on a live input stream.
- The latency should be low enough to allow for near-zero latency computer vision processing.
- Prioritize logic simplicity and pipelineability over compression ratio.
- Solution should be fully pipelineable, a key requirement to this is having each encoded pixel of a predetermined size.
- Demonstration should show HDMI output from the FPGA to a 1920x1080 resolution monitor.

## Engineering Standards

- Python, Java, and C++ for basic prototyping of the compression algorithms
- VHDL for the FPGA IPs
- Vitis HLS
- LZW video compression
- BWT data transformation
- AXI-Stream Video Formatting
- HDMI for transmitting the video from input and to output
- Data-size standardized formats (the Byte)
- Python for basic prototyping of the compression algorithms
- Java and C++ as well.
- VHDL for the FPGA IPs
- Vitis HLS
- LZW video compression
- BWT data transformation
- AXI-Stream Video Formatting
- HDMI for transmitting the video from input and to output
- Data-size standardized formats (the Byte)

## Security Concerns and Countermeasures

The only security concern for our project would be any third-party IPs we used hiding secret code that steals data streamed through it. This concern is counter-measured by using only IPs from reputable sources (Such as AMD).

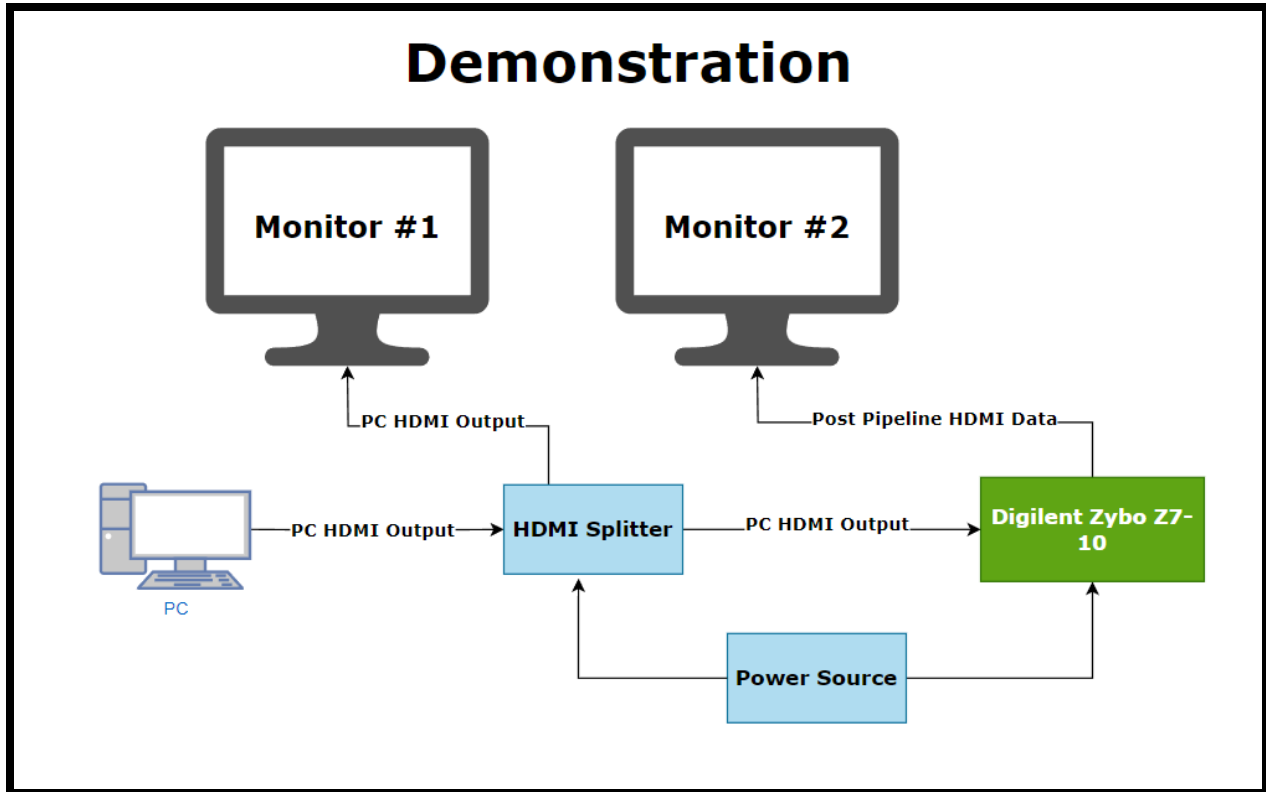
## Design Evolution

Our initial design had LZW compression and decompression, but our testing showed that this algorithm would not be sufficient for compression with images, let alone video. We decided to move to using pre-made compression and decompression IPs from the XILINX marketplace, and modifying those to fit with our near-zero latency HDMI passthrough system. This approach failed because of the extreme cost of these IPs. We also tried to import the Vitis HLS Data Compression Library into our project. While we could generate IPs using the HLS tool, we encountered problems with FPGA space and input/output compatibility. Finally, we decided to create our own IPs with simple compression and decompression schemes and successfully implemented this strategy. We also made a design decision to do the compression and decompression after the video stream had already made it through the VDMA. We made this decision because our new compression had a fixed size of 16 bits on the output. The original potential use of the VDMA was to use software to read the VDMA and provide metrics; however, since we have a fixed compression ratio, we no longer have to collect that data.

