ECE532: H.263 Media Compression Broadcast on FPGAs

Group 4 Final Report

Justin Hai

Isamu Poy

Bob Zhou

Table of Contents

1 Overview

1.1 Background

The increasing popularity of high definition video content being delivered on bandwidth-limited broadcast networks has driven new video compression standards and image processing techniques [1]. The COVID-19 pandemic and remote work and learning have made video broadcasting more important than ever. Video compression algorithms can be very complicated, and cause latency in processing due to heavy computations [2]. However, the use of FPGAs can speed up the process and allow for real time execution. With the use of an FPGA Network, we hope to create an FPGA video compressor which will allow current university students to share their videos in real time over a limited bandwidth connection.

Research that explores implementing video compression on FPGAs already exists [3]. For example, researchers at UCLA, have outlined a method to implement the discrete cosine transform, two dimensional filtering, vector quantization, and wavelet transforms on FPGAs [3]. However, there are disadvantages in this system, such as not providing a user interface, or allowing decompressed videos to be broadcasted across other FPGAs [3]. Other researchers have developed an FPGA-based Video compressor entirely on the FPGA [2]. However, their system is designed to run on a single FPGA and can only receive data using a computer PCI bus [2].

1.2 Motivation

The inspiration for our project comes from the popularity of media compression of FPGA technology. Since existing approaches seldom use a hybrid model, we believed that implementing part of the system on Microblaze will contribute to the development of compression algorithms on Xilinx FPGAs. Additionally, it would be an advantage over existing approaches as it will also include a software interface which will make it easier to compress and send a desired video file.

1.3 Goals

Our goal for this project is to create a full video compression system which will be capable of taking in a video, compressing it, and sending it over a bandwidth limited network to another FPGA, where it is decompressed and displayed on another computer. The system will contain major components on both the Microblaze soft processor and the FPGA hardware in Verilog. Ideally, we will be able to translate an existing algorithmic implementation of media encoding and convert it to hardware.

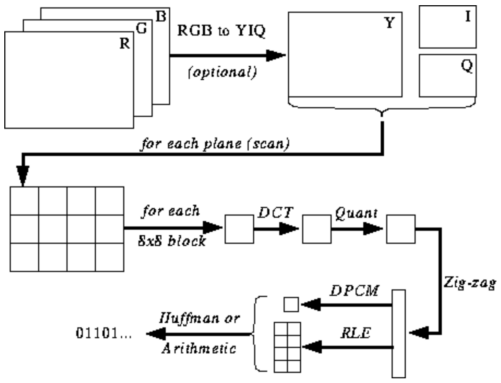


Figure 1. Visualization of compression algorithm

We determined that such an idea can be created over the course of the term because there are current software implementations which we can use to compare outputs, and use as a model for our hybrid pipeline. By using 2 FPGAs, we can use FPGA 1 to send image pixel information to the hardware which encodes the values to a compressed form. Then, we can connect FPGA 2 over a network, and implement a software decoding implementation which will produce an image at the output. In the event that video compression does not work, we have the option to scope down to image compression. Additionally, we hope to create custom hardware implementations that can be contributions to the field of media processing.

1.4 System Changes

Over the course of the term, several major changes were made to the system:

1. Focus on image compression instead of video compression
    a. **Description:** Instead of demoing video compression we decided to focus on getting image compression to work on a JPEG image.
    b. **Reason:** It was observed that video compression is simply image compression applied on a number of images with image compensation. Thus, by getting image compression to work, it would imply that video compression is also possible. Due to time constraints, and unexpected bugs, we felt image compression is sufficient to demonstrate the systems functionality.
2. Use UARTLite to interface to the Microblaze
    a. Description: Instead of using the ethernet to send data bits to the Microblaze, we decided to use the UARTLite IP to interface data sent from the PC to the soft processor.

b. **Reason:** We saw a lack of implementation for sending large amounts of data over the UART interface, and felt that creating an implementation would be a contribution to the hardware community.

3. Use the TEMAC instead of the Ethernet Block
    a. **Description:** We decided to use the TEMAC block to send data between two FPGAs instead of using the Ethernet Block implementation provided from ECE532 tutorials.
    b. **Reason:** In order for our system to work the data needs to flow from FPGA -> IP cores -> FPGA. Thus, we either had to use the TEMAC core or figure out a way to instantiate two Microblaze processors on 1 FPGA. Since we were not experienced with creating two microblazes in one project, and since there was little documentation about it online, we felt less confident about getting it to work. Since TEMAC core had Xilinx documentation, we felt it would be a safer option to go with.

1.5 System Block Diagram

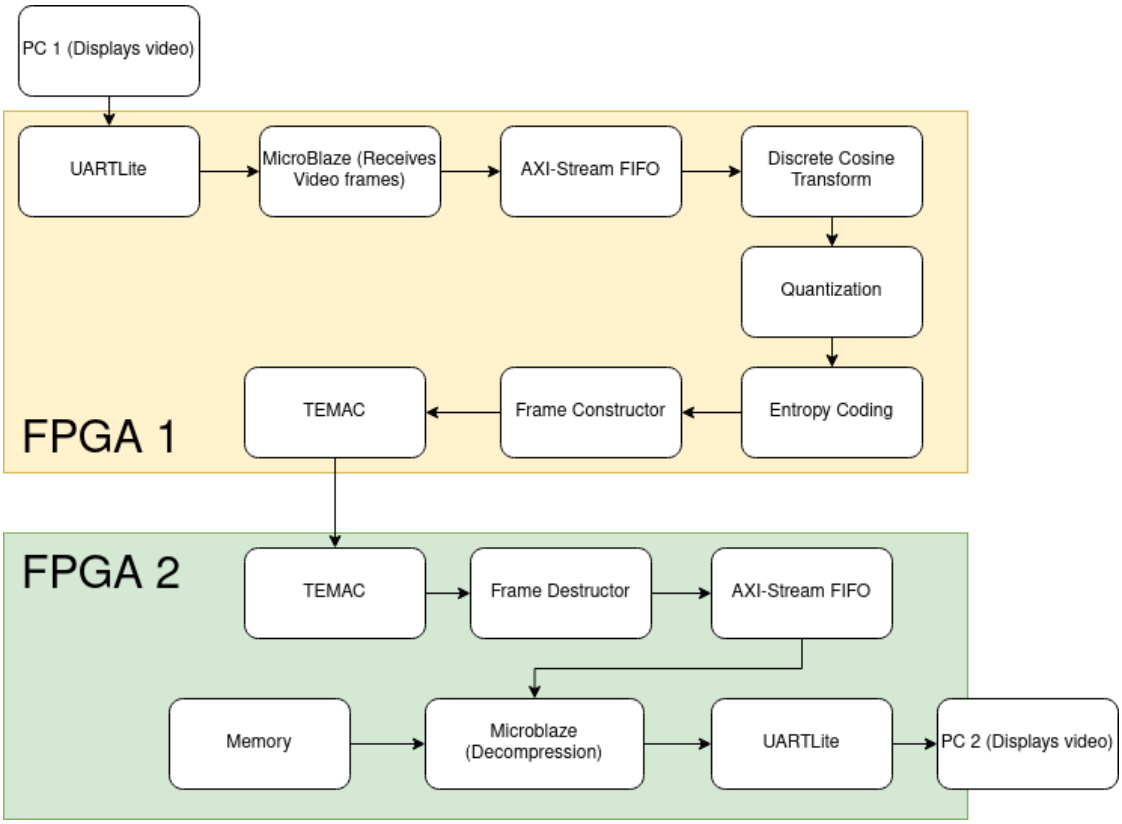


Figure 2: Block Diagram of Design

1.6 Brief Description of IP

IP Name	HW/SW	Origin	Function
two_dimension_dct	HW	Custom	Computes 2-D DCT of 8x8 image pixels
quantizer	HW	Custom	Divides 8x8 DCT coefficient matrix by quantization table
zig_zag	HW	Custom	Generates 64-element bitstream by rearranging 8x8 quantized coefficient matrix
run_length_encoder	HW	Custom	Converts 64-element bitstream into coefficient-frequency pairs
run_length_decoder	SW	Custom	Converts coefficient-frequency pairs into 64-element bitstream
de_zig_zag	SW	Custom	Generates 8x8 quantized coefficient matrix by rearranging 64-element bitstream
de_quantizer	SW	Custom	Multiplies 8x8 quantized matrix by quantization table
compute_inverse_dct	SW	Custom	Computes 2-D Inverse DCT using DCT coefficients to form 8x8 matrix of image pixels
tri_mode_ethernet_mac_0_ex	HW	Third-party	Example design of the tri-mode ethernet MAC
FPGA2_TEMAC	HW	Custom	Modified version of the TEMAC example design
frame_generator_custom.v	HW	Custom	Pack input data into ethernet frame structure
frame_destructor_custom.v	HW	Custom	Destroy ethernet frame structure and send data inside the frame
UARTLite	HW	Third-party	Obtain data between PC and microblaze

AXIS-FIFO	HW	Third-party	Interface between custom IP cores and microblaze software
MIG	HW	Third-party	Utilize DDR memory in order to store large amounts of pixel data

Table 1: IP descriptions

2 Outcome

2.1 Results

Throughout the project, we partitioned the project to make sure that each component of the project was working. Indeed, we were able to get the partitioned parts working, where an image is inputted, partitioned into data packets, sent through the microblaze to the pipeline (with proper output), and have a TEMAC core which successfully establishes a connection between 2 FPGAs. However, we were not able to integrate all components together due to the TEMAC exhibiting unexpected behaviour when we tried to integrate the TEMAC core with the custom IPs. As a result, for FPGA2 we demoed the functionality of the decompression algorithm on the Microblaze processor by itself. Thus, while our system was not interconnected completely, we showed that the goals we set for the individual components were met successfully. Figure 2 and 3 show the successfully received packets and encoded data values respectively. The data shown on Figure 2 has been received over the serial port.

