ECE532: Digital Systems Design

Final Project Report

Brytni Richards

Dylan Vogel

Lorna (Xi) Lan

2021-04-14

Table of Contents

List of Figures

List of Tables

1 Overview

1.1 Motivation

Our group aimed to prototype a 3D hyperspectral imaging compression and decompression pipeline for a small satellite imaging payload. However, hyperspectral imaging applications often require a large volume of data, which is problematic for the limited memory and communication downlink requirements fitted on a small satellite. Therefore, it is important to compress the images to reduce system requirements.

In real satellite applications, a ground station computer will be receiving compressed images taken from space. In this project, a PC would be used to mimic a ground station, where it monitors the FPGA behavior, receives statistics about compression ratio and bandwidth, and decompresses the image. Ideally, it should have user-friendly interfaces to display the decompressed image and satellite status.

Given these objectives, the project would include two FPGAs and one host computer. One FPGA would be performing compression on a live stream of image data, and transferring it over the network to another FPGA. The second FPGA would decompress the received image and send it to the host computer. There are three main tasks needed: compression IP block development, interaction with a bulk storage element, and a data transfer pipeline. This corresponds to DCT custom IP block design, SD card controller development, and TCP (or other serial) protocols handled on FPGAs.

1.2 Goals

The goal of the project was to implement a compression algorithm on a FPGA with bulk storage and capability of wireless communication. Based on this goal, we have the following functional requirements:

1. Shall be able to compress images
2. Shall be able to decompress images

3. Shall contain a custom IP block for algorithm acceleration
4. Shall be capable of inter-FPGA communication for transfer of compressed image data
5. Shall be capable of FPGA to PC communication
6. Shall be capable of collecting statistics on the performance of the algorithm
7. Shall be capable of displaying collected statistics
8. Shall communicate over the network for transfer of data

Requirement 4 of inter-FPGA communication stems from the unique learning environment we had this year, where we could only access the FPGAs via computers within the school network. Since all FPGAs were already connected to the network, our projects were required to have at least one FPGA-to-FPGA communication link. In our case, we proposed to have one FPGA being the compressor and the other the decompressor. This was however, de-scoped later as the compression block integration took longer than expected. The second FPGA was reframed to be a TCP mirror server, where it diverts the first FPGA's packets to the host PC without change.

For the completion status of each of these requirements, please refer to Table 1 in the Outcome section. Similarly, more detailed feature descriptions are listed in Table 2.

1.3 Block Diagrams

Given the objectives and functional requirements described above, Figure 1 represents our final system block diagram. Figure 2 shows the detailed breakdown of compression FPGA, with highlight on the custom DCT block. For the overall data transfer pipeline, USB UART was selected as a stand-in serial protocol from the host PC to the FPGA for its convenience as a point-to-point protocol. TCP was chosen as the wireless communication protocol stand-in due to its popularity and maturity.

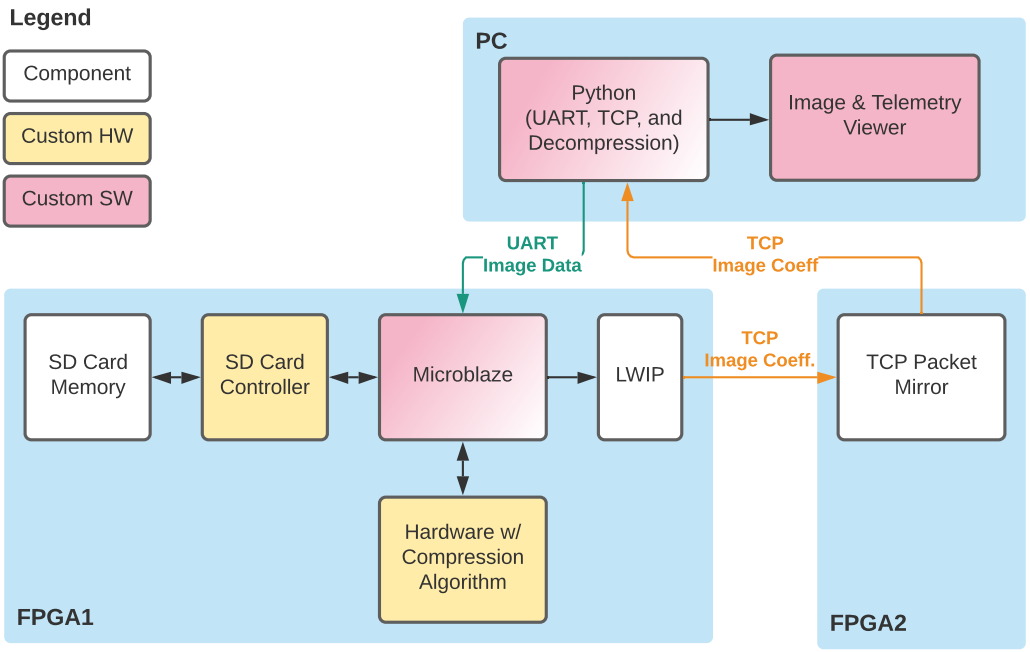


Figure 1: Final System Block Diagram

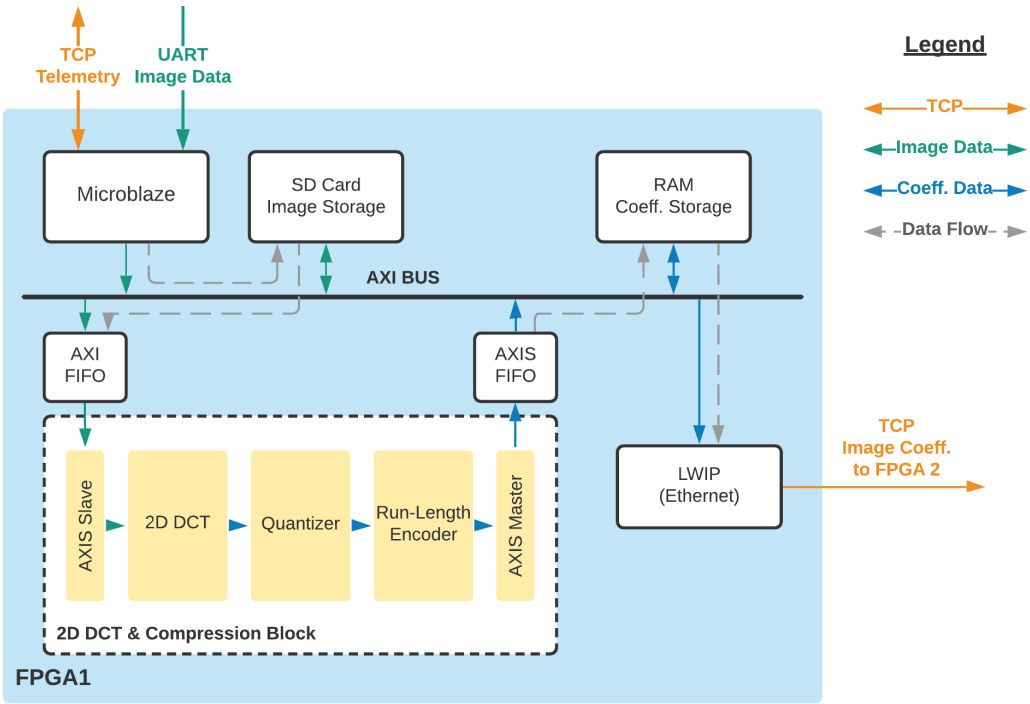


Figure 2: Detailed block diagram for the compression FPGA

The hypersectural image dataset we used for our project was from Columbia University. Each image was 512x512 KB with 16 bit-depth. For simplicity of the compression algorithm, each image pixel was read as 8-bit and transferred around in an 8x8 image block. A Python program on Host PC disassembled an image into blocks, then passed 7 of them at once over UART, which then wrote to SD card in a chunk of 488 bytes. Later the program read from SD card, passed each 8x8 image block to the DCT block, then read back the compressed coefficients calculated. Each 8x8 image block's compressed coefficients could at most be 130 bytes in length. These coefficients were then assembled into valid TCP packets by adding a data type and length header, sent to the second FPGA, then back to Host PC for decompression and display.