## Implementation Details

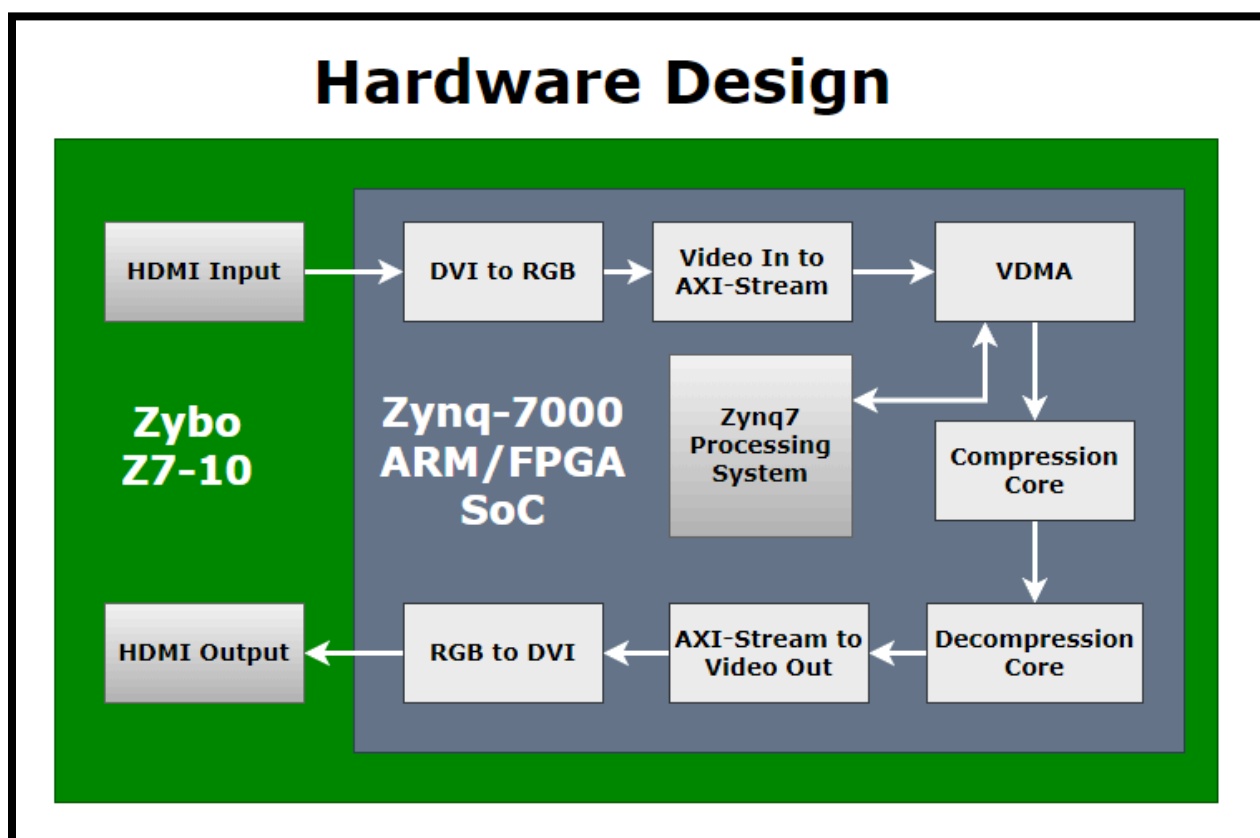
### Detailed Design

To describe our final implementation, we divided the discussion into three parts, each with its own visual aid. The sections are built top-down, starting from our top-level demonstration setup, then a flowchart outlining the functionality of our board, and lastly, we highlighted the details of our compression scheme below.



*Figure 1*

Figure 1 is a diagram that shows the top-level layout of our demonstration. Our project emphasized a need for very low-latency compression. To highlight our system's near-zero latency, we chose to demonstrate the HDMI video before and after the compression/decompression pipeline. This approach allows us to inspect any visual latency and analyze the effects of our loss side-by-side with the raw video version. Monitor 1 displays the video from the video source PC, and Monitor 2 displays the video from the output of our hardware pipeline. The demonstration runs using our Vivado-generated hardware platform file (.xsa) and our C source code from Vitis to flash the board with the hardware/software we designed while the PC drives the HDMI source.



*Figure 2*

Figure 2 lays out the hardware design and demonstrates the different components we used to build the pipeline. We chose the Zybo Z7-10 because it supports interfacing with HDMI Input and Output. The Zybo Z7-10 is also home to a Zynq-7000 ARM/FPGA SoC that allows us to interface with and map our solution to an FPGA. The plan for our design was to establish an HDMI passthrough and then add in our custom cores as needed. The pipeline starts by taking HDMI input and converting it to RGB using an IP provided by Vivado. We then take the raw RGB data and convert it into an AXI-Stream using other IP provided by Vivado. The Zynq7 processing system was initially used in our project to collect performance metrics. However, the final implementation only uses the processing system for configuring the VDMA and Video Timing Controllers (not included in the schematic above). Our design runs the compression and decompression after reading the frame buffer from the VDMA. Details regarding the compression scheme are outlined below in Figure 3. After the compression and decompression, the AXI-Stream must be converted back into HDMI output. To do this, we used the AXI-Stream to Video Out IP and the RGB to DVI IP to go from AXI-Stream to HDMI data for output from the Zybo Z7-10.