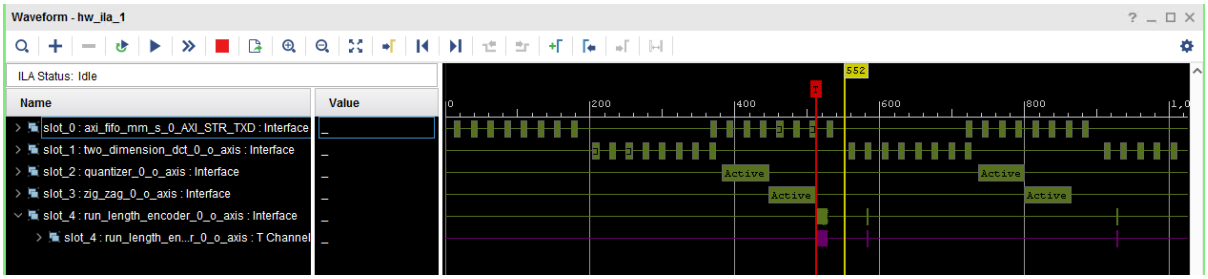


Figure 2: Waveforms of received packets

Figure 3: Waveforms displaying encoded data

2.2 Future Work

In future work, we encourage future developers to debug the TEMAC core to get the pipeline data transmitted from FPGA1 to FPGA2. Alternatively, one may try figuring out how to instantiate 2 Microblaze IPs into FPGA1 and connect FPGA2 with an ethernet IP. Since the main shortcoming of this project was the network connection between the 2 FPGAs, solving this problem will complete the pipeline.

One way to extend this project is to allow for video compression. To do so, one may allow the input of several images, or an MP4 file, and apply an image compensation algorithm. Giving the user the option to compress both images and video will make the system more exciting to see working. Additionally, the user interface may also be improved such that the user sending an image over a network, is able to see a message appear on the UI informing them that the image has been sent. Performance monitoring may also be implemented on the UI to show the user the rate at which media was sent.

While there may be other fancy features which can be added to the system, we believe the mentioned features are necessary for future development in order for the compressor to become a marketable product.

3 Project Schedule

At the project proposal stage, the original project milestones are planned as follows:

Milestone 1:

- Implement compression algorithm in C++ and conduct literature review on existing designs

Milestone 2:

- Create IP cores for compression and ethernet controller for FPGA connection
- Implement decompression algorithm in software

Milestone 3:

- Integrate software and hardware components
- Test functionality of compression hardware with test benches

Milestone 4:

- Develop python script to generate video on FPGA2
- Implement Ethernet Controller on FPGA2

- Ensure decompression functionality after compression on FPGA1

Milestone 5:

- Buffer week: spend time debugging IPs and software

Milestone 6:

- Create GUI and implement extra features
- Look into parallelized computations

However, due to the lack of information about the project details, our group has overestimated the project workload and the original project milestones cannot be achieved. Therefore, the project milestones are modified based on the availability of each team member during the week and the realization of the actual workload of a project component. The final project milestones are shown below:

Milestone 1:

- Explored H.263 implementations on software.
- Implemented DCT in C++, wrote code for inverse DCT.
- Found some H.263 reference code.

Milestone 2:

- Researched on the tri-mode ethernet MAC product guide.
- Implemented a JPEG python project containing the H.263 encoding method.
- Implemented the 1-D DCT and transpose core in C.

Milestone 3:

- Implemented pipeline to generate transformed 8x8 blocks in C.
- Tested the functionality of the TEMAC example code.
- Implemented and tested the 2-D DCT and IDCT, Quantization and De-Quantization, Zig-Zag in C.
- Implemented Run-Length Encoding in C.

Milestone 4:

- Implemented image generation based on pixel value in software.
- Completed the TEMAC design.

Milestone 5:

- Implemented serial communication via Python.
- Implemented UART connection between PC and FPGA.
- Implemented and tested on hardware Transpose and 2-D DCT cores.
- Implemented and tested the hardware Quantizer.

Milestone 6:

- Tested the UART and AXIS FIFO.
- Implemented a video frame divider in Python.
- Implemented Frame Generator in hardware.
- Implemented Zig-Zag and Run-Length Encoder cores in hardware.

Comparing the proposal project milestones with the achieved milestones, it is clearly observed that a lot of planned milestones are not achieved. The project hardware implementation was planned to be complete before Milestone 5, but is delayed to the final milestone 6, leaving no time for integration.

There are several reasons that cause the actual achieved milestone not catching up with the planned milestones, but the fundamental cause is always the lack of knowledge and experience to estimate the difficulty and the workload of the project.

4 Detailed Description of IP Blocks

4.1 DCT

Overview

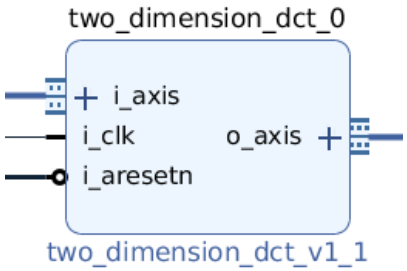


Figure 4: 2D-DCT IP core

The two_dimension_dct block performs a 2-D Discrete Cosine Transform on an 8x8 matrix of image pixels. The output of the two_dimension_dct block is an 8x8 matrix of DCT coefficients. See Appendix B.

Compatibility

The core is compatible with Xilinx Artix-7 FPGAs. In order to meet timing at 100MHz, the DCT core uses Digital Signal Processing (DSP) blocks to perform arithmetic operations. Therefore, the core should be compatible with other Xilinx FPGAs that have hardened DSP logic. Without hardened DSP's, the core should still be usable, but may not be able to meet timing at 100MHz.

Clocking

In its default configuration, the core has been tested at a clock frequency of up to 100MHz, but has timing slack and therefore can theoretically go higher. Without the use of DSP blocks, the core has been tested at a clock frequency of 50MHz.

User Parameters

Table 2: Parameters table

Parameter	Description
Value Width	The data width (in bits) of input pixel values. Must be less than 18 to make effective use of DSP blocks. Since the AXI-Stream protocol requires data widths to be whole bytes, the core will automatically set the i_axis data width to the nearest byte based on Value Width. When sending inputs to the core, the data should be Value Width-bits wide, starting from the LSB, and the upper bits can be padded with zeros or ones.
Scale	The bits allocated to the fractional part of the input pixel. For instance, if Value Width is 17, and the Scale is 7, the lowest 7 bits correspond to the fractional part, and the uppermost 10 bits correspond to the whole part. The 2-D DCT uses fixed point arithmetic. Integer values can be converted to fixed point by multiplying them by 2^Scale

Port Descriptions

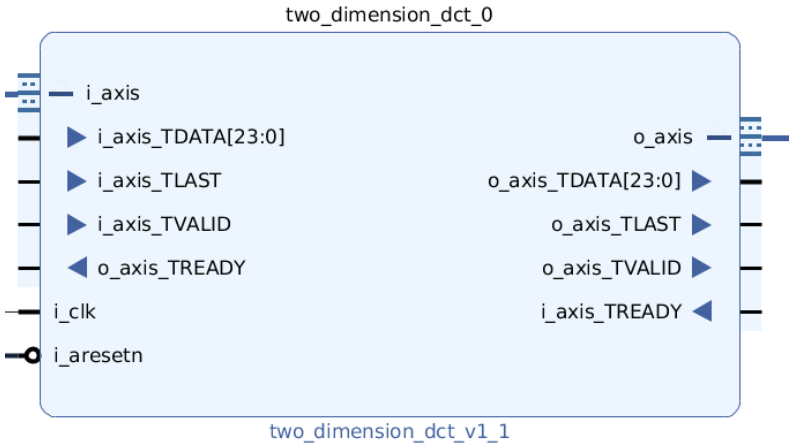


Figure 5: 2D-DCT IP interface

Table 3: Port descriptions of 2D-DCT core

Port	Direction/Type	Description
i_clk	Input Clock	Input clock signal
i_aresetn	Input Reset	Input reset signal (Active Low)
i_axis	AXIS Slave	Receives fixed-point image pixels. TLAST Signal is disconnected, the core will automatically count 64 transfers.
o_axis	AXIS Master	Transmits fixed-point DCT coefficients

Using the core

The i_axis interface receives images for compression. The interface expects 8 x 8 matrices of fixed-point pixel values, fed in 1 value at a time. For instance, let's say we want to send the following 8x8 matrix of pixel values to the compression algorithm via the i_axis interface. Each entry in the matrix is a Value (eg. the 128 highlighted in green

represents a single Value). An example Row in the matrix is highlighted in blue.

128	256	384	512	640	768	896	1024
1152	1280	1408	1536	1664	1792	1920	2048
2176	2304	2432	2560	2688	2816	2944	3072
3200	3328	3456	3584	3712	3840	3968	4096
4224	4352	4480	4608	4736	4864	4992	5120
5248	5376	5504	5632	5760	5888	6016	6144
6272	6400	6528	6656	6784	6912	7040	7168
7296	7424	7552	7680	7808	7936	8064	8192

Figure 6: matrix of potential pixel values

Figure 7 represents how to send these Values into the DCT core:

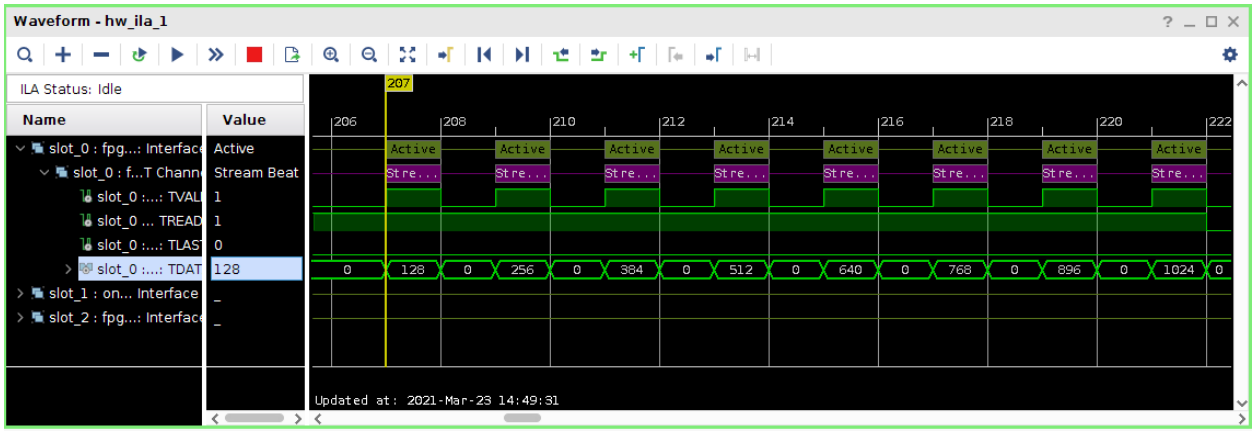


Figure 7: Waveforms of values transmitted

- **Values are sent to the interface 1 at a time**
- The compression algorithm takes in 1 row of values, begins compression, then receives the next row.
- Therefore, **the matrix values should be sent in row-order (eg. 128, 256, 384 .... 7936, 8064, 8192)**.
- Also, this means that TREADY will be high for 8 transfers (eg. 128 - 1024), then will go low for 20-30 cycles, then go high for the next 8 transfers.
- **All 64 transfers must be sent, even if Values are 0.**
- An AXI-Stream handshake is needed for each value.