1.4 IP Overview

The full project was implemented in Vivado 2018.3 and Xilinx's Nexys 4 DDR FPGA board. The complete IP block diagram for our hardware on the compression FPGA (FPGA1) is shown in Figure 3. It can be divided into four parts based on their functions. More details on the custom blocks will be provided in the later sections. For the second FPGA functioning as a TCP mirror server (FPGA2), its block diagram is largely the same as the first, minus the compression block and SD card controller. All IPs used in this project is listed below:

1. Central Processor: MicroBlaze and its supporting units
    (a) MicroBlaze
    (b) MicroBlaze local memory
    (c) MicroBlaze Debug Module
    (d) MicroBlaze Concat
    (e) MicroBlaze AXI Interrupt Controller
    (f) Clocking Wizard
    (g) AXI Timer
    (h) AXI Interconnect
    (i) Processor System Reset (one for MicroBlaze, one for MIG7 DDR2 memory)

2. Storage unit: memories and SD card
    (a) Memory Interface Generator (MIG 7 Series)
    (b) AXI SmartConnect
    (c) SD card controller (sd_control_ram_v1.0) [Custom]
3. Communications
    (a) AXI UartLite
    (b) AXI EthernetLite
    (c) Ethernet PHY MII to Reduced MII
4. Compression Block
    (a) Custom DCT block (custom_dct_axis_v2_4) [Custom]
    (b) AXI Stream FIFO
    (c) System ILA

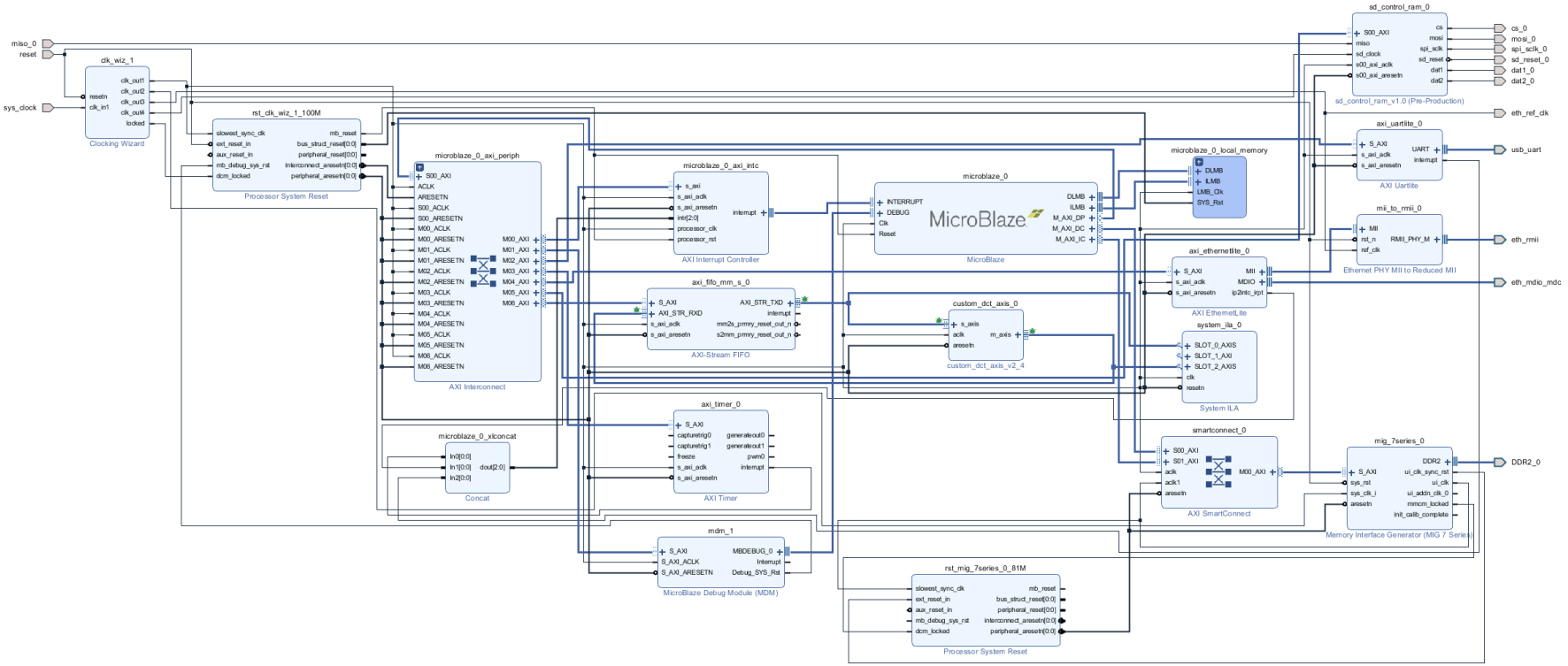


Figure 3: Final IP block diagram for the compression FPGA (FPGA1)

2 Outcome

2.1 Requirements and Criteria

The success of our project was judged based on the requirements and criteria outlined in the project proposal. These requirements have also been outlined in the tables below.

Requirement	**Status**
Shall be able to compress images	complete in hardware
Shall be able to decompress images	complete in software
Shall contain a custom IP block for algorithm acceleration	complete
Shall be capable of inter-FPGA communication for transfer of compressed image data	complete
Shall be capable of FPGA to PC communication	complete
Shall be capable of collecting statistics on the performance of the algorithm	complete
Performance monitoring on DESL Desktop	complete
Shall be capable of displaying collected statistics	complete
Shall communicate over the network for transfer of data	complete

Table 1: Functional Requirements

Baseline Requirements	**Status**
Discrete Cosine Transform (DCT) compression/decompression on 2D images	complete
MicroBlaze handling FPGA to FPGA networking with TCP/UDP and LWIP	complete
Desktop to FPGA network via TCP/UDP from Python script	complete
Desktop to FPGA data transfer using UART + MicroBlaze + Python	complete
Image transfer between MicroBlaze and SD card	complete
Performance monitoring on DESL Desktop	complete
Visualization of performance statistics in PC terminal	complete
Exceed Requirement	**Status**
Discrete Wavelet Transform (DWT) compression/decompression on 3D images	de-scoped
MicroBlaze handling FPGA to FPGA networking with raw IP/MAC packets	de-scoped
Desktop to FPGA network via raw IP/MAC packet and python	de-scoped
Custom SD card IP block	de-scoped
Performance monitoring in hardware	de-scoped
Visualization of performance statistics in fAnCy Ui (Python/Grafana)	complete

Table 2: Feature Requirements

Requirements	**Status**
Shall be able to achieve a compression ratio of greater than 1.5 for satellite imagery	complete
Shall be able to decompress images with minimal visual artefacts	partially complete
Shall be capable of FPGA to FPGA communication at a minimum of 1 Mbps	partially complete*
Shall be capable of FPGA to PC communication at a minimum of 1 Mbps	partially complete*
Shall be capable of measuring average compression ratio	incomplete
Shall be capable of measuring average network speed	incomplete
Shall be capable of measuring compression and decompression bandwidth	partially complete
Shall be capable of displaying the above statistics on a UI	complete

Table 3: Acceptance Criteria

Please note that in Table 3, the partially complete* for the networking component means that the communication was successful but the speed was not to-spec. However, our original proposition was aggressive. The TCP networking speed during the final demo test transferred 137,216 bytes of packet data in 19 seconds, which translates to a speed of 57.775 Kbps. While this seemed slow compared to our initial estimate, it was fast enough for the project operation.

2.2 System Changes

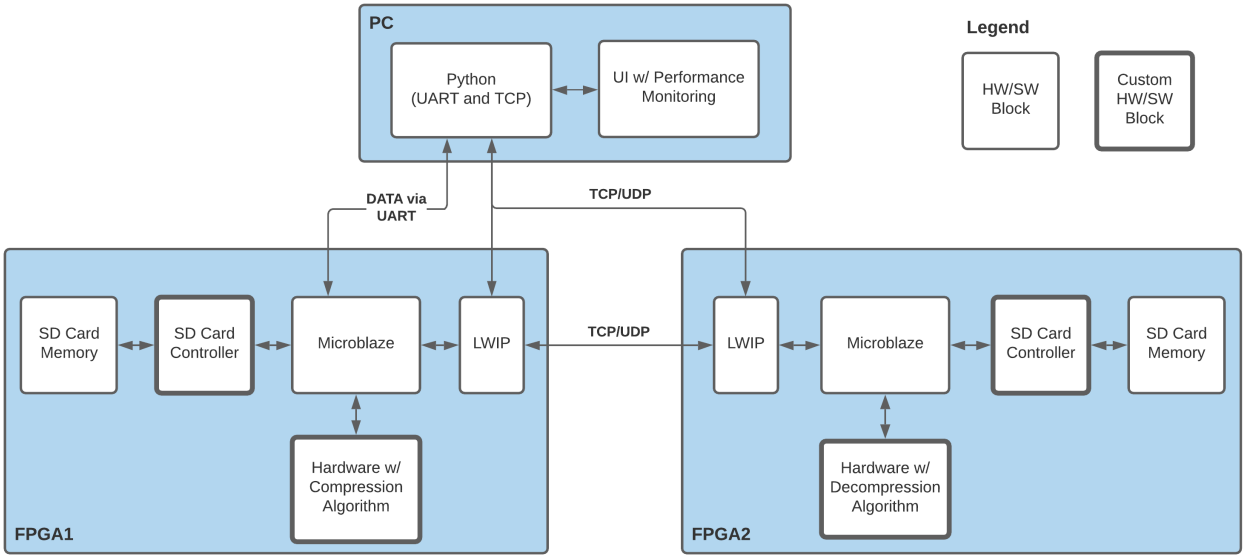


Figure 4: Proposed Block Diagram.