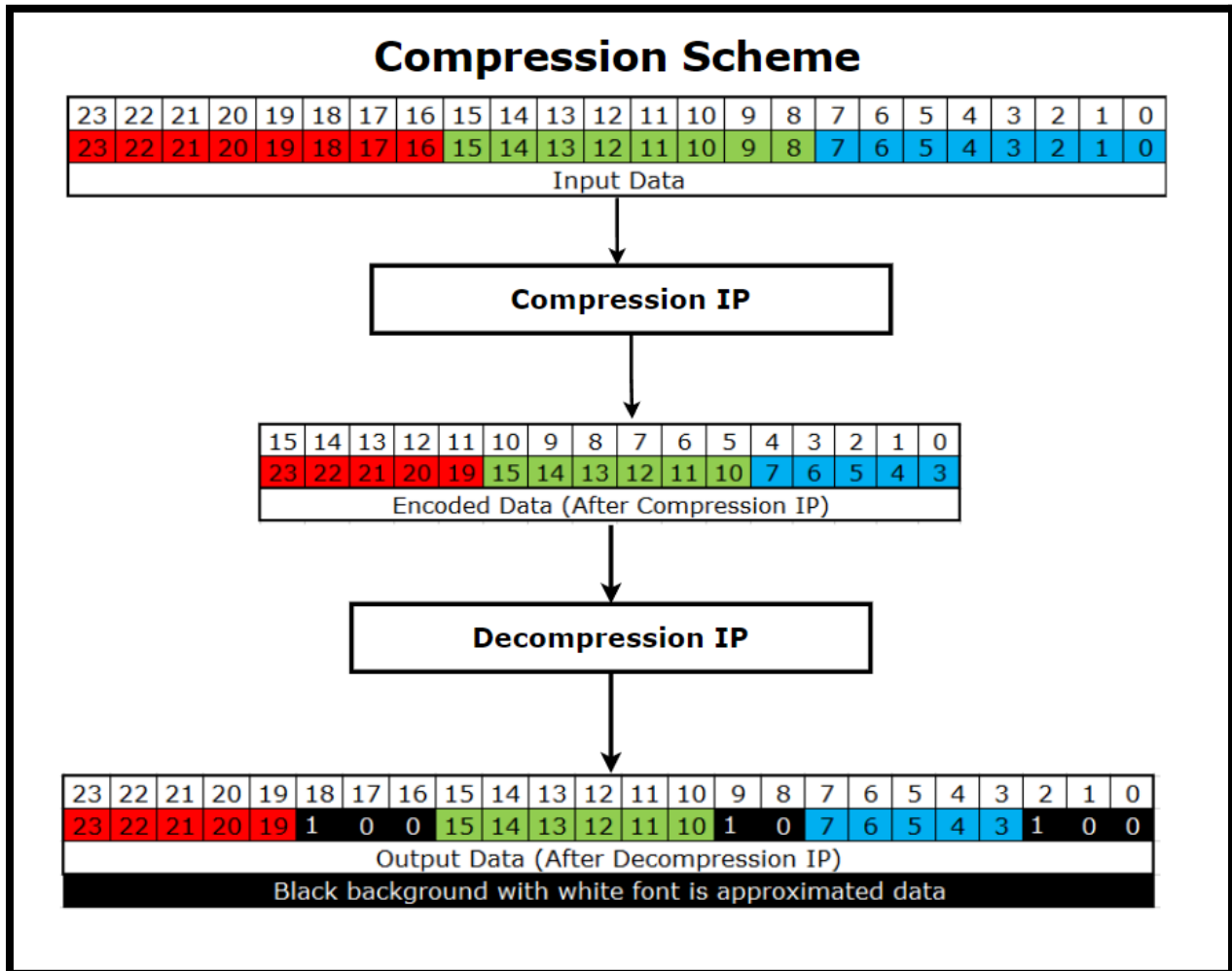


Figure 3

Figure 3 shows the compression scheme used in our final implementation. The algorithms are simple yet effective, especially when considering the emphasis in the requirements placed on our solution to be lightweight and logically simple. This compression scheme drops the least significant three bits for the Red and Blue channels and the bottom two least significant bits from the Green channel. This approach allows the amount of RAM needed to store the data to be only  $\frac{2}{3}$  of the original RAM needed while only needing one clock cycle to compute this. We chose an RGB scheme of 5:6:5 instead of 6:5:5 or 5:5:6 because human eyes are more sensitive to variations in green light than in red or blue. The encoded data is then directly fed into the decompression IP so it can be converted back to 24 bits. To recover the lost bits, we simply add “100” for three bits and “10” for two bits. We wrote a software program that found that “000” may be the most common. However, we still chose “100” as it is the median value between 1'b000 and 1'b111, and the distribution is near



normal. This compression scheme allows us only to use  $2/3$  of the original RAM requirement while achieving an error rate close to 0%. Although we are losing  $1/3$  of the original bits, the furthest off the red and blue component can be from the original is 4, and the furthest off the green can be is 2. This scheme yields a picture that saves a lot of data while maintaining the original image effectively by leveraging the most significant bits.

## Testing

### Process

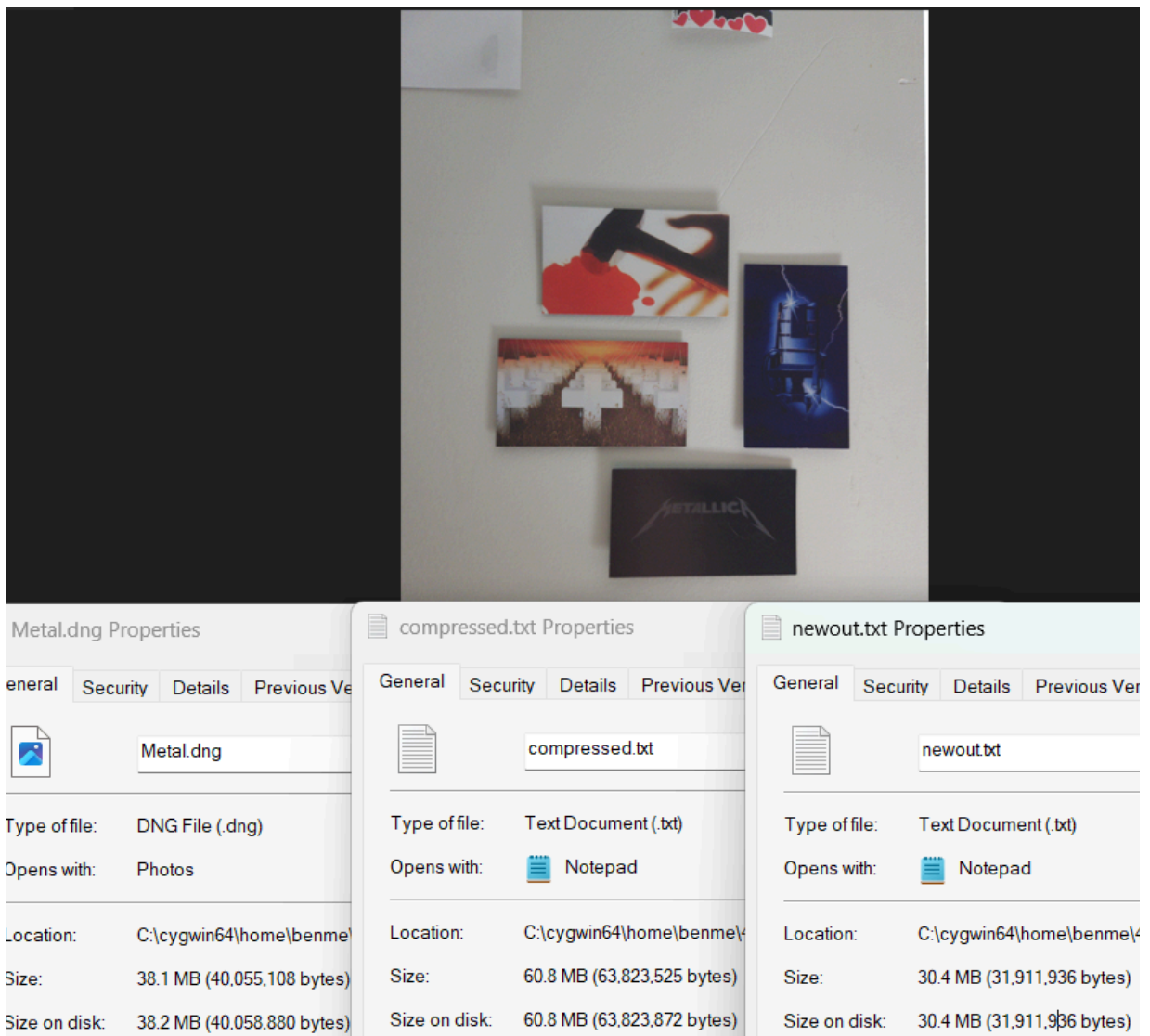
Our software testing for LZW compression consisted of a Java implementation of LZW coded by Benjamin. The LZW compression was also interfaced with a BWT in python in order to facilitate more compression-efficient results.

For hardware, testing was slightly challenging. One of our primary tests was to visually inspect to see if our implementation is working, which admittedly is not the most robust. While debugging our design we also used Internal Logic Analyzers provided by Vivado that act like miniature oscilloscopes within the FPGA. We confirmed that our compression/decompression was acting as expected when looking at the waveforms from the ILAs. The main issue we ran into while testing was having HDMI not output. There was very rarely an error that did not break HDMI output.