- **Value Width-bits are received.** For instance, when Value Width = 17, although the TDATA wire is 24 bits wide, **the compression algorithm will only receive the lowest 17 bits**.
- **The spacing (0 value, Valid going low) between 128 and 256 is completely optional, when TREADY is high a new value can be sent every cycle**.
- **TLAST is disconnected**, the algorithm tracks how many transfers it has received, and will know when it has received 64 transfers.

The image below shows the outputs of the DCT core:

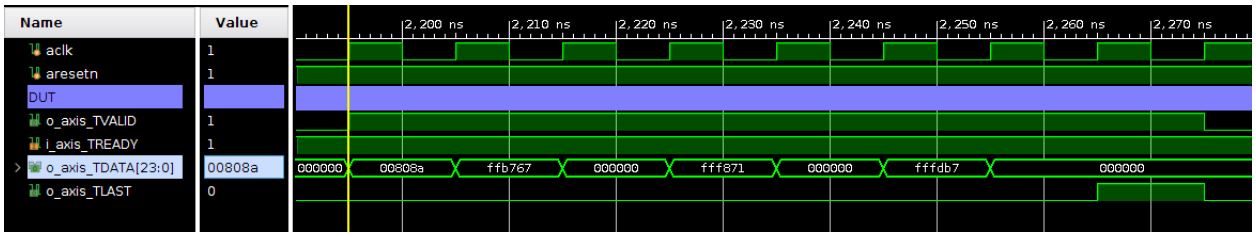


Figure 8: outputs of DCT core

- **DCT Coefficients are sent from the interface 1 at a time**
- **The coefficients will be sent in row-order (eg. 00808a, ffb767, 0, fff871…. Are all on the same row)**.
- **TVALID will be high for 8 transfers (eg. 00808a to 000000), then will go low for 20-30 cycles, then go high for the next 8 transfers**.
- TLAST goes high every 8 transfers (representing that a row has been sent)
- An AXI-Stream handshake is needed for each value.
- Value Width-bits are sent**, rounded up to the nearest byte.** For instance, if Value Width is 17 bits, the TDATA wire will be 24 bits wide, and the DCT core will sign-extend the uppermost 7 bits (negatives stay negative, positives stay positive).

4.2 Quantization

Overview

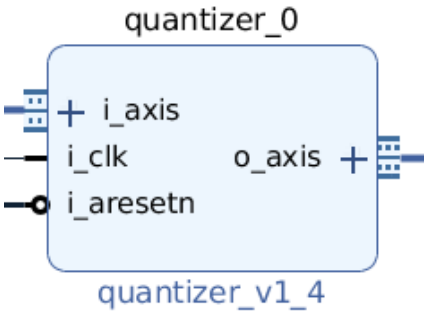


Figure 9: Quantizer IP core

The quantizer performs an element-wise division and clipping (rounding to nearest whole number) operation on an 8x8 matrix of DCT coefficients. This has the effect of dividing each element in the matrix with its corresponding quantization value in the quantization table below:

Quantization Table

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Figure 10: Quantization lookup table

This quantization table was selected because it is commonly used for image compression algorithms. For greater performance, the division is implemented as multiplication with the inverse of each quantization value. The quantization values are converted into fixed points (currently with a scale of 7) and are hard-coded inside the quantization block. Clipping is performed by shifting each quantized element by Scale bits to the right, thus eliminating the fractional part of each element, and converting the elements into integers. The output of this core is an 8x8 matrix of quantized integer DCT coefficients. See Appendix B.

Compatibility

The core is compatible with Xilinx Artix-7 FPGAs. In order to meet timing at 100MHz, the Quantizer core uses DSP blocks to perform arithmetic operations. Therefore, the core should be compatible with other Xilinx FPGAs that have hardened DSP logic. Without hardened DSP's, the core should still be usable, but may not be able to meet timing at 100MHz.

Clocking

In its default configuration, the core has been tested at a clock frequency of up to 100MHz, but has timing slack and therefore can theoretically go higher. Without the use of DSP blocks, the core has been tested at a clock frequency of 50MHz.

User Parameters

Table 4: Input parameters for Quantizer core

Parameter	Description
Input Value Width	The data width (in bits) of input DCT Coefficients. For instance, a matrix with 17-bit coefficients will have a Value Width of 17. Must be less than 18 to make effective use of DSP blocks. Since the AXI-Stream protocol requires data widths to be whole bytes, the core will automatically set the i_axis data width to the nearest byte based on Value Width. When sending inputs to the core, the data should be Value Width-bits wide, starting from the LSB, and the upper bits can be padded with zeros or ones.
Scale	The number of bits allocated to the fractional part of the input pixel. For instance, if Value Width is 17, and the Scale is 7, the lowest 7 bits correspond to the fractional part, and the uppermost 10 bits correspond to the whole part. The core uses fixed point arithmetic. Integer values can be converted to fixed point by multiplying them by 2^Scale (shifting the whole number portion to the left by Scale-bits).

Port Descriptions

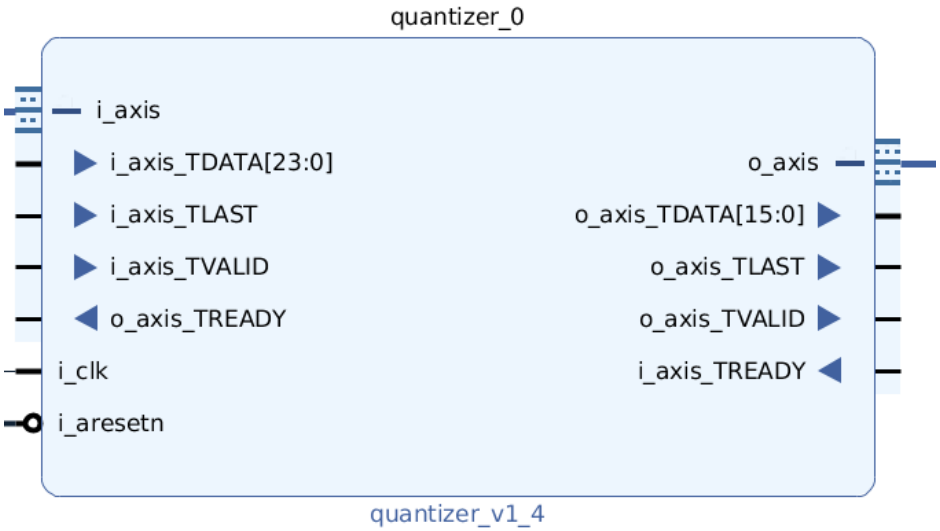


Figure 11: Quantizer IP core interface

Table 5: Port descriptions of Quantizer IP core

Port	Direction/Type	Description
i_clk	Input Clock	Input clock signal
i_aresetn	Input Reset	Input reset signal (Active Low)
i_axis	AXIS Slave	Receives fixed-point DCT Coefficients. TLAST Signal is optional, the core will automatically count 64 transfers.
o_axis	AXIS Master	Transmits quantized integer DCT coefficients

Using the Core

The i_axis interface receives DCT coefficients for quantization. The interface expects 8 x 8 matrices of fixed-point coefficients, fed in 1 value at a time. The image below represents how to send these Values into the Quantizer core:

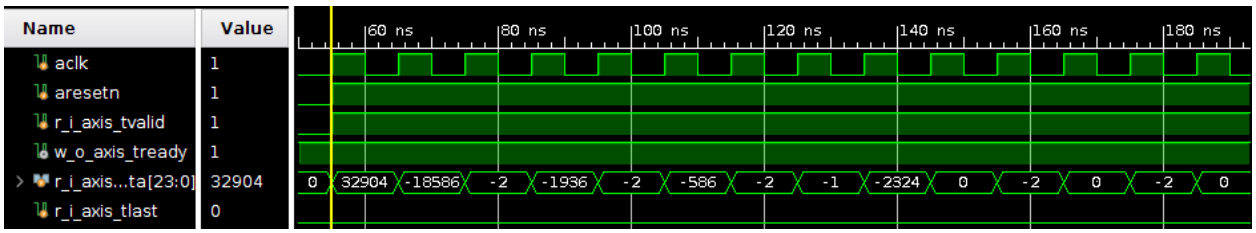


Figure 12: DCT output values

- **Values are sent to the interface 1 at a time**
- The quantizer will receive all 64 coefficients in row-order before beginning computation
- **The coefficient matrix values should be sent in row-order (eg. 32904, -18586, -2, -1936, -2, -586, -2, -1 are in the first row)**.
- **All 64 transfers must be sent, even if Values are 0.**
- An AXI-Stream handshake is needed for each value.
- **Input Value Width-bits are received.** For instance, when Value Width = 17, although the TDATA wire is 24 bits wide, **the compression algorithm will only receive the lowest 17 bits**.
- **TLAST is disconnected**, the algorithm tracks how many transfers it has received, and will know when it has received 64 transfers.