Based on the requirements outlined in the previous section, the proposed system is as seen in Figure 4. In the proposed system, the steps are as follows:

1. The set-up is initialized.
   (a) Image on the PC is transferred to the MicroBlaze through UART
   (b) MicroBlaze stores the image into the SD card
2. The stored image data is read from the SD card by MicroBlaze
3. MicroBlaze passes 8x8 image block data to the DCT compression IP
4. The DCT IP block returns the compressed coefficients to MicroBlaze
5. Compressed coefficients are sent to a second FPGA through TCP
6. Second FPGA MicroBlaze sends the compressed coefficients to the DCT decompression IP
7. DCT IP returns decompressed image data

8. decompressed image data is stored in the SD Card

9. Decompressed image on SD card is transferred through TCP to the PC for statistic purposes

The final system demonstrated in the final demo was shown in Figure 1. The second FPGA was simplified due to time constraints. Instead, it became a mirror server that mirrored the TCP packets it received to the PC TCP client. The PC client was then responsible for decompressing the image and showing any telemetry data.

2.3 Results

According to our requirements, we were successful in achieving many of our original goals. We met all of our functional requirement goals along with our baseline requirements for the features. While many of the exceed feature requirements were de-scoped from our final product due to time constraints, we were still able to achieve one of them.

Some of the acceptance criteria were only partially complete due to some bugs in the integrated system. However, much of the functionality behind the acceptance criteria worked as expected either individually or with integration. Figure 5 shows the decompressed image using the compression coefficients retrieved from the DCT IP and decompressed using software. From the image, it can be seen that there are some minor artefacts, which led us to think that the second acceptance criterion was only partially complete.

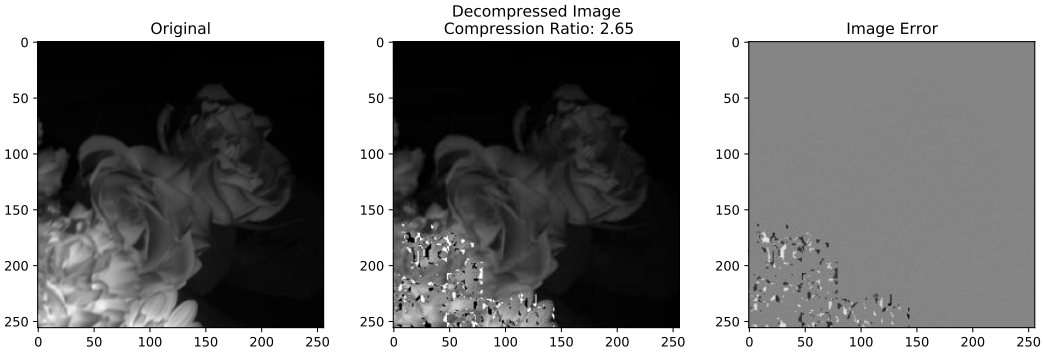


Figure 5: Image compression analysis

Our final system was fairly similar to our proposed design. Although the second FPGA was

simplified, its features complemented the first FPGA and this change did not drastically retract the difficulty or complexity of the project according to the point difficulty allocations for different features. In conclusion, our team was satisfied with the final demo result and learnt many concepts from the experience.

2.4 Improvements

A couple of components in our system could be improved. The AXI Stream FIFO that communicates with the DCT compression IP needed to be correctly integrated with the rest of the system. Currently, the AXI Stream FIFO does not read data it is given, so the MicroBlaze is unable to communicate with the DCT compression block. An improvement would be to fix this interaction so that MicroBlaze could obtain the correct compression coefficients.

Another issue was that the SD card required a couple of seconds to process each write, which was longer than the expected time of 0.5ms [2]. This may be due to the SPI clock speed, which could have potentially have been faster. Alternatively, the native SD card protocol SDIO could've been used instead, which is a bit faster than SPI [3].

As mentioned before, many of our exceed feature goals were not met. A future improvement would be to implement these features to improve our system. Also, instead of the second FPGA being a mirror server, it could be designed to follow the original system and decompress the DCT coefficients in hardware.

Since this project was aimed for use on a small satellite payload, tests could be done to assess whether the system is qualified for space operations. Moreover, techniques that mitigate the effects found in space such as radiation that may adversely affect the system could be incorporated.

3 Project Schedule

This section outlines the initial and accomplished project schedule. A table tracking the individual milestone targets and completed work for each member is given in Tables 4, 5, and 6 for Brytni, Dylan, and Lorna, respectively.

Brytni's proposed and actual accomplishments were similar. There was one unexpected challenge between Milestones 3 and 4 due to the unusual SPI requirements for the SD card interface. However, Milestone 4 was two weeks apart from Milestone 3 and gave plenty of time to sort out the bugs and catch up.

Dylan's proposed and actual accomplishments were similar, although difficulties were encountered towards the end of the project when trying to integrate the DCT AXI Stream interface with the Xilinx AXI DMA and AXI Stream FIFO blocks. In hindsight, it may have been wiser to stick with the inefficient AXI-Lite interface and focus on improving other aspects of the system. This likely would have allowed the project to stay on schedule.

Lorna's proposed and actual accomplishments were close other than implementing a RAW IP Ethernet protocol. She was on track for the first 3 milestones but was a bit behind for milestone 4. The team's focus and need for milestone 5 and 6 shifted from individual development to integration, so the custom networking protocol was de-scoped. In hindsight, the entire data pipeline might have been better to be relying on one protocol, such as TCP only (and develop the RAW IP protocol a lot earlier), rather than a mix of UART and TCP. While the latter might be more common in a realistic project, it takes more time to deal with the integration and conflicts between two protocols.

Milestone	Proposed Accomplishment	Actually Accomplished
1	SD card implementation and protocol research	No change
2	Implement physical layer (SDIO/SPI)	Completed Milestone 2 and 3; decided to use AXI QSPI IP and tested the SPI protocol
3	Create tests for SDIO/SPI	Worked on Milestone 4; SD card initialization was written (along with the read and write) but the SD card did not return the expected values; bugs were present in the initialization sequence
4	Finish initialization for SD card and start on SD card read and write functionality	Completed Milestone 4 and half of Milestone 5; due to unexpected behaviour required from SPI, quit using the AXI QSPI and instead used controller from [4]. SD card initialization, read and write functionality were completed and returned the expected values
5	Finish SD card read/write and start on integration to store images	No change; integrated SD card controller into the system with Lorna for individual read and writes
6	Finish SD card integration and assist in UI and design	No change; Completed SD card controller integration with Lorna for image storing and reading
Final Demo	Finish visualization	No change; Completed Python demo and visualization of decompressed image

Table 4: Project Schedule for Brytni Richards, who was mainly responsible for the SD card controller and demo visualization

Milestone	Proposed Accomplishment	Actually Accomplished
1	Research and planning for DCT	No change; Downloaded several research papers describing DCT hardware design
2	DCT algorithm running in test bench with arrays	Completed algorithm in Python and tested with 2D images. Have started but not finished DCT building in hardware
3	Initial performance visualization demo and initial integration of DCT block	Completed Milestone 2 and half of Milestone 3; Finished 1D DCT algorithm in test bench and got visualization demo running with packets. Defined team packet format
4	MicroBlaze interaction with DCT block finished, can input and output images as part of the pipeline	No change; Completed transpose buffer between 1D DCT stages, implemented AXI-Lite interface, integrated with MicroBlaze, and developed test bench to show interaction. Some improvements could be made to the interface
5	Hardware performance monitoring implemented for DCT, performance data being sent back to Python UI	De-scoped hardware monitoring, and passed telemetry data on to Lorna who was handling the main MicroBlaze code. Changed DCT AXI-Lite interface to AXI Stream, finished quantization and run-length encoding blocks on end of DCT stage
6	Flesh out Python UI and finish systems integration	De-scoped Python UI and passed off to Brytni as I worked to finish systems integration. Heavy debugging of the new AXI Stream interface and Xilinx AXI DMA as I worked to get it running with MicroBlaze. Created full simulation test bench to help debug
Final Demo	Debug the DCT interaction and close out integration	Pivoted to Xilinx AXI Stream FIFO at the last minute after being unable to get the AXI DMA S2MM side to work properly over multiple iterations. Could not get AXI Stream FIFO working in time, pivoted to running DCT decompression in Python and pulled coefficients from DCT hardware simulation