Since the hardware was challenging to test formally, we used software to get some of our metrics. The compression standards implemented in hardware were also implemented in software to allow us to see our data loss. We used raw images as the input of a software program that compared the compressed and decompressed data to the original data and found the percentage error difference.

## Results

Our software testing showed that LZW was insufficient to use for compression because we saw an increase in our raw image size of 50% in the compressed form, even with a BWT formatting used beforehand. One such test showed a Raw file image (.RAF) going from 55 to 85 MB of data. We found that the images did not have enough patterns in the data to be useful for compression by LZW. Because LZW is one of the only compression algorithms that would be simple enough for our small team to implement in compression, this prodded us to come up with a plan to either focus on premade compression algorithms such as those already designed by AMD for the Zynq-7000 or a simple lossy algorithm for compression and decompression.



The image shows a photograph of a wall with several posters and a 'METALLIC' sign. Overlaid on the bottom of the image are three Windows file property windows. The first window is for 'Metal.dng', the second is for 'compressed.txt', and the third is for 'newout.txt'. Each window shows the file's type, size, and location.

File Name	Type of file	Opens with	Location	Size	Size on disk
Metal.dng	DNG File (.dng)	Photos	C:\cygwin64\home\benme\	38.1 MB (40,055,108 bytes)	38.2 MB (40,058,880 bytes)
compressed.txt	Text Document (.txt)	Notepad	C:\cygwin64\home\benme\	60.8 MB (63,823,525 bytes)	60.8 MB (63,823,872 bytes)
newout.txt	Text Document (.txt)	Notepad	C:\cygwin64\home\benme\	30.4 MB (31,911,936 bytes)	30.4 MB (31,911,936 bytes)

*Figure 4*

Our hardware testing showed the quality and latency of the video coming out of the FPGA. Figure 5 shows the regular video streaming on the left monitor while the pipelined and decompressed video stream is on the right. This setup allowed us to see how the fine details of an image looked the same after the compression/decompression, but when there was a solid color (such as the sky), some distortion was introduced.

*Figure 5*

Figure 6, shown above, pictures the right monitor outputting raw data from the PC and the left monitor outputting the data after the compression/decompression pipeline. We are displaying a stopwatch from the laptop in order to identify any latency. We do not have a specific latency we were trying to meet; however, in this image, one can see that the HDMI data is going through the pipeline faster than it is getting to the display of the laptop driving the display. This demonstration meets the near-zero latency requirement as it supports the entire stream with zero visual lag.



Figure 6

Finally, shown below are three formulas that tell us a lot about our design's performance. The first calculation is the compression ratio, a metric that profiles how much data is being saved. We focused more on a fast throughput but still had a ratio of 1.5:1, meaning we only store  $\frac{2}{3}$  of the original data. The following formula is throughput. Our system uses a 134 MHz pixel clock and operates on 3 bytes per cycle. These settings allow our compression and decompression IPs to support up to 402 MB/s, which is more than enough for our use cases. The last formula is how we enumerate the loss of our data. It compares pre-pipeline and post-pipeline data and figures out the percentage of error. For images, we take the average error per pixel as our metric. Figure 8 below shows that our values were within 1 percent of the original value. This fact means our image is at least 99% "correct." This metric does not account for everything when considering lost data. However, it helped us find that RGB 5:6:5 was our best format and that by dropping less significant bits, our "correctness" is higher than the amount of data we are losing.

- $Compression\ Ratio = \frac{Original\ Data\ Size}{Compressed\ Data\ Size} = 1.5$
- $Throughput = Pixel_{Freq} \times \frac{\#\ Pixels}{cycle} \times \frac{\#\ Bytes}{Pixel} = 134\ MHz \times 1 \times 3 = 402\ \frac{MB}{s}$

- $$Pixel_{Error\%} = (Pipeline_{Pixel} - Original_{Pixel}) \times \frac{1}{256} \times 100$$

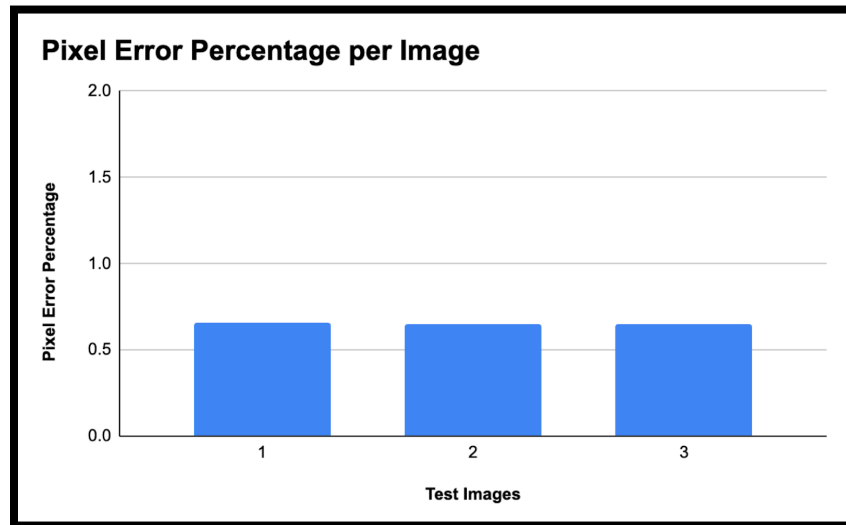


Figure 7

Figure 7 displays the values outputted after running three raw test images through our Python script, which implements the  $Pixel_{Error\%}$  equation shown above. As we can see, the average error percentage per pixel for the 3 test images is below 1%. Considering that we are compressing the size by 1/3rd of the original image size, having the average pixel error percentage below 1% is a major success.

## Broader Context

**Public Health, Safety, and Welfare:** This project affects the safety of a John Deere customer because the video compression/decompression pipeline can be used for enhanced machine vision capabilities, making the product safer. For example, compressing video data allows more cameras to be running at once, which can provide more data for an autonomous system.

**Global, Cultural, and Social:** As mentioned above, this project gives equipment the ability to have more cameras, which can improve autonomous capabilities. Autonomy will be essential to aid in the farm labor shortage, keeping the agricultural industry efficient and making food accessible to even more people.

**Economic:** This project decreases the memory needed, thus decreasing the hardware cost of equipment, allowing manufacturers to decrease production costs and the final sale price for farmers.

**Environmental:** With the decreased equipment cost, as mentioned previously, farmers can invest in equipment for environmentally friendly practices such as strip-tillers, which reduce soil erosion and increase organic matter in the soil.