The image below shows the outputs of the Quantizer core:

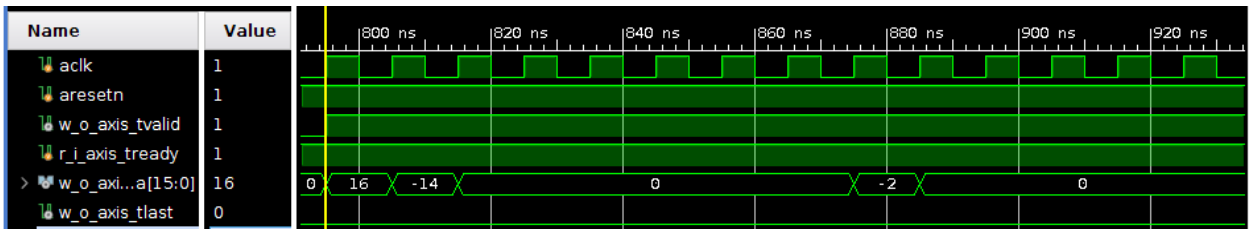


Figure 13: outputs of Quantizer core

- **Quantized Coefficients are sent from the interface 1 at a time**
- **The coefficients will be sent in row-order (eg. 16, -14, 0 …. 0 are all on the same row)**.
- TLAST goes high every 64 transfers (representing that a matrix has been sent)
- An AXI-Stream handshake is needed for each value.
- (Input Value Width-Scale) bits are sent**, rounded up to the nearest byte.** For instance, if Value Width is 17 bits, and the Scale is 7, the output will be 10 bits wide. The TDATA wire will be 16 bits wide, and the core will sign-extend the uppermost 6 bits (negatives stay negative, positives stay positive).

4.3 Zig-Zag

Overview

The Zig-Zag core rearranges the elements of an 8x8 quantized integer DCT coefficient matrix according to the pattern below:

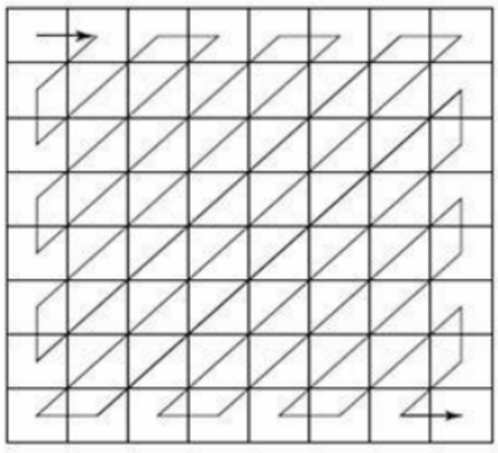


Figure 14: visual concept of zig-zag encoding

This is done because quantized DCT coefficients towards the top left corner of the matrix are typically the largest, because those coefficients correspond to cosine functions with lower frequencies, and are therefore typically larger. Furthermore, the quantization values in the bottom right corners are typically larger, because those coefficients correspond to functions with higher frequencies, and therefore can be reduced significantly with minimal loss of accuracy, often to zero. The output of the Zig-Zag core is effectively the 8x8 input matrix converted into a 64-element bitstream with larger (non-zero) elements gathered towards the start, and zeroed coefficients gathered towards the end. See Appendix B.

Compatibility

The core is compatible with Xilinx Artix-7 FPGAs. Furthermore, the core does not require any hardened logic, and therefore should be compatible with all Xilinx FPGAs.

Clocking

In its default configuration, the core has been tested at a clock frequency of up to 100MHz, but has timing slack and therefore can theoretically go higher.

User Parameters

Table 6: Input Parameters of zigzag core

Parameter	Description
Value Width	The data width (in bits) of input quantized DCT Coefficients. For instance, a matrix with 10-bit coefficients will have a Value Width of 10. Since the AXI-Stream protocol requires data widths to be whole bytes, the core will automatically set the i_axis data width to the nearest byte based on Value Width. When sending inputs to the core, the data should be Value Width-bits wide, starting from the LSB, and the upper bits can be padded with zeros or ones.

Port Descriptions

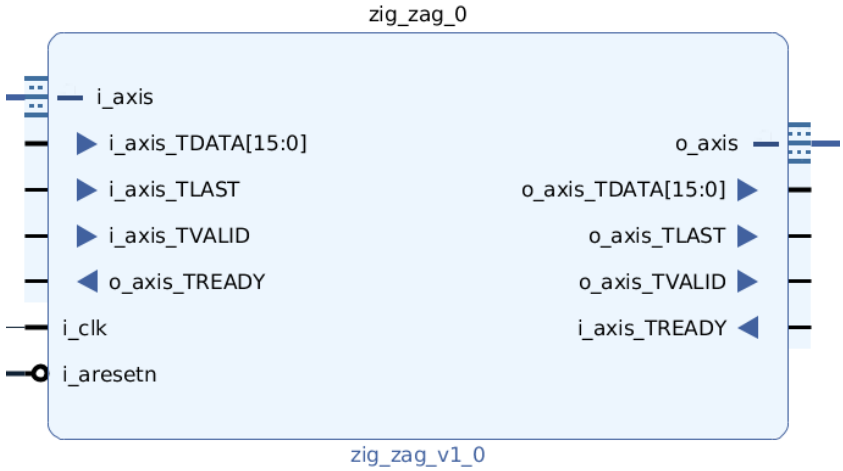


Figure 15: Zigzag IP Core

Table 7: Port descriptions of zigzag core

Port	Direction/Type	Description
i_clk	Input Clock	Input clock signal
i_aresetn	Input Reset	Input reset signal (Active Low)
i_axis	AXIS Slave	Receives integer DCT Coefficients. TLAST Signal is disconnected, the core will automatically count 64 transfers.
o_axis	AXIS Master	Transmits bitstream of quantized DCT coefficients rearranged by the Zig-Zag algorithm

Using the Core

The i_axis interface receives quantized coefficients for zig-zagging. The interface expects 8 x 8 matrices of integer coefficients, fed in 1 value at a time. The image below represents how to send these Values into the Zig-Zag core:

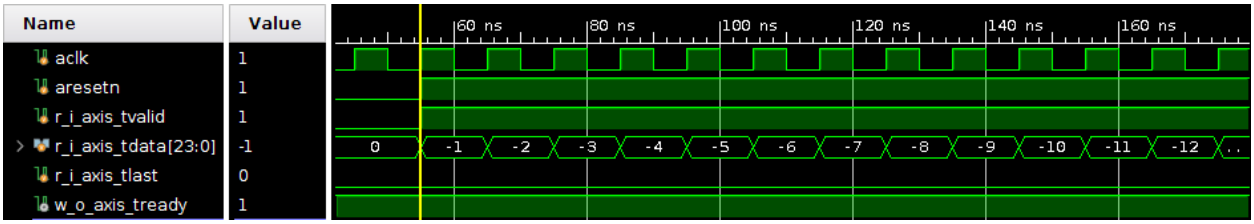


Figure 16: Outputs of Zigzag core

- **Values are sent to the interface 1 at a time**
- The Zig-Zag core will receive all 64 coefficients in row-order before beginning computation
- **The coefficient matrix values should be sent in row-order (eg. -1 … -8 are in the first row)**.
- **All 64 transfers must be sent, even if Values are 0.**
- An AXI-Stream handshake is needed for each value.
- **Value Width-bits are received.** For instance, when Value Width = 10, although the TDATA wire is 16 bits wide, **the compression algorithm will only receive the lowest 10 bits**.
- **TLAST is disconnected**, the algorithm tracks how many transfers it has received, and will know when it has received 64 transfers.
- **Zig-Zagged Coefficients are sent from the interface 1 at a time**
- **The coefficients will be sent in ascending order (eg. -1, is the start of the bitstream and sent first)**.
- TLAST goes high every 64 transfers (representing that a matrix has been sent)
- An AXI-Stream handshake is needed for each value.
- Value Width-bits are sent**, rounded up to the nearest byte.** For instance, if Value Width is 10 bits, the TDATA wire will be 16 bits wide, and the core will sign-extend the uppermost 6 bits (negatives stay negative, positives stay positive).

4.4 Run Length Encoder

Overview

The Run Length Encoder compresses a bitstream into a smaller bitstream consisting of a bitstream value concatenated with the number of its consecutive occurrences. For instance, the bitstream (16, -14, 0, -2, 0, 0, 0, 0, 0) becomes:

1 x 16

1 x -14

1 x 0

1 x -2

5 x 0

The input to the Run Length Encoder is a 64-element bitstream of zig-zagged quantized DCT coefficients. The output of the Encoder is a bitstream whose values are in the following format:

Figure 17: Encoder output format

In an m-bit value, the lowest n bits (n=10 above) represent the value of a coefficient, and the upper m-n bits (6 bits above) represent the number of its consecutive occurrences. For instance, (1 x 16) will be represented by 0000010000010000.

Since the total number of output transfers will vary based on the input bitstream, TLAST is used to indicate the last transfer for a 64-element bitstream. See Appendix B.

Compatibility

The core is compatible with Xilinx Artix-7 FPGAs. Furthermore, the core does not require any hardened logic, and therefore should be compatible with all Xilinx FPGAs.

Clocking

In its default configuration, the core has been tested at a clock frequency of up to 100MHz, but has timing slack and therefore can theoretically go higher.

User Parameters

Table 8: Input parameters of run length encoder

Parameter	Description
Value Width	The data width (in bits) of input quantized DCT Coefficients. For instance, a matrix with 10-bit coefficients will have a Value Width of 10 (n=10). Since the AXI-Stream protocol requires data widths to be whole bytes, the core will automatically set the i_axis data width to the nearest byte based on Value Width. When sending inputs to the core, the data should be Value Width-bits wide, starting from the LSB, and the upper bits can be padded with zeros or ones.
Counter Width	The width (in bits) of the counter used to track frequency (m-n). The counter can count up to (2^(Counter Width) - 1). This width governs the maximum number of consecutive occurrences that can be tracked by the Encoder core. If the frequency exceeds the maximum occurrences, a transfer will be sent out and the counter will be reset to 0 (splitting the value-frequency pair into more than 1 transfer)
Num Values per Transfer	Set to 1. This parameter was originally included for future plans to compress more values into a single AXI-Stream transfer. However, the feature was not tested fully due to time constraints, so it is recommended to send 1 Value-Counter pair per transfer.