Table 5: Project Schedule for Dylan Vogel, who was mainly responsible for building the DCT hardware blocks

Milestone	Proposed Accomplishment	Actually Accomplished
1	Outline the steps/blocks need to do to achieve full networking picture; Start looking at ways on Desktop to FPGA data transferring with UART	No change; started Python script to transfer image block over UART
2	Implement Host PC to FPGA data transfer using UART + MicroBlaze + Python with floating arrays	No change; finished the UART pipeline (Host PC script and MicroBlaze C code)
3	Finish Host PC to FPGA data transfer with UART + MicroBlaze + Python (can transfer images); Get MicroBlaze handling FPGA to FPGA networking with TCP	No change; started the DDR2 pipeline, preliminary integration with SD card
4	Finish FPGA to FPGA networking with raw IP/MAC packets (can transfer image)	Did not start raw IP implementation, was fixing UART + TCP integration issue (interrupt conflicts)
5	Start working on MicroBlaze handling FPGA to FPGA networking with raw IP/MAC packet and python; Help with performance data transfer ot visualization if needed	De-scoped raw IP implementation, successful integration with SD card, finalizing TCP code
6	Finish MicroBlaze handling FPGA to FPGA networking with raw IP/MAC packets (can transfer images)	Continued integration with data pipeline with SD card and DCT block
Final Demo	Demonstrate the full data pipeline from Host PC to FPGA1 for compression, FPGA1 to FPGA2 and back to Host PC via TCP	Wrote TCP mirror server for FPGA2, demonstrated full data pipeline

Table 6: Project Schedule for Lorna Lan, who was mainly responsible for creating the network interfaces and integrating the project parts.

4 Description of the Blocks

This section provides a detailed description of each IP block in the project. For each block we describe whether we built it from scratch or borrowed from existing IP, how it functions, and an overview of the testing that we performed.

4.1 SD Card Controller

The SD Card Controller IP is responsible for reading and writing data to the SD card. The possible communication protocols for the SD card are SPI (Serial Peripheral Interface) and SDIO (Secure Digital Input Output). SPI is also used for other peripherals other than SDIO and has a lot of community support and details, therefore we chose to use SPI to communicate with the SD card.

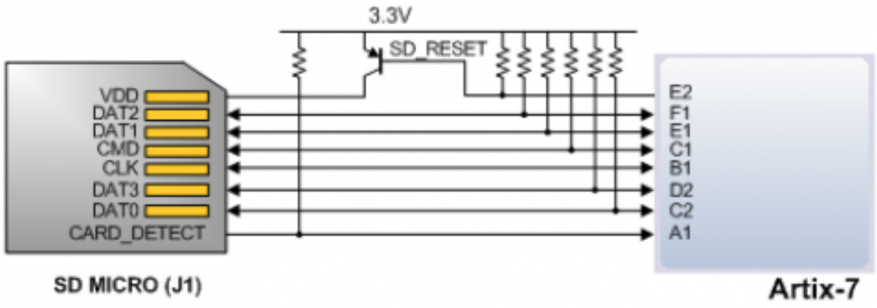


Figure 6: SD Card FPGA Connections [1]

Figure 6 shows the connections between the FPGA and SD card. In order to power the SD card, the E2 pin must be set low. For SPI, the DAT1 and DAT2 lines are set low. DAT0 and CMD are MISO (Master in slave out) and MOSI (Master out slave in) in SPI respectively. Finally, DAT3 is the SPI CS (chip select) [5]. The CLK is clocked at 25MHz, although it could optionally be clocked faster.

Upon powering up, the SD card requires a 1ms delay and 74 SPI clock cycles when the CS is set high [6]. To set up the SD card device for reading and writing, it must complete the initialization sequence outline in Figure 7. After the SD card is in the idle state, the card is ready to accept read and write commands. The CMD and ACMD are specific commands that the FPGA sends to the SD card through SPI.

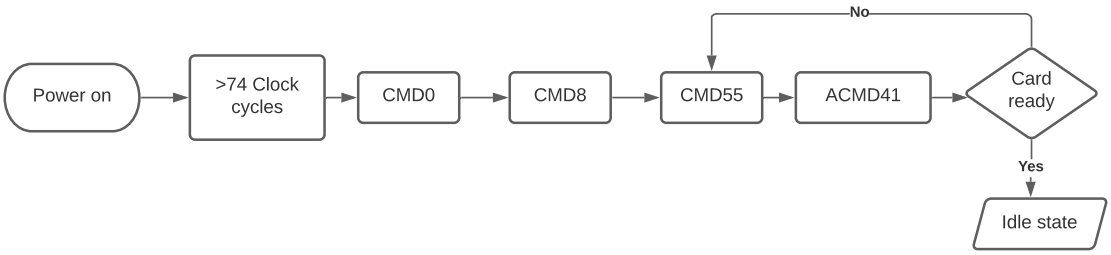


Figure 7: SD Card Initialization sequence

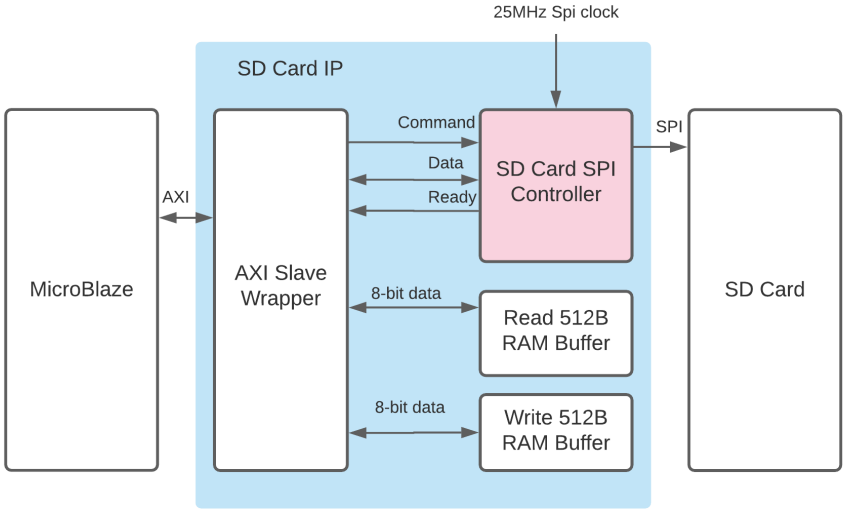


Figure 8: SD Card Controller Block Diagram

Figure 8 shows an overview of the SD card controller IP. The component responsible for the SD card initialization, SPI, and specific SD card commands outlined above was the SD Card SPI Controller; the pink block. This controller was borrowed from [4]. Some modifications were made to this borrowed controller so that it would work with the 2nd version of SD cards, namely the command parameters were changed. CMD8 was added after CMD0, and CMD55 was added before ACMD41 [4, 7]. In addition, two new registers were added to the SD card SPI controller to specify which read/write RAM address to read/write data to/from.

The RAM read and write buffers are required for time synchronization. The SD Card write and read commands operate on a block of 512 bytes [6]. In addition, the read and write SD card operations are block addressable (not byte addressed). Therefore, when a write occurs, the IP will obtain the write

Name	**Address**	**R/W**	**b31-b8**	**b7-b6**	**b5-b4**	**b2**	**b1**	**b0**
Command	0x0	W				SD read	SD write	SD reset
Status	0x1	R						R/W ready
Block Address	0x2	W	SD Block Address to R/W					
Debug	0x3	R			SD state			
Write data	0x4	W		Write RAM data				
Read data	0x5	R		Read RAM data				
RAM Command	0x6	W				RAM read	RAM write	unused
Write RAM Address	0x7	W	RAM address to write to					
Read RAM Address	0x8	W	RAM address to read from					

Table 7: SD card controller IP register mapping, where b, R and W represent bit, read and write respectively.

data from the Write RAM buffer, then store that information on the SD card. For reads, the IP will store the data read back into the Read RAM buffer, which can then be obtained from the AXI Slave Wrapper.

To communicate with MicroBlaze, an AXI Slave interface was included. This code used much of the default AXI Slave code generated from Vivado with modifications made for the RAM and SD Card SPI Controller. The communication is treated similar to a memory mapped IO, where certain registers in the AXI Slave wrapper will command the SD card to perform different operations. This specification is user-defined. A description of the different AXI registers, bits and their commands are in Table 7. Note that the command registers are active high. Also, the SD state is the state of the finite state machine on the SD Card SPI Controller and specifies which command it is currently executing. SD card testing with MicroBlaze integration is located in `sd_card.c`.

4.2 DCT Block

This section outlines the design of the DCT compression block. While the decompression side was also initially scoped, it was de-scoped during development due to challenges integrating the compression side. All the DCT modules were written from scratch in Verilog with inspiration from [8]. The source code for the DCT block can be found in our GitHub.