# Conclusions

## Progress Review

This semester, we began creating the hardware side of the compression by developing an HDMI passthrough on the Xilinx Zynq-7000 Dev Board platform. At the same time, the software team began work on getting real-world numbers for the LZW compression algorithm. The software team's testing of LZW showed poor results and pivoted the team's approach. After the two teams finished their work, the hardware team and software team worked on attempting to use various IP's such as those from AMD, and other 3rd party IP's as well. Those compression IPs given by AMD were found to be incompatible with the AXI-stream. Our team managed to get a 3rd party compression algorithm working. However, there was no easy way to decompress from that algorithm. We determined that the compression algorithm would be useless for a demo. Our team then worked on creating a simple compression and decompression algorithm that we used in our final demo. After getting a working version of that algorithm, we tested various different parameters, like having more compression for blue values and less compression for red and green values. Given the roadblocks we have encountered that caused the course of our project to change, our implementation is excellent. We still managed to produce an implementation that met as many requirements and constraints as possible in the amount of time we had.

## Discussion of Value

First and foremost, we helped our client by proving the idea of near-zero latency compression and decompression of video is possible using a lightweight FPGA. We also delivered a working implementation for our client to work with. We provided insight into creating compression and decompression IPs and the feasibility of purchasing a premade IP. We also delivered a software program to optimize the compression loss rate depending on the type of images to be compressed (e.g., dry farm fields with lots of reds vs. growing farm fields with lots of greens). Finally, we created an excellent foundation for a future team of engineers to implement a more robust compression and decompression algorithm in hardware.

## Future Steps

Our implementation of our FPGA is a perfect starting point for a larger (or another senior design team) team to write and test more complex pipelined compression and decompression algorithms than our team of 5 could realistically develop. Larger, more robust compression and decompression algorithms allow for higher compression ratios and a lower loss rate while being relatively similar in latency. Given our project as a starting point, another team would not have to spend any time choosing hardware and building up the HDMI passthrough and would benefit from the knowledge we gained regarding purchasable and open-source FPGA solutions.

## Appendices

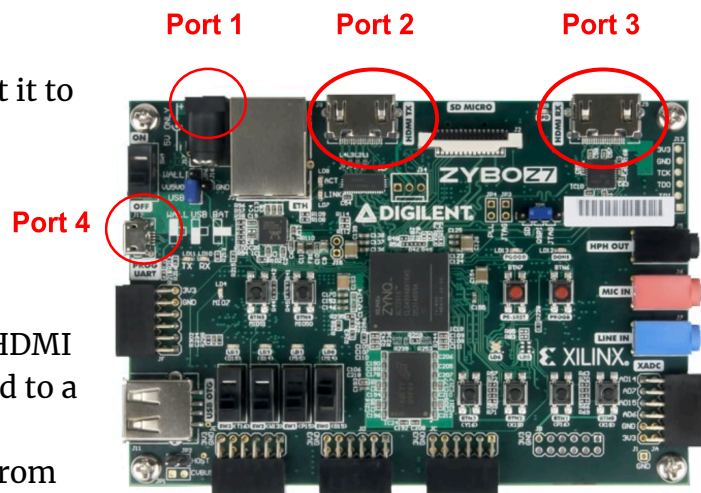
### Appendix 1 – Operation Manual

#### 1) Required Materials

- Zybo Z7-10
- 4 HDMI Cables
- 1:2 HDMI Splitter
- MicroUSB to USB Cable
- PC
- 2x Power Adapters (Zybo & HDMI Splitter)
- 2x HDMI Monitors
- Vivado 2020.1
- Vitis 2020.1

#### 2) Physical Setup

- I. Plug in the Zybo board's power adapter in port 1 and then connect it to a nearby outlet.
- II. Plug in the micro-USB side of the micro-USB to USB Cable into port 4, then plug in the USB side into your computer.
- III. Plug in the power adapter of the HDMI splitter and then have it connected to a nearby outlet.
- IV. Connect one of the HDMI cables from your computer to the input port on the HDMI splitter.





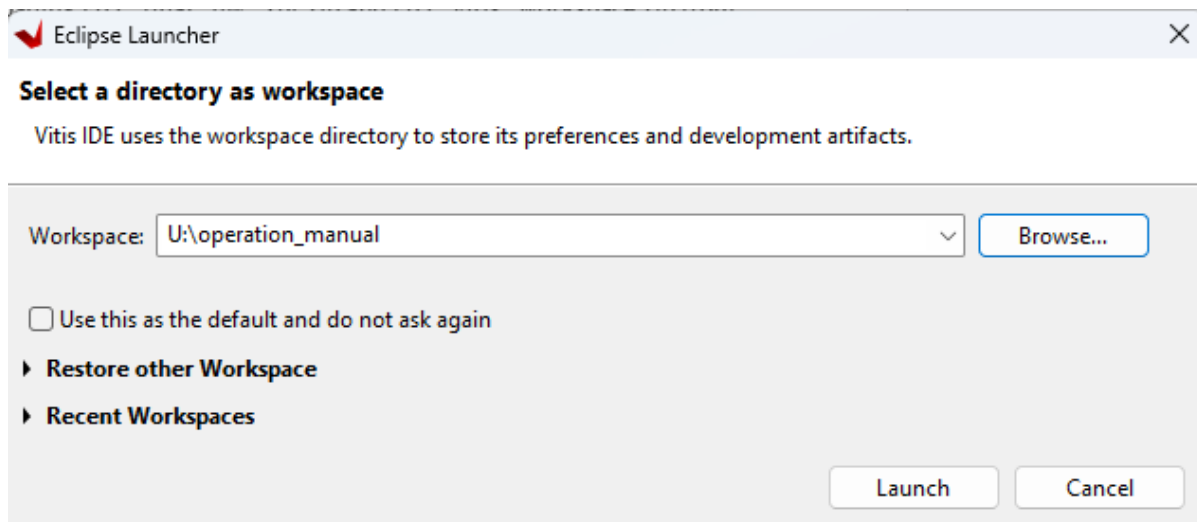
- V. Connect the second HDMI cable from the output port on the HDMI splitter to port 3 (HDMI RX) on the Zybo board.
- VI. Connect the third HDMI cable from the other output port on the HDMI splitter to the first monitor's HDMI port.
- VII. Connect the fourth HDMI cable from port 2 (HDMI TX) on the Zybo board to the second monitor's HDMI port.
- VIII. Finally, make sure that the switch above port 4 on the Zybo board is flipped to "ON". You may now proceed to step 3.