Port Descriptions

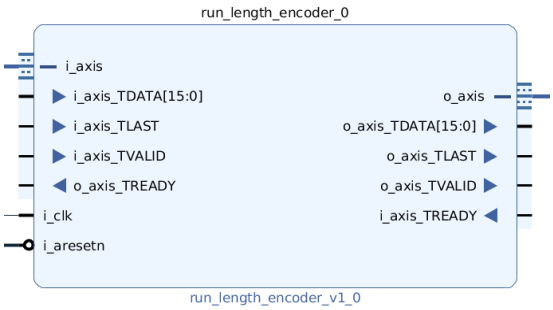


Figure 18: Run length encoder IP core

Table 9: Port descriptions of run length encoder

Port	Direction/Type	Description
i_clk	Input Clock	Input clock signal
i_aresetn	Input Reset	Input reset signal (Active Low)
i_axis	AXIS Slave	Receives zig-zagged quantized DCT Coefficients. TLAST Signal is disconnected, the core will automatically count 64 transfers.
o_axis	AXIS Master	Transmits bitstream of coefficient-frequency pairs. Last coefficient-frequency pair of each bitstream will be marked by TLAST.

Using the Core

The i_axis interface receives zig-zagged quantized coefficients. The interface expects a 64-element bitstream of zig-zagged coefficients, fed in 1 value at a time. The image below represents how to send these Values into the Run Length Encoder. Each coefficient a is sent to the Encoder a times (eg. 1 is sent once, 2 is sent twice, etc):

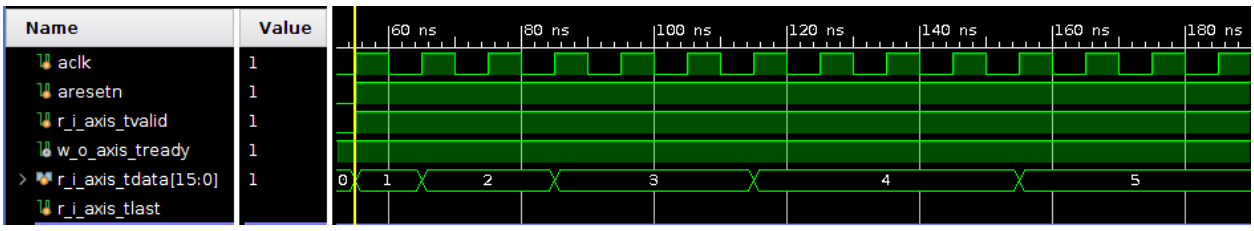


Figure 19: Input data into the run length encoder

- **Values are sent to the interface 1 at a time**
- The Encoder will receive all 64 coefficients in ascending order before beginning computation

- **The coefficient matrix values should be sent in ascending order (eg. start of bitstream is sent first)**.
- **All 64 transfers must be sent, even if Values are 0.**
- An AXI-Stream handshake is needed for each value.
- **Value Width-bits are received.** For instance, when Value Width = 10, although the TDATA wire is 16 bits wide, **the compression algorithm will only receive the lowest 10 bits**.
- **TLAST is disconnected**, the algorithm tracks how many transfers it has received, and will know when it has received 64 transfers.

The image below shows the outputs of the Encoder:

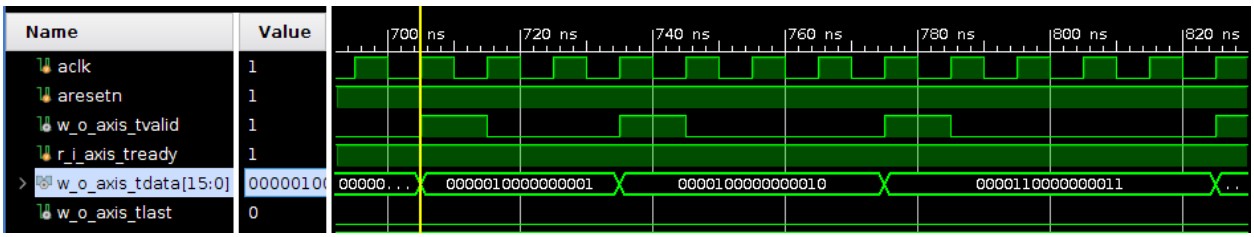


Figure 20: output data into the run length encoder

- **Coefficient-frequency pairs are sent from the interface 1 at a time**
- **TLAST goes high on the last transfer for a bitstream.**
- An AXI-Stream handshake is needed for each value.
- (Value Width + Counter Width)-bits are sent**, rounded up to the nearest byte.** For instance, if the combined width is 10 bits, the TDATA wire will be 16 bits wide, and the core will zero-extend the uppermost 6 bits.
- If the coefficient's consecutive occurrences exceeds maximum frequency, the coefficient-frequency pair will be transmitted in multiple transfers. For instance, if the counter has a Value Width of 4 and Counter Width of 3 (maximum frequency of 7), and the bitstream has 15 x 1, the Encoder will send 3 transfers: 1110001 (7 x 1), 1110001 (7 x 1), 0010001 (1 x 1).

4.5 Run Length Decoder

Overview

The Run Length Decoder takes an encoded bitstream of Coefficient-Frequency pairs, and converts it into a 64-element bitstream of coefficients. The core has been implemented in C, and is run on a Microblaze processor. It is assumed that frequency counters can go up to 64.

User Parameters

Table 10: Input parameters of run length decoder

Parameter	Description
VALUE_WIDTH	The data width (in bits) of input quantized DCT Coefficients. For instance, a matrix with 10-bit coefficients will have a Value Width of 10 (n=10).

Inputs and Outputs

```
void run_length_decoder(u32* encoded_bitstream, int32_t* quantized_matrix)
```

encoded_bitstream: Pointer to a dynamically-allocated array containing the coefficient-frequency pairs.

quantized_matrix: Pointer to a blank, dynamically-allocated 64-element array where the decoded values will be placed.

Output: None

4.6 De-Zig-Zag

Overview

The De-Zig-Zag function takes a 64-element bitstream of coefficients, and performs an inverse zig-zag operation to form an 8x8 quantized matrix. This is performed by traversing over the output matrix using the Zig-Zag pattern, and filling in the values sequentially. The function has been implemented in C, and is run on a Microblaze processor.

To see how the De-Zig-Zag core works, please see hardware/fpga2/src/helloworld.c.

User Parameters

None

Inputs and Outputs

```
void de_zig_zag(int32_t* bitstream, int32_t* dct_coeffs)
```

bitstream: Pointer to a dynamically-allocated 64-element array containing the zig-zagged bitstream.

dct_coeffs: Pointer to a blank, dynamically-allocated 64-element array which represents the 8x8 matrix where de-zig-zagged values will be placed.

Output: None

4.7 De-Quantizer

Overview

The De-Quantizer function de-quantizes an 8x8 matrix of quantized DCT coefficients by performing element-wise multiplication with a quantization table identical to the one used by the Quantizer core. The De-Quantizer performs fixed point multiplication. The function has been implemented in C, and is run on a Microblaze processor.

User Parameters

Table 10: Input parameters of de-quantizer

Parameter	Description
VALUE_WIDTH	The data width (in bits) of input quantized DCT Coefficients. For instance, a matrix with 10-bit coefficients will have a Value Width of 10 (n=10).
SCALE	The number of bits allocated to the fractional part of the input pixel. For instance, if Value Width is 17, and the Scale is 7, the lowest 7 bits correspond to the fractional part, and the uppermost 10 bits correspond to the whole part. The core uses fixed point arithmetic. Integer values can be converted to fixed point by multiplying them by 2^Scale (shifting the whole number portion to the left by Scale-bits).

Inputs and Outputs

```
void dequantizer(int32_t *quantized_matrix, int32_t *quantization_table, int32_t *dequantized_matrix)
```

quantized_matrix: Pointer to a dynamically-allocated 64-element array which represents the 8x8 matrix of quantized coefficients.

quantization_table: Pointer to a dynamically-allocated 64-element array which represents the 8x8 quantization table. Each quantization value is in fixed-point.

de_quantized_matrix: Pointer to a blank, dynamically-allocated 64-element array which represents the 8x8 matrix where de-quantized coefficients will be placed.

Output: None

4.8 Compute Inverse DCT

Overview

The Inverse DCT function uses an 8x8 matrix of fixed-point DCT coefficients to construct an 8x8 matrix of fixed-point image pixel values. The Inverse DCT performs fixed point multiplication. The function has been implemented in C, and is run on a Microblaze processor.

User Parameters

Table 11: Input parameters of inverse DCT

Parameter	Description
VALUE_WIDTH	The data width (in bits) of input quantized DCT Coefficients. For instance, a matrix with 10-bit coefficients will have a Value Width of 10 (n=10).
SCALE	The number of bits allocated to the fractional part of the input pixel. For instance, if Value Width is 17, and the Scale is 7, the lowest 7 bits correspond to the fractional part, and the uppermost 10 bits correspond to the whole part. The core uses fixed point arithmetic. Integer values can be converted to fixed point by multiplying them by 2^Scale (shifting the whole number portion to the left by Scale-bits).

Inputs and Outputs

```
void compute_inverse_dct(int32_t* al_values_for_ul, int32_t* al_values_for_vl, int32_t* dct_coeffs, int32_t* image)
```

al_values_for_ul: Pointer to a dynamically-allocated 16-element array which represents a 4x4 matrix of constants needed to perform the Inverse DCT. To form:

1. Dynamically allocate a 16-element array of integers (eg. called al_values_for_ul).
2. Call the function `void compute_inverse_al_for_ul(int32_t* al_values);` on al_values_for_ul to fill the array with the constants.

al_values_for_vl: Pointer to a dynamically-allocated 16-element array which represents a 4x4 matrix of constants needed to perform the Inverse DCT. To form:

1. Dynamically allocate a 16-element array of integers (eg. called al_values_for_vl).
2. Call the function `void compute_inverse_al_for_vl(int32_t* al_values);` on al_values_for_vl to fill the array with the constants.

dct_coeffs: Pointer to a dynamically-allocated 64-element array which represents the 8x8 matrix of quantized coefficients. Each value is in fixed-point.

image: Pointer to a blank, dynamically-allocated 64-element array which represents the 8x8 matrix of image pixels in fixed-point.