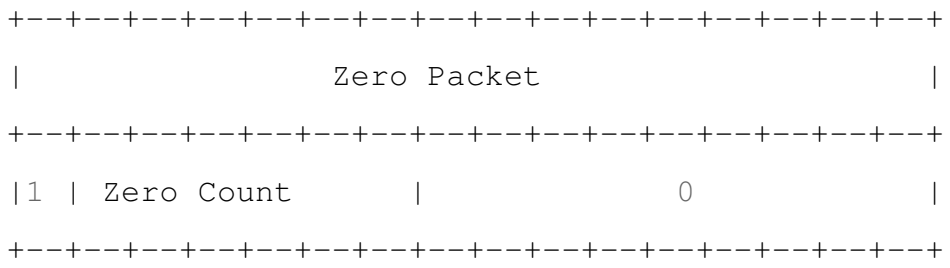
A block diagram showing the DCT block interaction with the rest of FPGA1 was shown in Figure 2. At the module level, we have implemented two 1D DCT stages and a transpose buffer (both inside `2D DCT`), a quantizer stage, a zig-zag encoder (not shown), a run length encoder, and an AXI Stream master and slave interface. The `dct_main.v` module is responsible for wiring all of the DCT modules together, and is wrapped by the `custom_dct_axis.v` module to provide AXI Stream support.

The design also uses two Xilinx FIFOs v13.2 for buffering and a Xilinx AXI Stream FIFO v4.2 for high-level interfacing to MicroBlaze. At the time of writing, there was an issue interfacing from MicroBlaze to the Xilinx AXI Stream FIFO, which prevented the correct coefficients from being pulled from the DCT block. To the best of our understanding, the DCT AXI Stream interface followed all protocol requirements when debugged on an ILA or in simulation. Therefore, we believed that the issue was with the C code that reads and writes from the FIFO. Potential future users should make note of this limitation in our implementation. If you happen to discover the bug, or if the vanilla block works out of the box for you, feel free to send an email to `dylan.vogel (at) mail.utoronto.ca` or leave a comment on the GitHub repository.

4.2.1 AXI Stream Interface

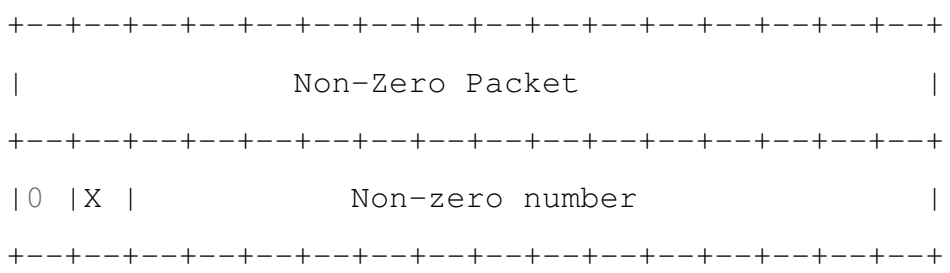
The AXI Stream interface assumes that you have a 16-bit data bus transferring the first pixel in the lower 8-bits and the second pixel in the upper eight bits. This was done because the DCT modules take two inputs per cycle. The output is a 32-bit wide bus, with the lower 16 bits being the first encoded coefficient and the upper 16 bits being the second. The total latency is 85 clock cycles from input to output, assuming that an entire 64-pixel block is written sequentially. The run-length encoding scheme is as follows:

Zero Packets



Bit 15 being set to 1 indicates that the packet is zero. Bits [14:9] indicate the number of zeros, and a value of 0b000000 indicates that there are 64 zeros (the value wraps around). The lower 9 bits [8:0] are currently unused and set to zero. A Huffman encoding scheme would pack these coefficients a lot more tightly, but would be a big step up in terms of complexity for relatively small gains.

Non-Zero Packets



Bit 15 being set to 0 indicates that the packet is nonzero. Bit 14 is currently don't-care (at least in our 14-bit coefficient implementation) and bits [13:0] are the signed coefficient value. In this case, bit 13 is the sign bit and requires appropriate conversion on the decompression side. The end of a particular `packet` is indicated by an end-of-field (EOF) value of all 1's.

All DCT modules process two inputs per clock cycle and will also transmit two outputs per clock cycle once the result is ready. All blocks have some form of input valid signal to indicate when the two input values are valid and can be latched. All blocks also have some form of output valid signal. There is no handshaking, so the downstream module must be able to either process two inputs per clock cycle or contain sufficient buffer size. This design was chosen to ensure that the DCT algorithm operated absolutely as fast as possible, following some initial latency. The total latency is roughly 85

clock cycles from input to output, assuming that an entire 64-pixel block has been input.

4.2.2 DCT Main

The DCT main module connects together the 1D DCT row stage, the transpose double buffer, the 1D DCT column stage, the quantization stage, the zig-zag encoder stage, and the run-length encoder stage. Like the other DCT modules, DCT Main takes two inputs per clock cycles when the input valid signal is high, and outputs two valid half words when the output sync signal is high. DCT main has several parameters which can be adjusted, and are propagated downstream to the other modules:

1. `DATA_WIDTH`: Sets the width of the input pixel coefficients (default: 8)
2. `COEFF_WIDTH`: Sets the width of the DCT coefficients (default: 9)[1]
3. `FIRST_STAGE_WIDTH`: Output width of the 1D DCT row stage (default: 21)
4. `SECOND_STAGE_WIDTH`: Output width of the 1D DCT column stage (default: 25)
5. `QUANT_STAGE_WIDTH`: Output width of the quantizer stage (default: 14)
6. `RUNL_STAGE_WIDTH`: Output width of the run-length encoder stage; should be strictly greater than the quantizer stage output (default: 16)

A test bench for DCT Main was also created and can be found in `tb_dct_main.v`. This test bench assumes that a memory file called `dct_test_block.mem` exists in the project and contains 8-bit pixel values. This test bench can be used to simulate inputting an entire 8x8 pixel block into the DCT algorithm and confirming that the correct outputs are returned.

4.2.3 1D DCT Stage

This stage performs a 1D DCT and is located in `dct_stage.v`. It has a latency of four input cycles (8 pixels) before it starts processing data. Each 1D DCT stage uses 4 DSP blocks, for a total of 8 DSPs

[1] Note that the coefficient memory file `dct_coeff.mem` also needs to be regenerated when this parameter is changed.

used for both row and column DCT operations. The derivation for this operation is provided below. The 2D DCT operation can be expressed in matrix form as:

$$D = MXM^T$$

Where D is the encoded image block, M is the DCT coefficients, and X is the input image block. Exploiting separability, there are numerous ways to decompose this into a series of 1D DCT operations. In this case, we exploit the first-order factorization by Woods et al. [9] where the even output values are given by:

$$\begin{bmatrix} F[0] \\ F[2] \\ F[4] \\ F[6] \end{bmatrix} = \begin{bmatrix} c_4 & c_4 & c_4 & c_4 \\ c_2 & c_6 & -c_6 & -c_2 \\ c_4 & -c_4 & -c_4 & c_4 \\ c_6 & -c_2 & c_2 & -c_6 \end{bmatrix} \begin{bmatrix} f[0]+f[7] \\ f[1]+f[6] \\ f[2]+f[5] \\ f[3]+f[4] \end{bmatrix} \tag{1}$$

And the odd output values are given by:

$$\begin{bmatrix} F[1] \\ F[3] \\ F[5] \\ F[7] \end{bmatrix} = \begin{bmatrix} c_1 & c_3 & c_5 & c_7 \\ c_3 & -c_7 & -c_1 & -c_5 \\ c_5 & -c_1 & c_7 & c_3 \\ c_7 & -c_5 & c_3 & -c_1 \end{bmatrix} \begin{bmatrix} f[0]-f[7] \\ f[1]-f[6] \\ f[2]-f[5] \\ f[3]-f[4] \end{bmatrix} \tag{2}$$

Where $f[0-7]$ are the input pixel coefficients along one row of X, $F[0-7]$ are the encoded image values, and c_{1-7} are the DCT coefficients given by

$$c_k = \frac{1}{2}\cos\frac{k\pi}{16}$$

With separability, we can transform the original 2D DCT equation into:

$$D^T = M(MX)^T \tag{3}$$

Hence, we can simply perform a 1D DCT operation on the rows of X, transpose the output, and perform another 1D DCT on the columns. The DCT coefficients are loaded into memory from a `ram_coeff.mem` file, which is generated by our `coeff_gen.py` Python script. This was done to simplify the amount of calculations required in the 1D DCT stage.

