FPGA BASED HARDWARE ACCELERATOR FOR REAL TIME IMAGE COMPRESSION

By
Yicheng Zhou

Senior Thesis in Electrical Engineering
University of Illinois at Urbana-Champaign
Advisor: Thomas Moon

December 2024

Abstract

As imaging sensors evolve, so does the need for high-throughput data compression systems to handle increasing resolutions and frame rates. This thesis investigates the implementation of a hardware-accelerated encoder, designed in HDL for FPGA and ASIC platforms, providing a flexible, high-performance solution for various matrix-based data types. Additionally, a complementary decoder is developed in a high-level programming language to validate and demonstrate the encoder's functionality.

Subject Keywords: FPGA (Field-Programmable Gate Array), Image Processing, Real-Time Data Processing,

Contents

1 Introduction 1

2 Literature Review 2

2.1 Intra-prediction . . . 3

2.1.1 Hardware Optimization . . . 3

2.2 Discrete Cosine Transform . . . 4

2.2.1 Hardware Optimization . . . 4

2.3 Quantization . . . 5

2.4 Entropy Encoding . . . 5

2.4.1 Hardware Optimization . . . 6

3 Technical Overview 7

3.1 Interface . . . 8

3.2 Memory Controller . . . 8

3.2.1 RAM Array . . . 9

3.3 Computational Unit . . . 10

3.3.1 Intra Prediction . . . 10

3.3.2 Core Transform . . . 12

3.3.3 Entropy Encoding . . . 13

3.4 Output Parser . . . 13

4 Experiment and Results 15

4.1 Experimental Setup . . . 15

4.1.1 Software Testbench . . . 15

4.1.2 Hardware FPGA . . . 15

4.1.3 Hardware Physical . . . 16

4.2 Performance Result . . . 17

4.2.1 Resource Utilization . . . 17

4.2.2 Latency . . . 18

4.2.3 Reconstructed Image Quality . . . 19

4.2.4 Compression Ratio . . . 21

4.3 Design Challenges . . . 21

4.3.1 Intra Clock Timing Closure . . . 21

4.3.2 Inter Clock Signal Integrity 22
4.3.3 Pipeline Flow Control 22

5 Conclusion 23

6 Future Work 24

1 Introduction

Matrix compression is essential in various applications to reduce the data required for transmitting or storing information efficiently. A common example is video compression, widely used in streaming and storage. Traditionally, video encoding and decoding are performed by high-level languages on CPU/GPU-based systems. However, these systems face limitations, including reduced flexibility, limited scalability, and less predictable performance under varying loads. In contrast, FPGA/ASIC-based systems provide lower latency, greater energy efficiency, and a more self-contained design that many applications demand for high-speed, real-time data handling.

In addition to video compression, this project introduces pre-CPU processing to reduce data transmission stress between high-resolution or high-frame-rate sensors and processors. By offloading computational tasks directly onto the hardware, the system minimizes the bandwidth required for data transfer, enabling efficient operation in bandwidth-constrained scenarios. This approach has significant implications for applications such as medical imaging, satellite imaging, drone imaging, and mobile devices designed for slow-motion recording. The integration of pre-processing capabilities highlights the potential for versatile FPGA implementations in diverse, high-performance environments.

This study explores the feasibility of a hardware-based solution for video compression, emphasizing flexibility and scalability. By leveraging FPGA development techniques, this project implements critical components of video compression, such as discrete cosine transform, entropy encoding, and context-based intra-frame prediction. The design also integrates scalable, modular memory architectures to support the complex storage requirements of large-scale encoding.

This study aims to contribute to the field by demonstrating how FPGAs can be utilized to address key challenges in hardware-based compression. By focusing on modularity and adaptability, this study aims to offer insights into more accessible and flexible FPGA design approaches, potentially paving the way for advanced development in hardware-accelerated data processing solutions.

2 Literature Review

The implemented algorithm is inspired by the H.264 standard developed by the Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T)[1]. It incorporates key components such as Context-Adaptive Variable Length Coding (CAVLC) for entropy coding and uniform scalar quantization for the quantization process.