Output: None

4.9 AXIS FIFO Implementation

The AXIS FIFO core contains the necessary designs for interfacing between the Microblaze processor and the custom made IP cores in the pipeline. This IP core is provided by Xilinx, and documentation describes its internals [4]. The implemented design is programmed on the SDK, and follows an example posted on Xilinx's GitHub repository [5]. The code was modified such that it works together with the UARTLite code to forward information taken from the UART interface, and only sends information to the IP core rather than doing both send and receive operations.

4.10 UARTLite Implementation

The UARTLite core in this system contains the designs necessary in order to interface between the FPGA and PC via USB and serial communication. The IP core is provided by Xilinx, and documentation describes how the core can be controlled via drivers or manually by storing information in reserved addresses [6]. For our design, the drivers

were used by following the example posted on Xilinx's GitHub repository [7]. The code was run on a microblaze IP, which was connected to a UARTLite. In order to interface with the PC's terminal, a python script was run on the command line to send pixel data to the serial port iteratively. The example UARTLite code was modified to reformat the received data from ASCII to decimal and stored the information into a data structure. To ensure that all bits were received in the right order, the UARTLite code was modified to check the order tags of each incoming data packet.

4.11 TEMAC Core

The TEMAC core in the context of this report is a complex IP core that contains the tri-mode ethernet MAC and many submodules to ensure its functionality. The overall design of the TEMAC core is a modification of the example design provided by Xilinx Vivado 2018.3 version [8]. The code of the modified version and the original version are all in the provided github link for public access.

For data transmission and receipt, the TEMAC core is designed to have a 8-bit wide AXI4-Stream interface as its main data input/output on the FPGA side and a 4-bit wide MII interface input/output on the PHY layer side. Despite the data bus input/output, the AXI4-Stream interface in the TEMAC core has the TVALID, TLAST, and TREADY bits and the MII interface in the TEMAC has the enable, error bits for transmission and valid, error bits for receipt.

For other functionalities, the TEMAC core takes two clock signals with frequency of 100MHz and 125 MHz as inputs. The 100 MHz clock feeds to the AXI4-lite controller inside the TEMAC core and the 125 MHz clock feeds into the rest of the modules in the TEMAC core. It also has two 1-bit signal MDC output and MDIO INOUT that connects the AXI4-Lite controller to the external PHY layer, allowing the external PHY layer to control the TEMAC.

The data going into and out of the TEMAC core needs to be encapsulated in frames before sending into the ethernet. Figure X shows the standard ethernet frame format. The input data should have the frame structure containing the destination address, the source address, the length/type of the data in order. The preamble and the start of frame delimiter (SFD) are appended in front of the destination address by the TEMAC

core and the data constructed in ethernet frame format is sent to the output.

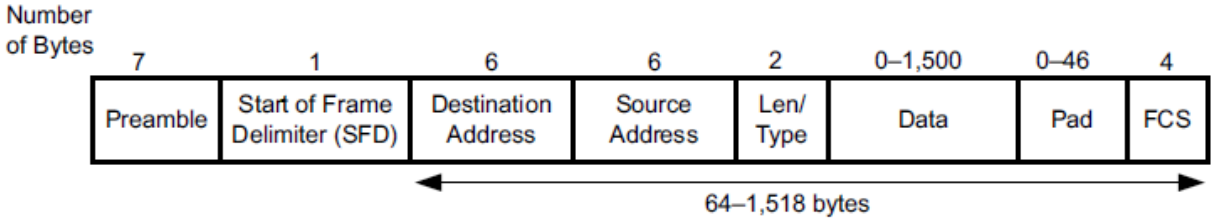


Figure 21: Standard ethernet frame format

The TEMAC contains the following major modules:

Tri-mode Ethernet MAC is the Vivado IP. It is the core IP that needs to be integrated with the rest of the modules to ensure its functionality.

User_side_FIFO is a RTL AXI4-Stream Asynchronous FIFO. It is directly connected to the tri-mode ethernet MAC.

Axi_lite_controller is an RTL AXI4-lite state machine that controls the TEMAC core. It contains a frame filter that allows the users to set up specific frames that the TEMAC should receive. Any frame that does not match with the frame filter will be blocked.

Basci_pat_gen is an RTL module that generates a certain data format in a frame

structure. It contains a submodule called address_swapper that collects the receiver FIFO outputs, swaps the frames' destination address and source address and sends the modified frame data to the transmitter FIFO inputs. This module is not included in the final design, but is frequently used for testing purposes.

To implement the TEMAC core into our FPGA design, part of the example design code of the TEMAC has been modified. The clock wizard and the reset generator inside the example design project are deleted as they are conflicting with the top level clock wizard and reset generator. The TEMAC transmission speed is set to be fixed at 100Mb/s, which is the highest speed of the MII interface. The MAC mode is set to default full duplex instead of half duplex to avoid complexity. Many inputs and outputs related to the simulation are deleted such as frame_error, activity_flash, etc. The frame filter inside the AXI4-lite controller is set to match the FPGA board MAC address.

Furthermore, due to the limitation of the lab FPGA board, the TEMAC core needs to be connected to the MII to Reduced-MII Vivado IP block to convert the MII interface to Reduced-MII before connecting to the external PHY layer.

For communication of 2 FPGAs, there will be one TEMAC implementation on each FPGA. As will be mentioned in the next section, the TEMAC core on the first FPGA (transmitter) will be connected to a Frame Generator custom IP block and the TEMAC core on the second FPGA (receiver) will be connected to a frame Destroyer custom IP block.

4.12 Frame Generator and Frame Destroyer

Both the Frame Generator and the Frame Destroyer are custom IP using AXI4-Stream interface. The Frame Generator is designed to package the received data into the frame structure mentioned in Section 3.7 and send it to the input of the TEMAC core. The Frame Destroyer, on the other hand, receives the data in the frame format from the TEMAC core, destroys the frame and sends the data inside the frame into the output.

The inputs and outputs of the Frame Generator and the Frame Destroyer are similar to the one of a FIFO block. They both are using a clock with 100 MHz frequency. The transmitter side and the receiver side[1], the Frame Generator and Frame Destroyer also share the same clock and reset signal (unlike asynchronous FIFO).The Frame Generator has a 2-byte wide data bus for the input data and 1-byte wide data bus for the output data. Despite the data channel, both the transmitter side and the receiver side of the Frame Generator have the TVALID bit and the TREADY bit. The transmitter side has an additional TLAST bit that indicates the last byte of the frame. The Frame Destroyer takes 1-byte wide data as the input and outputs 2-byte wide data. Similar to the Frame Generator, its transmitter and receiver sides have the TVALID and TREADY bit. However, its transmitter side has a TLAST bit that is used for the Microblaze to identify the last two bytes of the 8x8 encoded fix point value.

The Frame Generator has a state machine that controls its behavior. It is designed to wait for the data coming in. When the data is received from the inputs, the frame generator stores it in its own block memory. The Frame Generator sends the frame output when the block memory has stored 200 bytes of data, therefore it sends the frame containing data length of 200 bytes. It will also send the frame output when it has waited for 100 cycles without receiving any data from the input. In this case, the frame sends any data that remains in the block memory with data length less than 200 bytes.

The Frame Destroyer, can be considered as a simpler and reversed version of the frame generator. It takes in the frame data from the TEMAC and does nothing until it reads the byte indicating the data size inside the frame. It then stores the size value and

[1] The transmitter side of the Frame Generator is connected to the TEMAC core, while the receiver side of the Frame Destroyer is connected to the TEMAC core.

starts to transmit two bytes of data from the frame to the output of the block. Inside the data bytes there is a two-byte marker indicating that the data byte before it is the last byte information of an encoded 8x8 matrix. The Frame Destroyer will then trigger the TLAST bit at the transmitter output.

5 Design Tree Description

Legend:

- **Folder**
- File.txt

G4_H263_Compression:

1. **doc:** Holds all documents including presentations, etc.
    a. final_report.pdf
    b. presentation_mid_project_demo.pdf: Slides from mid-project demo.
    c. presentation_final_demo.pdf: Slides from final demo.
2. README.MD:
3. **src:** Holds all Verilog, C and Python files used in project
    a. **files:**
        i. **fpga1:**
            1. cat.jpeg: Test image
            2. fifo_mem.v: FIFO used by frame generator
            3. frame_destructor_custom.v: Converts Ethernet frames into AXI-Stream transfers
            4. frame_generator_custom.v: Converts AXI-Stream transfers into Ethernet frames
            5. one_dimension_dct.v
            6. quantizer.v
            7. run_length_encoder.v
            8. Serial2.py: Serial communication
            9. transpose.v
            10. two_dimension_dct.v
            11. zig_zag.v
        ii. **fpga2:**
            1. Helloworld.c: Contains decompression functions and Microblaze flow
    b. **ip_repo:**
        i. **fpga1:**
            1. **fpga1_deployment_receiving_block:** Receives AXI-Stream packets