The customization parameters for this block are passed down from DCT Main, although it also contains an additional parameter called `INPUT_SHIFT`. This flag determines whether the input value

should be shifted by -127 (1) or not shifted (0) prior to processing. This is useful, for example, if the inputs to the 1D row DCT are unsigned and require input shifting. DCT Main sets these values appropriately for both the row and column DCTs.

The block operates by first latching the input data. Once enough data has been input, the results of addition and summation on the right side of Equations 1 and 2 are stored in `addsub`. For the next four clock cycles, the module multiplies and accumulates each column of the matrix multiplication to generate the encoded output values. The final value is then latched and output at two values per cycle.

A simple test bench for this module has also been created in `tb_dct_stage.v`. This test bench inputs two pixel values up to 8 each cycle and displays the output, although different sets of pixel values can be tested to confirm that the output is correct.

4.2.4 Transpose Buffer

This stage transposes and double buffers the values between the 1D row DCT and the 1D column DCT, as per Equation 3. The module can be found in `two_wide_transpose_buf.v`. Like the other DCT modules, the double buffer takes two values per cycle and outputs two values per cycle when the `rsync` signal is high. The double-buffering is achieved by keeping track of a `wpage` and `rpage` flag which determine which block of memory is being written to or read from. The page is flipped once 64 values are written to memory. No read handshaking is used, and so the downstream block must be able to accept two values per clock cycle whenever `rsync` is high.

A simple test bench for this module has also been created in `tb_two_wide_bufs.v`. The test bench writes 128 values to memory and displays the values coming out. It also has a couple cases where the reset and input valid flags are toggled, to confirm that the correct amount of data is being read out.

4.2.5 Quantization Stage

The quantization stage is where DCT compression becomes lossy, and is located in `quant_stage.v`. This stage simply bit-shifts the input values depending on the current position in the input block and the corresponding quantization coefficient. In this case, we've rounded each JPEG standard quan-

tization coefficient[2] to the nearest factor of 2 in log scale. The rounding is done in Python via `quantization_gen.py` and stored the memory file `quant_coeff.mem`. Again, this was done to simplify the amount of computation in Verilog and to allow us to play around with different interpretations of the standard matrix.

4.2.6 Zig-Zag Stage

The zig-zag stage is also technically a double buffer, and is located in `zig_zag_stage.v`. The input is transposed to undo the tranpose of D in Equation 3. The data is then read out in zig-zag order. The `zigzag_lookup.mem` file stores the LUT for reading out the matrix in zig-zag order, and is generated in Python to save writing it in Verilog. This module can also be tested with the `tb_two_wide_bufs.v` test bench.

4.2.7 Run Length Encoder Stage

The run length encoder is possibly one of the most complicated blocks in the DCT module. Its function is to read the input data and output it according to the encoding scheme defined in Section 4.2.1. However, the difficulty is that each iteration may produce zero, one, two, or three output values. Thus, the module is written as a finite state machine (FSM) which operates on four input flags. These input flags indicate whether we're currently tracking a zero count, whether the first and second input values are zero, and whether we're at the end of a block.

Based on these flags the FSM will perform different actions. At the most basic level, the FSM will read the input values and write between 0-3 values to a write buffer. The FSM will then increment its current pointer to memory. When the FSM is done reading 64 values, it will update the last pointer to point to the largest address in the write buffer and write out an EOF indicator. The module will then read out the encoded data until reaching the last pointer. Thus, for every 64 inputs the module could output between two and 66 output values. The latter case only occurs if every input coefficient is nonzero and the encoder adds two EOF indicators to pad the output to an even number. A Xilinx FIFO is used on the input of the module, since we may end up having to pause the input stream if our read

[2] See here as an example

buffer is longer than 64 values. However, this is unlikely given that the DCT quantization generates a large number of zeros.

A test bench was also developed for the module and is located in `tb_run_length_stage.v`. The test bench reads from a `rl_test_block.mem` file which can be used to exercise different input coefficient scenarios, and displays the encoded output.

4.3 Networking

4.3.1 Serial Communication - UART

The UART protocol in this project was interrupt based on the MicroBlaze processor, which means that instead of looping to wait for a character arrive/send, we have a receive or send callback function to handle the character arrival events. A fixed-length UART buffer is defined. When there are sufficient received bytes filling the buffer, the MicroBlaze will then write the buffer to SD card in a chunk of 7 8x8 blocks (488 bytes) to speed up the overall image transfer process. To facilitate the transfer of data block chucks, after MicroBlaze finished writing to SD card, it would send the handshake character (arbitrarily defined as "!" in this case) back to the Host PC to signal the transfer of the next 7 image blocks. The code snippet below outlines the main functions doing the procedure described.

```
// Receive Callback functions
void RecvHandler(void *CallBackRef, unsigned int EventData)
{
        XUartLite_Recv(&UartLite, ReceiveBufferPtr, 2);
        ReceiveBufferPtr += 2;
        TotalReceivedCount += 2;
}

// UART + SD card operaton
void uart_sd(u32 sd_addr){
    while (1){
        //If we've reached the end of the buffer, write to memory
        if (ReceiveBufferPtr >= (&ReceiveBuffer[0] + UART_BUFFER_SIZE)){
                // copy the buffer over
```

```
                for(int i=0; i<UART_BUFFER_SIZE; i++){
                        BufferCopy[i] = ReceiveBuffer[i];
                        xil_printf("received data %d\n", BufferCopy[i]);
                }
            // xil_printf("Resetting Receive Buffer\n");
            resetBuffer();
            ReceiveBufferPtr = &ReceiveBuffer[0];
            TotalReceivedCount = 0;
            break;
            }
    }
    // SD card write
        sd_write(sd_addr, UART_BUFFER_SIZE, &BufferCopy[0]);
        xil_printf("done writing to SD card\n");
}

// main UART event loop in main.c
for(u32 i=0; i<UART_BLOCK_NUM; i++){
        xil_printf("Waiting to receive UART packets\n");
        uart_sd((u32)(0x00000000 + i));
        xil_printf("\nimage block %d wrote to SD card\n\n", i);

        sleep(1);
        SendBuffer[0] = 33; // ASCII "!"
        XUartLite_Send(&UartLite, &SendBuffer[0],1);
        resetBuffer();
}
```

On the Host PC Python end, there are many existing libraries for handling serial communications. We selected PySerial as the main library for opening serial ports and sending packets, and the Struct library to properly convert the data type to the `Byte` format, so it can be properly decoded from the MicroBlaze receiving end. The Python script reads the image, splits it into 8x8 blocks, and sends 7 flattened blocks over the UART at once. It then waits for the MicroBlaze to respond with the hard-coded "!" character until sending further packets. The Python code snippet below shows the function implementing this procedure.

```
def uart_op(img_list):
    packet_num = int((img_list.shape[0] / 7))

    ser = serial.Serial(
        port=ComPort,
        baudrate=115200,
        bytesize=8,
        timeout=0,
        stopbits=serial.STOPBITS_ONE,
        rtscts=1,
    )

    while 1:
        for i in range(packet_num):
            if i > 0:
                ser.open()
                # use uart handshake
                rec = ser.read().decode("ASCII")
                while rec != "!":
                    rec = ser.read().decode("ASCII")

            for k in range(7):
                img_payload = img_list[i * 7 + k].flatten()

                for j in range(len(img_payload)):
                    # send one pixel, 8-bit at a time
                    # use B to represent unsigned char
                    sendData = struct.pack(">B", img_payload[j])
                    x = ser.write(sendData)
            # refresh serial port between packets
            ser.close()
            print("Sent block {}".format(i * 7 + k))
        break

    ser.close()
    print("Done sending image, closed serial port")
```

4.3.2 Wireless Communication - TCP

TCP stands for Transmission Control Protocol, which aims to provide a reliable data stream delivery service. It is considered a connection-oriented protocol, where the network connections of a device is established and maintained until both the sender and receiver finished their exchange [10]. It is reliable as it has multiple sync-up signals to ensure the connection is valid. Luckily, the TCP library is already provided, which was built on top of the LwIP stack in MicroBlaze.