3) Start by downloading vitis\_492\_final.ide.zip from <https://git.ece.iastate.edu/sd/sdmay24-12>

4) Open Vitis 2020.1 on your machine (The icon should look like this) :

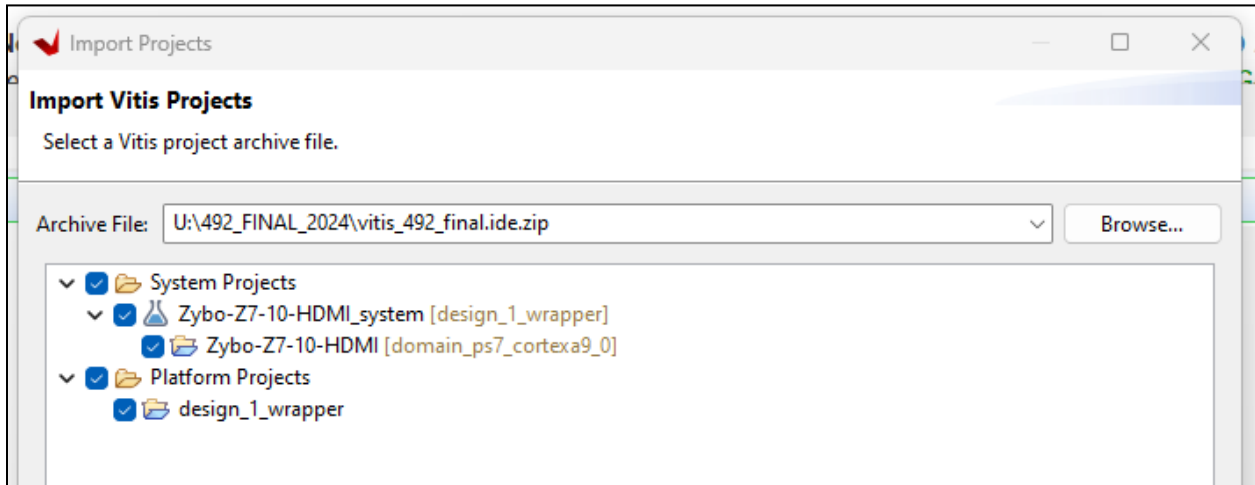


5) Select a workspace to hold all of the associated directories and files from this project and press launch, if you choose one that has not been created Vitis will create it for you

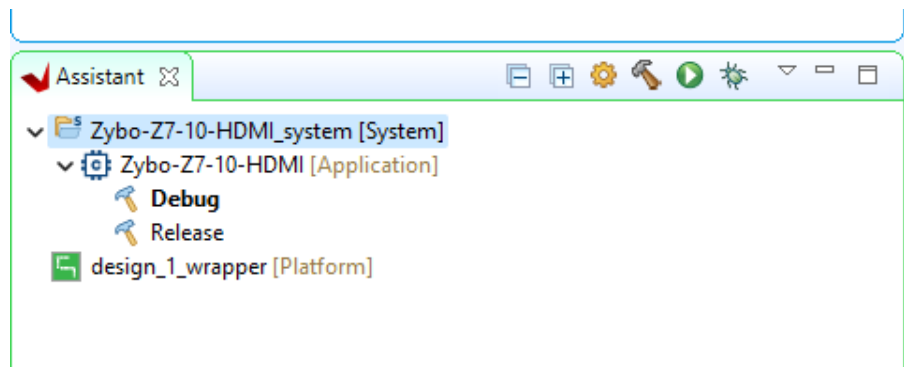


6) Once Vitis is open select “Import Project” from the bottom left side of the User Interface, choose “Vitis project exported .zip file” and select vitis\_492\_final.ide.zip (downloaded in step 3)

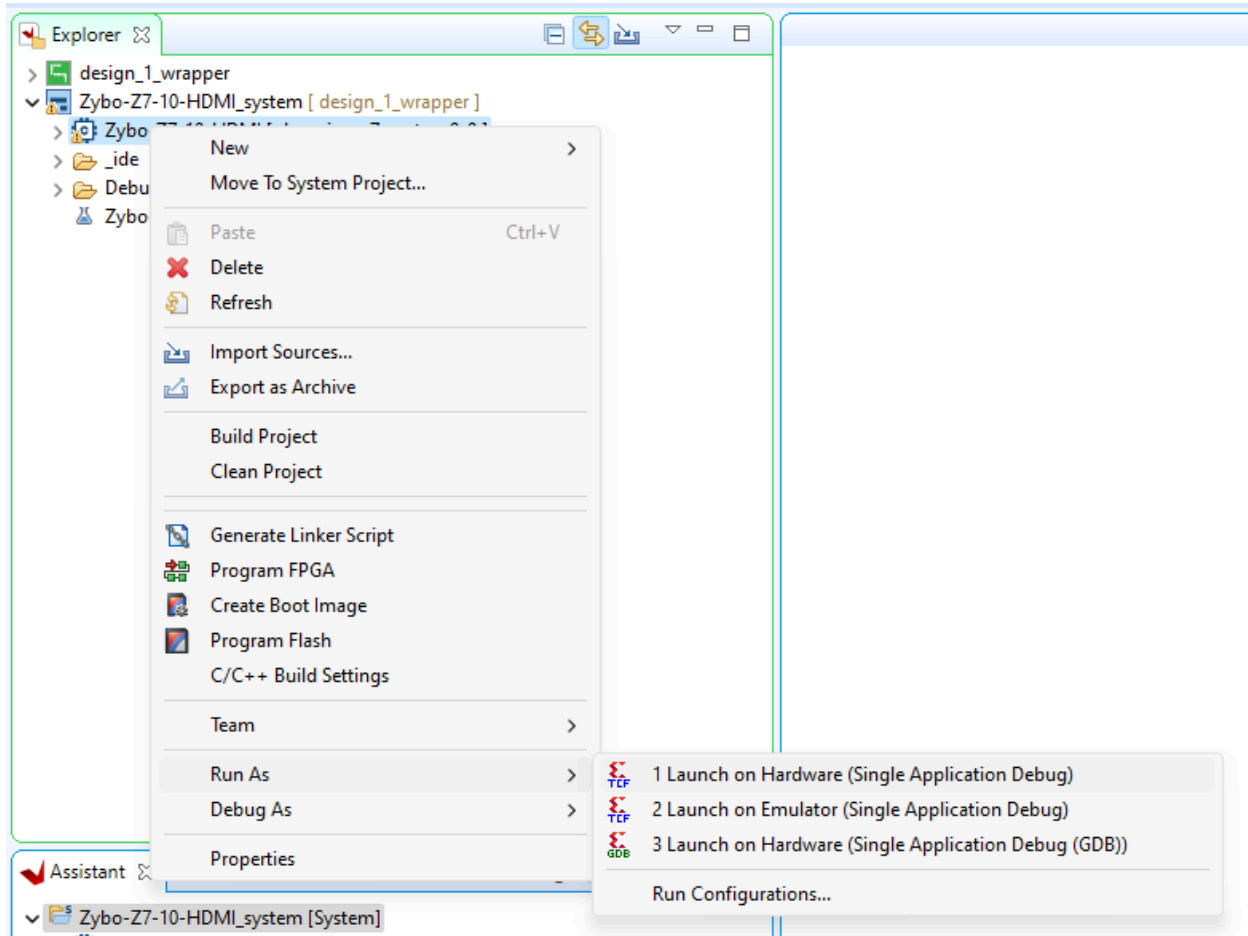
7) Ensure that all projects are selected and press finish