2. **fpga1_deployment_source_block:** Sends AXI-Stream packets
3. **ip_frame_destructor_custom**
4. **ip_frame_generator_custom**
5. **ip_quantizer**
6. **ip_run_length_encoder**
7. **ip_transpose**
8. **ip_zig_zag**
9. **one_dimensional_dct**
10. **two_dimensional_dct**

c. **projects:** Holds all Vivado projects used to generate bitstreams/test cores
   i. **fpga1:**
      1. **fifoStream:** Contains top-level block design for FPGA 1
         a. **fifoStream.sdk/streamFIFO/src/**helloworld.c: Contains functions for connecting to PC 1 via UARTLite and sending data to compression cores.
      2. **proj_compression_deployment:** Used to test compression algorithm on FPGA.
      3. **proj_frame_generator_custom:** Used to test in simulation.
      4. **proj_frame_generator_custom_deployment:** Used to test core on FPGA.
      5. **proj_frame_destructor_custom:** Used to test in simulation.
      6. **proj_frame_destructor_custom_deployment:** Used to test core on FPGA.
      7. **proj_one_dimensional_dct:** Used to test in simulation.
      8. **proj_one_dimensional_dct_deployment:** Used to test core on FPGA.
      9. **proj_quantizer:** Used to test in simulation.
      10. **proj_quantizer_deployment:** Used to test core on FPGA.
      11. **proj_run_length_encoder:** Used to test in simulation.
      12. **proj_run_length_encoder_deployment:** Used to test core on FPGA.
      13. **proj_transpose:** Used to test in simulation.
      14. **proj_transpose_deployment:** Used to test core on FPGA.
      15. **proj_two_dimension_dct:** Used to test in simulation.
      16. **proj_two_dimension_dct_deployment:** Used to test core on FPGA.
      17. **proj_zig_zag:** Used to test in simulation.
      18. **proj_zig_zag_deployment:** Used to test core on FPGA.
   ii. **fpga2:**

1. **proj_fpga_2_deployment:** Contains top-level block design for FPGA 2.

d. **tb:** Holds all testbench files
   i. **fpga1:**
      1. tb_frame_destructor_custom.sv
      2. tb_frame_generator_custom.sv
      3. tb_one_dimension_dct.sv
      4. tb_path_switch.sv
      5. tb_quantizer.sv
      6. tb_run_length_encoder.sv
      7. tb_transpose.sv
      8. tb_two_dimension_dct.sv
      9. tb_zig_zag.sv

6 Tips and Tricks

- Avoid using 3rd party IP cores if you can because they can sometimes be like a blackbox during debugging, which can be very difficult.
- The example design project can be accessed by right clicking on a Vivado IP block -> open example design. Note that the code in the generated example design project matches with the current IP settings.
- Writing synthesizable Verilog code is an important concept to be aware of.
- For Serial communication, make sure to account for delays in sent data.
- When writing hardware, plan out core behaviour at high level before writing code (eg. diagrams, tables, etc), but begin writing code as soon as the plan is complete. You'll probably run into edge cases or problems that you didn't think of in the planning stage.
- Consider writing/simulating unfamiliar hardware cores in C before implementing in Verilog. This allows you to prototype and functionally verify your algorithms quickly, and means that you have a potential Microblaze implementation in case your hardware cores cannot be completed.
- Plan in detail how cores will communicate with one another. Consider using standards (eg. AXI, etc), and consider keeping an interface doc so that all group members are on the same page. Integration will be much easier.

References

[1] I. (n.d.). Broadcast Video Infrastructure Implementation Using FPGAs. Retrieved from

https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-brdcst0306.pdf

[2] “An FPGA-based video compressor for H.263 compatible bitstreams,” *IEEE Xplore*. [Online]. Available: https://ieeexplore.ieee.org/document/854657. [Accessed: 31-Jan-2021].

[3] Schoner, B., Villasenor, J., Molloy, S., & Jain, R. (n.d.). Techniques for FPGA Implementation of Video Compression Systems. *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays, FPGA*.

[4] Xilinx. “AXI4-Stream FIFO v4.1.” LogiCORE IP Product Guide, www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s/v4_1/pg080-axi-fifo-mm-s.pdf.

[5] Jacobfeder. “Jacobfeder/Axisfifo.” *GitHub*, github.com/jacobfeder/axisfifo/blob/master/axis-fifo.c.

[6] Xilinx. “AXI UART Lite v2.0.” LogiCORE IP Product Guide, www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf.

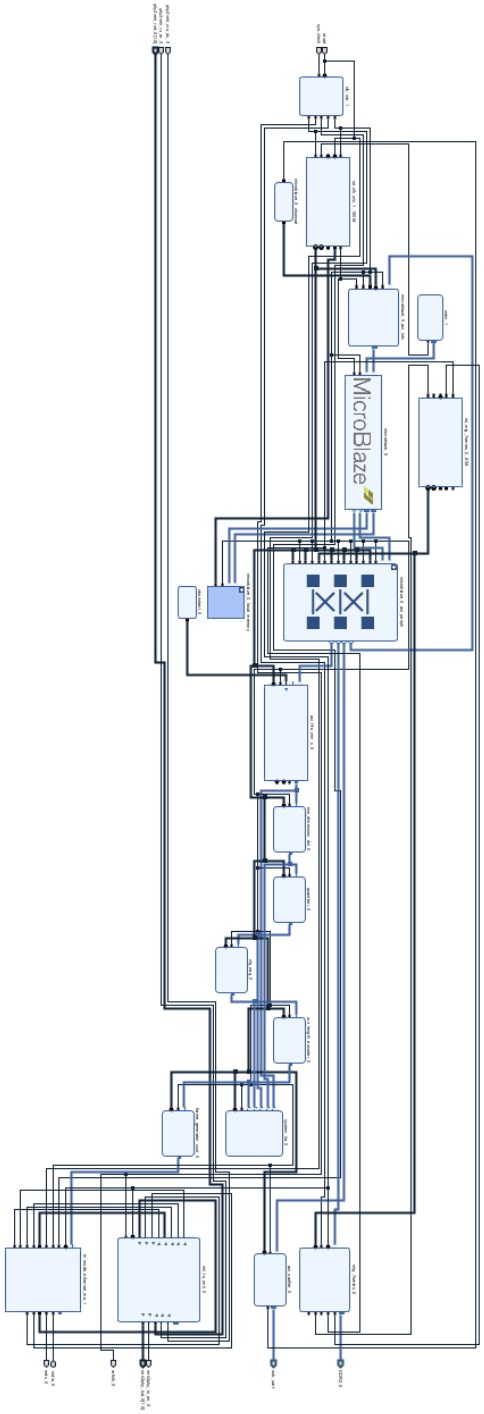
[7] Xilinx. “Xilinx/Embeddedsw.” *GitHub*, 10 Apr. 2020, github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/uartlite/examples/xuartlite_polled_example.c.

[8] Xilinx. “Tri-Mode Ethernet MAC v9.0.” LogiCORE IP Product Guide, www.xilinx.com/support/documentation/ip_documentation/tri_mode_ethernet_mac/v9_0/pg051-tri-mode-eth-mac.pdf.

[9] Shnain, Hayder Waleed, et al. “Implementation of Run Length Encoding Using Verilog HDL.” *International Journal of Science and Research (IJSR)*, 2018, pp. 1–4., doi:https://www.ijsr.net/archive/v9i3/SR20306192039.pdf.

Appendix

Appendix A: FPGA1 Block Design

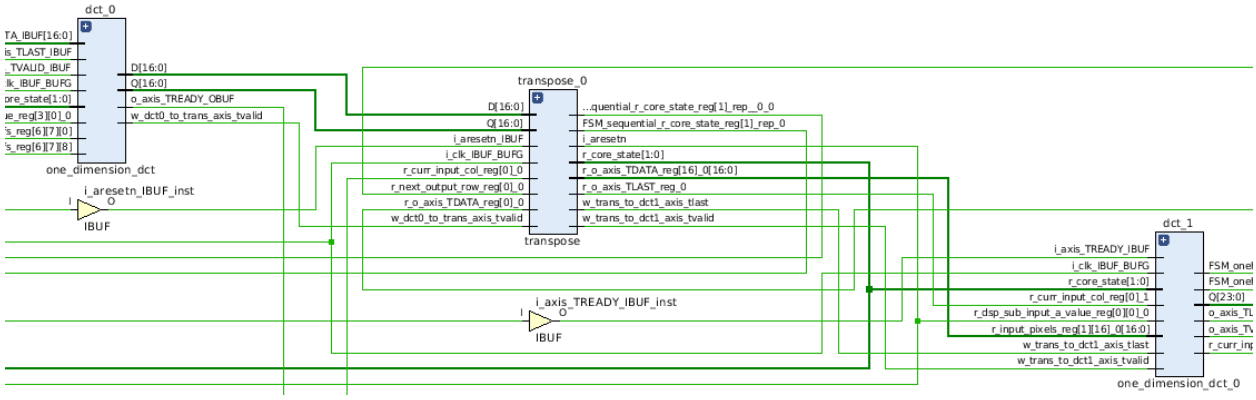


Appendix B: Further Design Details

How It Works

two_dimension_dct

The two_dimension_dct converts an 8x8 matrix of fixed-point image pixels into an 8x8 matrix of fixed-point DCT coefficients. The implementation follows the design in [2]. The coefficients are computed using two one_dimension_dct's separated by a matrix transpose:



one_dimension_dct

The one_dimension_dct performs a 1-D DCT on each row of the 8x8 matrix of image pixels, one row at a time. The algorithm is inspired by the design in [2]. The core is implemented using an FSM consisting of 3 states:

1. STATE_INPUT_COLLECTION: The core reads in 8 values (1 row) of the input DCT coefficient matrix. Changing states after reading 8 values causes the 2-D DCT core to have several cycles between reading rows of the matrix.
2. STATE_COMPUTATION: The DCT core contains 8 DSP blocks set up to perform DSP pre-add-and-mul as well as pre-sub-and-mul operations. Computation of each DCT coefficient (marked by its index in the row) is split amongst the DSP blocks. Image pixels and DCT constants are fed into the DSPs. To effectively track the values inside the DSP blocks, a register pipeline tracks the indices that are currently at each stage of the DSP pipeline:

r_dsp_1_setup: Contains the index that is being loaded into the input registers of the DSP blocks. Increments by 1 every cycle.

r_dsp_2_load: Contains the index that is being loaded into the input registers of the DSP blocks.

r_dsp_3_preadd/presub: Contains the index that is being preadded/presubtracted by the DSP.

r_dsp_4_mul: Contains the index that is currently being multiplied by the DSP.

r_dsp_5_finished: Contains the index that can be read this cycle.