Our TCP packet handling was also interrupt based, and the desired packet sender behavior in FPGA was realized by modifying its callback function as the following.

```
// connect handler, initiate the communication by sending the
// first packet to the mirror-server
static err_t
tcp_client_connected(void * arg, struct tcp_pcb * tpcb, err_t err) {
  u8_t apiflags = TCP_WRITE_FLAG_COPY;

  if (err != ERR_OK) {
    tcp_client_close(tpcb);
    xil_printf("Connection error\n");
    return err;
  }

  xil_printf("Connection to server established\n");

  //Store state (for callbacks)
  c_pcb = tpcb;
  is_connected = 1;
  u8 packetinput[TCP_SEND_BUFSIZE] = {0};

  if (telem_num > 0) {
    xil_printf("sending %d telemetry data\n", telem_num);
    memcpy((u8 * ) packetinput, (u8 * ) telem_ratio, TCP_SEND_BUFSIZE);
    telem_num = 0;
  }
```

```
  //Loop until enough room in buffer (should be right away)
  while (tcp_sndbuf(c_pcb) < TCP_SEND_BUFSIZE);

  //Enqueue some data to send
  err =
    tcp_write(c_pcb, /*changed here */ packetinput, TCP_SEND_BUFSIZE,
      apiflags);

  if (err != ERR_OK) {
    xil_printf("TCP client: Error on tcp_write: %d\n", err);
    return err;
  }

  err = tcp_output(c_pcb);

  if (err != ERR_OK) {
    xil_printf("TCP client: Error on tcp_output: %d\n", err);
    return err;
  }

  xil_printf("sent packet\n");

  //Set callback values & functions
  tcp_arg(c_pcb, NULL);
  tcp_recv(c_pcb, tcp_client_recv);
  tcp_sent(c_pcb, tcp_client_sent);
  tcp_err(c_pcb, tcp_client_err);

  return ERR_OK;
}

// receive handler
static err_t
tcp_client_recv
(void * arg, struct tcp_pcb * tpcb, struct pbuf * p, err_t err) {
  //If no data, connection closed
  if (!p) {
```

```
  xil_printf("No data received\n");
  tcp_client_close(tpcb);
  return ERR_OK;
}

xil_printf("Packet received, %d bytes\n", p -> tot_len);

//Print packet contents to terminal
char * packet_data = (char * ) malloc(p -> tot_len);

pbuf_copy_partial(p, packet_data, p -> tot_len, 0);

// RDY, 3 bytes
// waiting until the handshake signal comes
if (packet_data[0] == 82 && packet_data[1] == 68 && packet_data[2] == 89
   && packet_sent >= 0) {
  xil_printf("received packet %c%c%c\n", packet_data[0],
    packet_data[1], packet_data[2]);
  u8_t apiflags = TCP_WRITE_FLAG_COPY;

  u8 packetinput[TCP_SEND_BUFSIZE] = {0};

  // send telemetry data first
  if (packet_sent < IMG_BLOCK_NUM) {
    xil_printf("packet sent is %d\n", packet_sent);
    // send all coefficient data
    xil_printf("coefficient packet number to sent is %d\n",
      packet_sent);
    memcpy((u8 * ) packetinput, (u8 * )(coeff_arr[packet_sent]),
      TCP_SEND_BUFSIZE);
    packet_sent++;
    //Loop until enough room in buffer (should be right away)
    while (tcp_sndbuf(c_pcb) < TCP_SEND_BUFSIZE);
    //Enqueue some data to send
    err =
      tcp_write(c_pcb, /*changed here */ packetinput, TCP_SEND_BUFSIZE,
        apiflags);
```

```
      if (err != ERR_OK) {
        xil_printf("TCP client: Error on tcp_write: %d\n", err);
        return err;
      }

      err = tcp_output(c_pcb);

      if (err != ERR_OK) {
        xil_printf("TCP client: Error on tcp_output: %d\n", err);
        return err;
      }

      xil_printf("sent packet\n");

    }
  } else {
    // not valid handshake message
    xil_printf("received packet %c%c%c\n", packet_data[0], packet_data[1],
      packet_data[2]);
  }

  //Indicate done processing
  tcp_recved(tpcb, p -> tot_len);
  //Free the received pbuf
  pbuf_free(p);

  return 0;
}
```

For the packet definition, we define our packet format to be as shown in Figure 9, with the first byte being the server type, where ”R” stands for mirror-server receives and ”S” stands for mirror-server sends. The second byte is the data type, the third and fourth bytes are the length of the packet, and the rest 130 bytes payload are all coefficient data (or telemetry, depending on data type). The code snippet following the diagram shows how we assembled the packet after acquiring the coefficients from the DCT block.

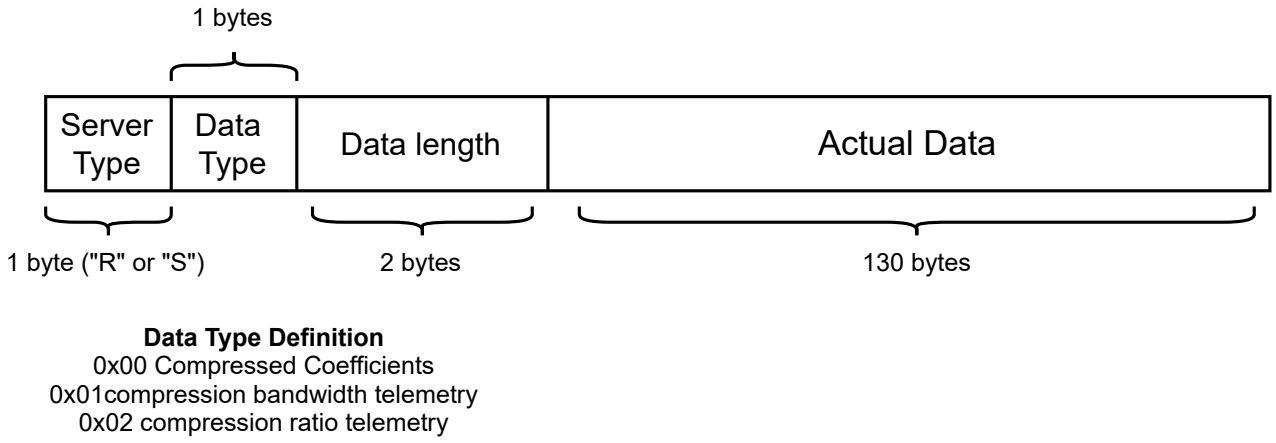


Figure 9: Project TCP packet definition

```
// packet assembly according to definition
void
assemble_packets
(u8* packet_ptr, u8* coeff_ptr, u32 len, u8 type){

        packet_ptr[0] = 82;
        packet_ptr[1] = type;
        packet_ptr[2] = (u8) ((len>>8) & 0xff);
        packet_ptr[3] = (u8) (len & 0xff);

        memcpy((u8*)(packet_ptr + 4), (u8*)coeff_ptr, len);
}
```

The second FPGA mirror server was largely inspired by the TCP echo server starter code given by Xilinx in Vivado SDK. The main modifications are in the `echo.c` file. We would like to echo the packets to another arbitrary client connected to the server in the network instead of the same message sender. This was realized by having the server type byte defined in the data packet, where the TCP server will then decode the request based on it. The code snippet below shows how this is achieved.

```
enum req_type decode_request(char * req, int l) {
  char * receive_str = "R";
  char * send_str = "S";

  if (!strncmp(req, receive_str, l)) {
```

```
    return RECV;
  }
  if (!strncmp(req, send_str, l)) {
    return SEND;
  }
  printf("Received package: %d\n", req[0]);
  return UNKNOWN;
}

int
generate_response
  (struct tcp_pcb * pcb, struct pbuf * p, char * payload, int len) {

    enum req_type msg_type = decode_request(payload, 1);

    switch (msg_type) {
    case RECV:
      return do_receive(pcb, p, payload, len);
    case SEND:
      return do_send(pcb, p, payload, len);
    default:
      dump_payload(payload, len);
      return do_404(pcb, p, payload, len);
    }
  }
```

In the *do_receive*() function, the server writes all the received packet data to its memory without modification. In case we run out of memory, the number of packets it expects to receive is fixed, depending on the image size we are processing. When a second client (Host PC) connects to the server and requests the data, the server will dump all packets written in its memory to the requester in *do_send*(). The Host PC then decodes the data according to our data format definition in Python and visualized our decompressed images and telemetry.

5 Description of Design Tree

As part of our final report, we've uploaded all of our code to GitHub. The readme in the root directory contains a decent overview of the repository, but for completeness we've copied the repository structure section here. Some of this information is repeated in Section 4.2.

5.1 Repository Structure

The overall file structure of this project is as follows. A brief description of each of the folders is provided below

```
docs
figures
src
   axis_custom_dct
   sd_card
   tb_axis_custom_dct
MicroBlaze
   compression-main
   compression-main2
   mirror-server
python
```

5.1.1 docs

Contains our project presentations and final report, for reference.

5.1.2 figures

Contains some figures used in the README.