8) To build select Zybo-Z7-10\_HDMI\_system [System] and hit the build button



9) Now, to launch this vitis application on hardware, ensure the Zybo Z7-10 is fully set up to the PC, the HDMI interfaces, powered on, and the power adapter is plugged in. Then, select the Application project from Vitis with a right click, navigate to Run As, then select “Launch on Hardware (Single Application Debug)”



10) The monitor hooked up to the HDMI TX port of the Zybo Z7-10 should now be displaying the video after it has been compressed and decompressed.

11) A majority of the work we did for this project was on the hardware side. The Vitis release already contains the hardware platform we developed, but if anyone would like to edit or inspect our hardware project, it can also be found at <https://git.ece.iastate.edu/sd/sdmay24-12>. To launch the project, select the .xpr file after unzipping the archived project. The steps for updating the hardware platform file from Vivado to a Vitis project can be found here if interested: [Vitis-In-Depth-Tutorial/Vitis Platform Creation/Introduction/02-Edge-AI-ZCU104/step1.md](https://github.com/Xilinx/Vitis-In-Depth-Tutorial/blob/master/02-Edge-AI-ZCU104/step1.md) at 2020.1 · Xilinx/Vitis-In-Depth-Tutorial · GitHub

## Appendix 2 – Design Iterations

To dive deeper into the iterations mentioned earlier, we started the first semester researching potential compression algorithms that would match project requirements while matching feasibility concerning the amount of time we have. The result was to implement an LZW encoder and decoder in C/Java and see which would be the most efficient to use as an IP in the schematic. Early second semester, we started to develop the first few iterations of the encoder in parallel with the production of the HDMI passthrough. The code worked flawlessly with testing, but once we introduced images into the input, the compression size increased the file size rather than reduced. To try and make the data more repetitive, which is where LZW thrives, we tried adding a BWT implementation into the algorithm to rearrange the data into alphanumeric sorted sequences. This addition did not help. Since the data worked well with testing, we found that pulling in bytes of data would always be treated as a signed byte leading to negative ASCII values (they do not exist but were being created), messing up the encodings. This issue was a challenge we could not overcome on top of the runtime of the bitstream taking longer than required.

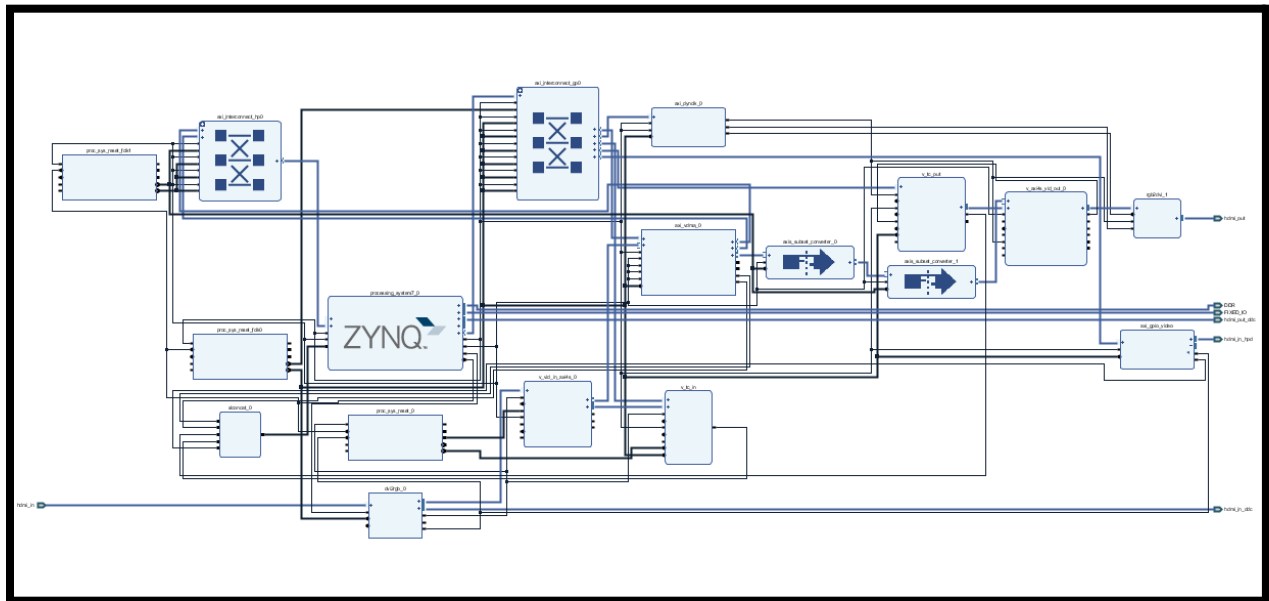
Due to time constraints, we stopped trying to develop an IP with a specific algorithm. We decided to look for a premade custom IP to add to the project but could not find one that worked on our FPGA and fit within our allotted budget (all were over \$20,000 and did not allow a free evaluation period). Next, we set out to implement an open-source Vitis HLS compression scheme by packaging it into an IP Core but faced challenges of FPGA space and input/output compatibility. We also tried implementing CCSDS123 compression into our system. We got the Internal Logic Analyzer from Vivado to show that the CCSDS123 IP was outputting an AXI-Stream. However, there was little to no documentation on how to decompress CCSDS123. Shifting plans again, we decided to create our own IP with simple compression and decompression. We implemented a compression algorithm with a quantization scheme that would convert RGB 8:8:8 to RGB: 5:6:5. Then we decompressed those values by adding bits back. We then used our one software program to optimize and quantify the values of the bits we add back. We decided to add back the middle value of the 2 or 3 missing bits. This approach means that if 3 bits were removed when compressing, then to decompress, we will append the bits “100” to the end of the 5 bits to bring it back to 8 bits. In binary, “100” is the middle value of “111” and “000”, which provides a more neutral approximation.

## Appendix 3 – Other Considerations

As mentioned in Appendix 2, we considered many other strategies to implement compression and decompression other than the strategy we used. These strategies included creating our own IP by writing a VHDL program of a compression algorithm such as LZW, using a pre-made IP from the XILINX marketplace, and implementing an open-source compression scheme into a packaged IP.