In every cycle, row **r_dsp_1_setup** is loaded into the input registers of the DSP. Simultaneously, the outputs of the DSP blocks are added together and bit-shifted for fixed-point correctness, before being loaded into row **r_dsp_5_finished** of the output coefficients row.

3. STATE_OUTPUT_TRANSMISSION: Once all indices have been loaded into the output row, it is transmitted via AXI-Stream. After all transmissions are complete, the core reverts back to STATE_INPUT_COLLECTION.

Transpose

The transpose core accumulates, transposes, then transmits an 8x8 matrix. The implementation follows the design in [2]. The core uses an FSM with 3 states:

1. STATE_INPUT_COLLECTION: The core reads in all 64 values of the input DCT coefficient matrix.
2. STATE_COMPUTATION: The core performs an 8x8 transpose in column-major order. All 8 elements of each column are streamed into the following design [2

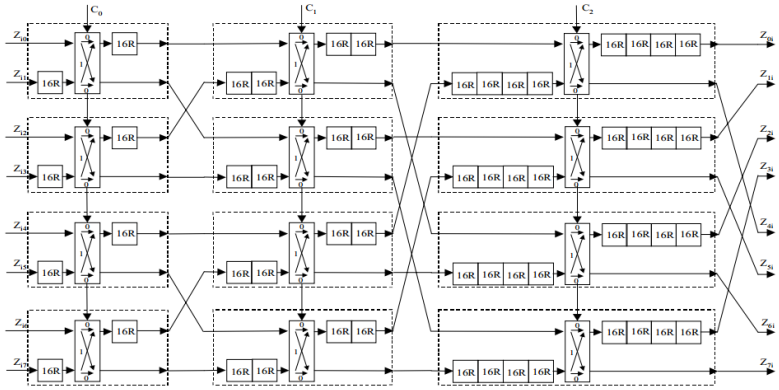


Figure 4. Bit serial transposition operator.

Each stage of the transpose core uses registers, counters and routing to switch the paths of the elements so that the elements are in the correct rows at the end of the last stage.

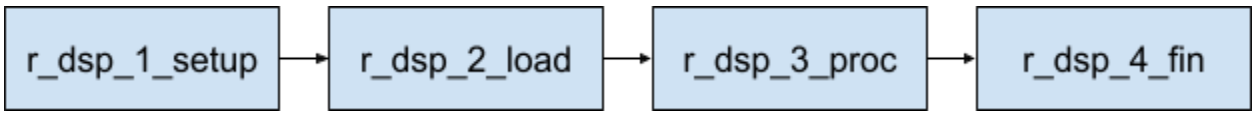
3. STATE_OUTPUT_TRANSMISSION: Once all columns have been loaded into the output matrix, the 8x8 matrix is transmitted via AXI-Stream in row-major order. After all transmissions are complete, the core reverts back to STATE_INPUT_COLLECTION.

Quantizer

The quantizer performs element-wise multiplication of the input 8x8 DCT coefficient matrix with an 8x8 inverse quantization table. Both matrices are in fixed point, but the results are converted to integers by arithmetically shifting the outputs to the right by Scale bits to eliminate the fractional components. Arithmetic shifting preserves positive and negative numbers.

The core is implemented using an FSM consisting of 3 states:

4. STATE_INPUT_COLLECTION: The core reads in all 64 values of the input DCT coefficient matrix.
5. STATE_COMPUTATION: The quantizer core contains 8 DSP blocks set up to perform multiple operations. DCT coefficients and quantization values are fed into the DSPs in row-major order. To effectively track the values inside the DSP blocks, a register pipeline tracks the rows that are currently at each stage of the DSP pipeline:



r_dsp_1_setup: Contains the row that is being loaded into the input registers of the DSP blocks. Increments by 1 every cycle.

r_dsp_2_load: Contains the row that has been loaded into the input registers of the DSP blocks.

r_dsp_3_process: Contains the row that has just been processed (quantized) by the DSP.

r_dsp_4_finished: Contains the row that has just been output by the DSP.

In every cycle, row **r_dsp_1_setup** is loaded into the input registers of the DSP. Simultaneously, the outputs of the DSP blocks are clipped and loaded into row **r_dsp_4_finished** of the output coefficients matrix.

6. STATE_OUTPUT_TRANSMISSION: Once all rows have been loaded into the output matrix, the 8x8 matrix is transmitted via AXI-Stream in row-major order. After all transmissions are complete, the core reverts back to STATE_INPUT_COLLECTION.

zig_zag

The zig_zag core rearranges the input 8x8 matrix of quantized DCT coefficients into a 64-element bitstream with larger values clustered towards the head of the bitstream, and zeros gathered at the tail.

The core is implemented using an FSM consisting of 2 states:

1. STATE_INPUT_COLLECTION: The core reads in all 64 values of the input quantized DCT coefficient matrix.
2. STATE_OUTPUT_COLLECTION: The core transmits quantized coefficients one at a time, starting with element [0][0]. Once an element has been successfully transmitted (TREADY = 1 indicating that a handshake was achieved), the next coefficient to be transmitted is selected according to the zig-zag algorithm

The zig-zag algorithm has 2 main components:

1. Zig-Zag state: This determines whether the algorithm will travel in a diagonal, horizontal, or vertical direction to get the next coefficient.
2. Direction:

```
/*
        Direction:
        | 1 2 3 4 |
        | 5 6 7 8 |
        | 9 A B C |
        | D E F 0 |
        1 (Positive) represents moving in direction towards top right (eg. 6->3)
        -1 (Negative) represents moving in direction towards bottom left (eg. 6->9)
*/
```

During STATE_OUTPUT_COLLECTION, the algorithm performs certain actions depending on current Zig-Zag state:

1. ZZ_STATE_DIAGONAL:

a. next_row = curr_row + 1,
b. next_col = curr_col - 1
c. Or reversed, based on direction
d. If we will hit a horizontal wall in the next cycle, Zig-Zag State = ZZ_STATE_HORIZONTAL
e. If we will hit a vertical wall in the next cycle, Zig-Zag State = ZZ_STATE_VERTICAL
f. Check for horizontal wall first because we want horizontal behaviour at corners

2. ZZ_STATE_VERTICAL:

```
/*
    Vertical wall:
    | 1 2 3 4 |
    | 5 6 7 8 |
    | 9 A B C |
    | D E F 0 |
    For instance, when 2 -> 5, we hit a vertical wall. In this case, we must then transition down to 9
    and reverse the diagonal direction, before proceeding diagonally again
*/
```

a. next_row = curr_row + 1,
b. next_col = curr_col
c. Or reversed, based on direction
d. Direction reversed
e. Zig-Zag State = ZZ_STATE_DIAGONAL

3. ZZ_STATE_HORIZONTAL:

```
/*
    Horizontal wall:
    | 1 2 3 4 |
    | 5 6 7 8 |
    | 9 A B C |
    | D E F 0 |
    For instance, when 6 -> 3, we hit a horizontal wall. In this case, we must then transition right to
    4 and reverse the diagonal direction, before proceeding diagonally again
*/
```

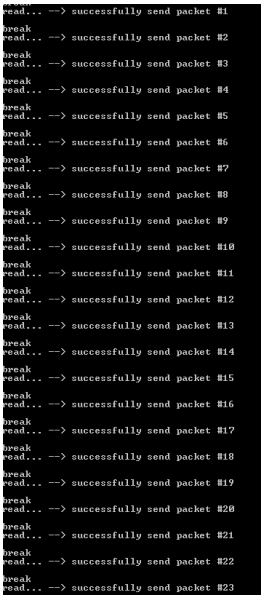
a. next_row = curr_row
b. next_col = curr_col + 1
c. Or reversed, based on direction
d. Direction reversed
e. Zig-Zag State = ZZ_STATE_DIAGONAL

run_length_encoder

The run_length_encoder converts a 64-element bitstream into a bitstream of coefficient-frequency pairs. The implementation is inspired by [9]. The core uses an FSM with 2 states:

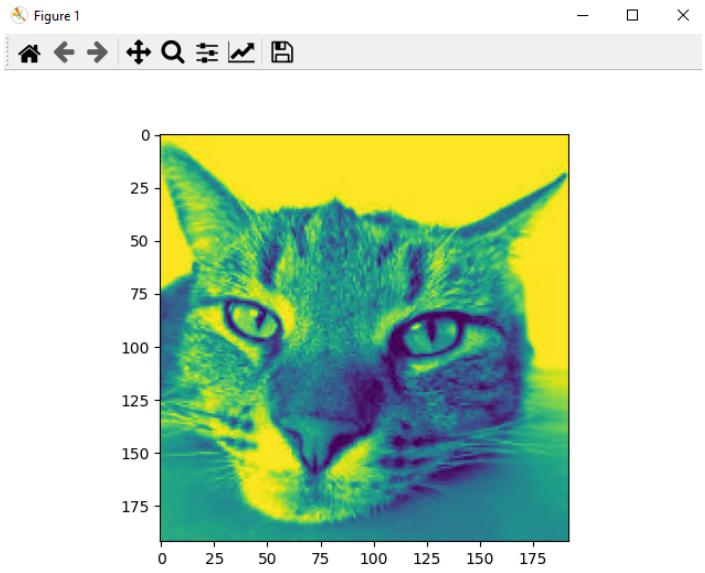
1. STATE_INPUT_COLLECTION: The core reads in all 64 values of the input bitstream.
2. STATE_OUTPUT_TRANSMISSION: The core goes through the input bitstream element-by-element.
    a. If the current element is the same as the current coefficient, the counter is incremented.
    b. If the current element and the current coefficient are different, the counter and current coefficient are concatenated and transmitted. The counter is reset to 1, and the current element is set as the current coefficient.
    c. If the current element is the same as the current coefficient but the timer is saturated, the counter and current coefficient are concatenated and transmitted. The counter is reset to 1, and the current element is set as the current coefficient.

Appendix C: Miscellaneous



Pyserial Interface sending ordered data over to Vivado SDK

Simple GUI interface to get user input



Sampled image for testing