5.1.3 src

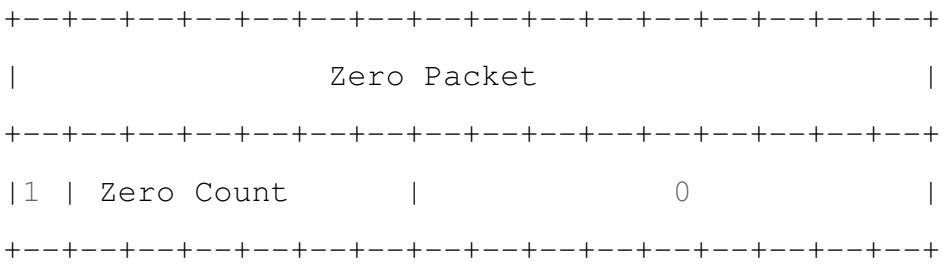
This folder contains all FPGA-related modules that we built for the project. Specifically, there is a dedicated `axis_custom_dct` and `sd_card` folder related to all the DCT and SD card modules, respectively. There is also a `tb_axis_custom_dct` folder which contains the block diagram and simulation files for performing a test of the custom DCT module using the Xilinx AXI VIP.

5.1.3.1 axis_custom_dct

Contains all of our source code, test benches, and memory files for our implementation of a DCT compressor accessed over AXI-Stream. The source code spans several modules, with each module having a 2-in, 2-out interface with input and output valid flags. The module will register input when the input is valid, and will signal valid output when the output is valid. No handshaking takes place, so the downstream module either needs to be able to handle two values per cycle or have sufficient buffer size. There are a couple of test benches for the key modules. that can be used to check that everything is working properly.

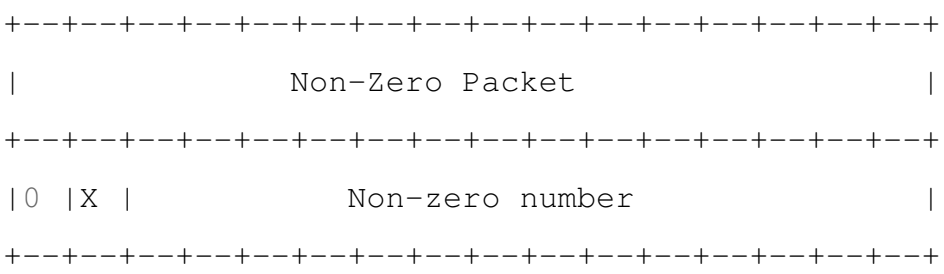
The design also uses two Xilinx FIFOs, which are Xilinx IP. The `ip_repo` contains the compiled module (so that the IP works), but you'll have to regenerate it yourself if you want to modify something.

The AXI-Stream interface assumes that you have a 16-bit data bus transferring the first pixel in the lower 8-bits and the second pixel in the upper eight bits. This was done because the DCT modules take two inputs per cycle. The output is a 32-bit wide bus, with the lower 16 bits being the first encoded coefficient and the upper 16 bits being the second. The run-length encoding scheme is as follows:



Bit 15 being set to 1 indicates that the packet is a zero. Bits `[14:9]` indicate the number of zeros, and a value of 0b000000 indicates that there are 64 zeros (the value wraps around). The lower 9 bits

[8:0] are currently unused and set to zero. A Huffman coding scheme would pack these coefficients a lot more tightly



Bit 15 being set to 0 indicates that the packet is non-zero. Bit 14 is currently don't-care (at least in our 14-bit coefficient implementation) and bits [13:0] are the signed coefficient value. In this case, bit 13 is the sign bit, and requires appropriate conversion on the decompression side.

5.1.3.2 sd_card

Contains all of our source code for the SD card interface module. The original Verilog module was taken from Introductory Digital Systems Laboratory (6.111) at MIT, and is included in the original_code repo. We've also created an AXI interface for the block, and wired it all together in sd_control_ram_v1_0.v.

5.1.3.3 tb_axis_custom_dct

This folder contains a block diagram and simulation files that we created for testing the DCT module with the complete AXI Stream interface. The Xilinx AXI DMA block is used to send and receive AXI Stream from the DCT module, and the Xilinx AXI VIP block is used to exercise the module.

As before, you'll have to have access to the relevant Xilinx IP if you'd like to re-create the block diagram exactly. An image of the block diagram can be found in the GitHub.

5.1.3.4 MicroBlaze

5.1.3.4.1 Compression-main

This folder contains the source code (in C) for the first program uploaded to FPGA#1 MicroBlaze microprocessor. It receives hyperspectral images from the Host PC via UART and stores them into SD card.

UART is set to baud rate 115200 in hardware. Writing to SD card is the most time consuming operation, takes about 0.8 seconds per write.

5.1.3.4.2 Compression-main2

This folder contains all source code (in C) for the second program uploaded to FPGA1 MicroBlaze microprocessor. It reads the image pixel intensities from SD card, passes them to DCT for compression, and reads back the coefficients. It then assembles the coefficients into a custom packet format and sends them to FPGA#2 over TCP.

The DCT FIFO streaming interface is currently not functioning fully, so the coefficients read back are not correct. Otherwise, the data pipeline was tested to be functional.

5.1.3.4.3 Mirror-Server

This folder contains all source code (in C) for FPGA#2. This FPGA acts as a TCP packet mirror server. It receives the packets from one client and stores them in memory, depending on the client request type defined in the first byte in a message. Later, when another client connects and request receiving server's stored packets, the server will send the packet it stored in memory without change. Thus the name "mirror", since it sends and receives packets with no decoding or modification.

This requires the EthernetLite IP in hardware, and is mostly inspired from the LwIP library and HTTP-server example provided by Xilinx.

5.1.3.5 python

Over the course of the project, we created a lot of Python scripts to either generate memory files, test the network interface, or test the DCT algorithm itself. This folder contains the complete set of

these scripts, along with a test image. The test image comes from the Columbia University CAVE Multispectral Image Database.

Perhaps the most important to using the DCT block itself would be `coeff_gen.py`, which generates the DCT coefficients. Second would be `quantization_gen.py`, which generates the quantization bit-shifts. The `host-pc-uart.py` file is used for transferring data to FPGA1 over UART, and the `pc-client.py`. file is used for receiving the run-length encoded coefficients.

6 Tips and Tricks

We came up with several tips and tricks throughout the coures of this project. Hopefully, these will be useful to future students.

- For components with non-standard protocol behaviour, such as the SD card with SPI, it is recommended to use a custom IP block. This allows flexibility in signal requirements, which is often not available in vendor IPs. In addition, vendor IPs sometimes do not support non-standard behaviour, therefore documentation about their external capabilities are scarce/non-existent.
- Make sure to test any Xilinx IP that you decide to use. Preferably, find some working example code online and check it out in a demo environment with known-good IP blocks. This will ultimately give you a better feel for how to interface with the block and hopefully save you some headache later down the line.
- You can right-click on a hardware platform folder in Xilinx SDK and select "Change Hardware Platform Specification" to change the linked .hdf file. This is particularly useful when moving the code between computers and the SDK is not linking to the updated .hdf file.
- If you have multiple communication protocols in the system that are all interrupt based, be careful of how the interrupt vector is set up, how the interrupt routine is initialized and make sure they don't conflict with each other. For example, using an interrupt-based UART and TCP protocols in the same program might cause one interrupt system to fail and the program will hang. A solution to this could be a) initializing the interrupt vector correctly, a) enabling and disabling certain interrupts in different parts of the program, b) setting up priorities to elevate a more frequently used protocol's interrupt over the others.

References

[1] Martha, "Nexys 4 ddr reference manual." https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual.

[2] K. winkel Robbert Willem, "The effect of writing and transmitting sd card data on the consistency of sd card write performance." http://essay.utwente.nl/82256/1/Krawinkel_BA_EEMCS.pdf, Jun 2020.

[3] J. Epler, "Adafruit microsd spi or sdio card breakout board." https://learn.adafruit.com/adafruit-microsd-spi-sdio, Apr 2021.

[4] G. P. Hom, "6.111 introductory digital systems laboratory." http://web.mit.edu/6.111/www/f2015/tools/sd_controller.v, Fall 2015.

[5] "How to use mmc/sdc." http://elm-chan.org/docs/mmc/mmc_e.html, Dec 2019.

[6] "Sd card tutorials." http://www.rjhcoding.com/avrc-tutorials-home.php.

[7] "What is the correct command sequence for microsd card initialization in spi?." https://electronics.stackexchange.com/questions/77417/what-is-the-correct-command-sequence-for-microsd-card-initialization-in-spi, July 2013.

[8] A. M. De Silva, D. G. Bailey, and A. Punchihewa, "Exploring the implementation of JPEG compression on FPGA," in *2012 6th International Conference on Signal Processing and Communication Systems*, (Gold Coast, Australia), pp. 1–9, IEEE, Dec. 2012.

[9] R. Woods, D. Trainor, and J. Heron, "Applying an XC6200 to real-time image processing," *IEEE Design Test of Computers*, vol. 15, pp. 30–38, Jan. 1998. Conference Name: IEEE Design Test of Computers.

[10] P. Pedamkar, "What is tcp protocol?." https://www.educba.com/what-is-tcp-protocol/. Accessed: 2021-04-12.