## Appendix 4 – Code

### Vivado Block Design



## Python Code for Testing Average Error Per Pixel:

```

import rawpy

if __name__ == "__main__":
    check0 = check1 = check2 = check3 = check4 = check5 = check6 = check7 =
totalVarR = totalVarG = totalVarB = 0
    # Open the file for image to test
    with rawpy.imread('C:/cygwin64/home/benme/491_Project/Java/Image3.NEF')
as raw:
    # Process image into an object
    rgb = raw.postprocess()
    height, width, _ = rgb.shape
    # Prints width, height, and color channels
    print(rgb.shape)
    print("Total Pixel Count: " + str(height*width))

    for pxlw in range(width):
        for pxlh in range(height):
            # Extract rgb values from pixel map twice
            r, g, b = rgb[pxlh, pxlw]
            rt, gt, bt = rgb[pxlh, pxlw]
            # Alter the numbers like compression algorithm
            rt = ((rt >> 3) << 3) + 4
            gt = ((gt >> 2) << 2) + 2
            bt = ((bt >> 3) << 3) + 4
            # Add to variance to total
            totalVarR += (abs(r - rt)) / 256
            totalVarG += (abs(g - gt)) / 256
            totalVarB += (abs(b - bt)) / 256
    # Divide variance of each variable by px count for average
    totalVarR = totalVarR / (width * height)
    totalVarG = totalVarG / (width * height)
    totalVarB = totalVarB / (width * height)
    print("Total Variance Red: " + str(totalVarR * 100))
    print("Total Variance Green: " + str(totalVarG * 100))
    print("Total Variance Blue: " + str(totalVarB * 100))
    print("Total Variance per Pixel: " + str(((totalVarR + totalVarG +
totalVarB) / 3) * 100))

```

### Java Code for the LZW Compression Algorithm from the Initial Design:

```
private static void LZW_Encoding() {
    int keyCount = 0;
    HashMap<String, Integer> enc = new HashMap<>(MAX_ENTRIES);
    // Populate the encryption HashMap with the 256 ASCII symbols
    for(; keyCount < ASCII_ENTRIES; keyCount++) enc.put("" + (char)keyCount,
keyCount);
    // Open a file and initialize bitstream variables
    try (BufferedInputStream bufferedInputStream = new
BufferedInputStream(new FileInputStream("./Java/Metal.dng"))) {
        FileWriter fw = new FileWriter("./Java/compressed.txt");
        BufferedWriter bw = new BufferedWriter(fw);
        int bytesRead;
        byte[] byteBuff = new byte[4096];
        Charset charset = Charset.forName("UTF-8");
        CharsetDecoder decoder = charset.newDecoder();
        // While bitstream is reading
        while ((bytesRead = bufferedInputStream.read(byteBuff)) != -1) {
            String asc = "";
            // Initialize asc
            for (int i = 0; i < byteBuff.length; i++) {
                if (byteBuff[i] >= 0) asc += (char)(byteBuff[i]);
                else asc += (char)(256 + (int)(byteBuff[i]));
            }

            int iter = 1;
            String P = "" + asc.charAt(0);
            String encryption = "";

            while (iter < asc.length()) {
                char C = asc.charAt(iter);
                // If next char is in the map add to pattern
                if (enc.containsKey(P + C)) {
                    P += C;
                }
            }
        }
    }
}
```

```

        } else {
            int[] code = new int[2];
            code[0] = ((enc.get(P)) >> 8 ) & 0xFF;
            code[1] = ( enc.get(P)          & 0xFF);
            // New char was not in known patterns so check if entries
are full
            if (enc.get(P) < ASCII_ENTRIES) { encryption += "0" +
(char)code[1]; }

            // HashMap is full so use the current possible code P
            else { encryption += (char)code[0] + (char)code[1]; }
            // Add new key and iterate keyCount
            if (keyCount < MAX_ENTRIES) {
                enc.put(P + C, keyCount);
                keyCount++;
            }
            // Reset the pattern for next iteration
            P = "" + C;
        }
        // Handling the last case of a stream
        if (iter == asc.length() - 1) {
            int[] code = new int[2];
            code[0] = ((enc.get(P)) >> 8 ) & 0xFF;
            code[1] = ( enc.get(P)          & 0xFF);
            if (enc.get(P) < ASCII_ENTRIES) { encryption += "0" +
(char)code[1]; }

            else { encryption += (char)code[0] + (char)code[1]; }
        }
        iter++;
    }
    bw.write(encryption);
}
bw.close();
} catch (IOException e) {
    e.printStackTrace();
}
}

private static void LZW_Decoding() {
    // Initialize variables for the HashMap

```



```

int keyCount = 0;
HashMap<Integer, String> enc = new HashMap<>(MAX_ENTRIES);
for(; keyCount < ASCII_ENTRIES; keyCount++) enc.put(keyCount, "" +
(char)keyCount);
// Open the compressed file and initialize stream
try (BufferedInputStream bufferedInputStream = new
BufferedInputStream(new FileInputStream("./Java/compressed.txt"))) {
    FileWriter fw = new FileWriter("./newout.txt");
    BufferedWriter bw = new BufferedWriter(fw);

    int bytesRead;
    byte[] buffer = new byte[4096];

    while ((bytesRead = bufferedInputStream.read(buffer)) != -1) {
        String asc = "";
        // Initailize variables for handling incoming bytes
        for (int i = 0; i < buffer.length; i++) asc += (char)(buffer[i] &
0xFF);

        int[] code = { asc.charAt(0), asc.charAt(1) };
        int iter = 2;
        int OLD = ((code[0] << 8) & 0xFF) + (code[1] & 0xFF);
        int NEW;

        String encryption = "" + enc.get(OLD);
        String S = "";
        char C = ' ';

        while (iter < asc.length()) {
            // Start looking to find the decoding sequence
            code[0] = asc.charAt(iter);
            code[1] = asc.charAt(iter + 1);
            NEW = ((code[0] << 8) & 0xFF) + (code[1] & 0xFF);
            // Add key if new sequence
            if (!enc.containsKey(NEW)) {
                S = enc.get(OLD) + C;
            } else {
                // Else grab sequence
                S = enc.get(NEW);
            }
        }
    }
}

```

```
        encryption += S;  
        C = S.charAt(0);  
        enc.put(keyCount, enc.get(OLD) + C);  
        keyCount++;  
        OLD = NEW;  
        iter += 2;  
    }  
    bw.write(encryption);  
}  
bw.close();  
} catch (IOException e) {  
    e.printStackTrace();  
}  
}
```