In this approach, a large matrix is divided into smaller 4x4 matrices, commonly referred to as macroblocks (MBs). Standard video compression techniques typically support macroblock sizes of 4x4, 8x8, and 16x16 to accommodate different resolution and compression needs. However, for this project, we focus exclusively on 4x4 macroblocks to simplify the implementation and emphasize fine-grained compression. Each macroblock then undergoes several distinct compression stages, as illustrated in Figure 1.

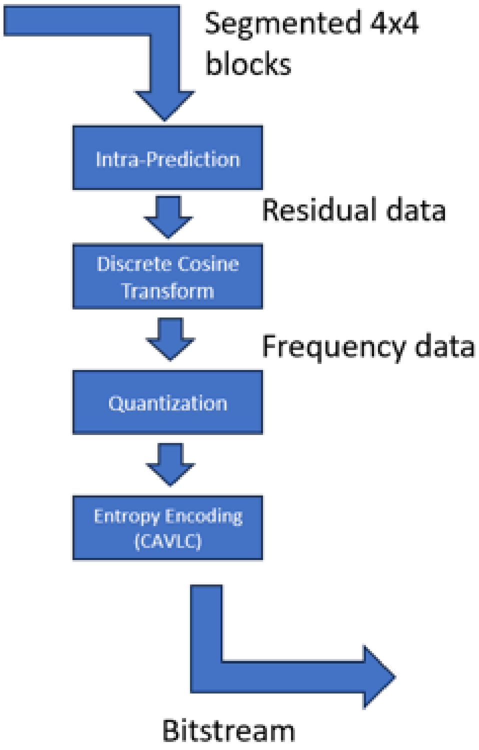


Figure 1: algorithm flow chart for each MB

2.1 Intra-prediction

This is the first stage of the compression process, which exploits spatial redundancy between macroblocks by predicting pixel values based on neighboring pixels. The predicted macroblock is then subtracted from the original resulting in a residual block that is encoded to reduce wasted space in the encoded data in the form of redundancy. Several prediction modes are utilized, and the mode that produces the smallest residual is selected for encoding.

- DC Prediction: Estimates the pixel based on the average of the value of available neighboring pixels, resulting in a uniform MB.
- Horizontal Prediction: Estimates the pixel based on the left neighbor pixel, effective when the MB features horizontal textures.
- Vertical Prediction: Estimates the pixel based on the top neighbor pixel, effective when the MB features vertical textures.

2.1.1 Hardware Optimization

There are 4 additional 4x4 prediction modes and several 8x8 and 16x16 prediction modes available in the standard[1]; however, due to the complexity of the operation and the timing stress it would introduce to the decision-making module since it is based on minimal sum, logic, and net delay almost directly scale with the amount of residual data to be decided.

Furthermore, the standard specified that the neighboring pixel used for prediction should be retrieved by decoding previously encoded macroblocks since the decoder would only have access to reconstructed data. However, this would limit the pipeline to sequential processing, which would not be a huge issue for CPU/GPU-based solutions, this would negate one of the most significant strength of FPGA: parallel processing. Thus, it is decided to cache the original pixel value and pass it to neighboring pixels, making the encoding process for each macroblock self-contained, enabling a parallel pipeline. Additionally, this would simplify the encoding process, which would reduce the critical path delay for macroblock processing, increasing throughput.

2.2 Discrete Cosine Transform

The next stage after the intra-prediction is the Discrete cosine transform. This algorithm converts the residual block data from the spatial domain to the frequency domain. The idea is to concentrate the signal energy into a smaller number of coefficients, or in other words, generate a matrix that consists of mostly zeros which allow a high compression ratio in the latter stages of the pipeline (entropy encoding)

$$X(u,v) = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{3}\sum_{y=0}^{3} x(x,y)\cos\left[\frac{(2x+1)u\pi}{8}\right]\cos\left[\frac{(2y+1)v\pi}{8}\right] \tag{1}$$

where

$$\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } u = 0, \\ 1 & \text{otherwise.} \end{cases} \tag{2}$$

Where X(u,v) is the DCT coefficient at frequency indices(u,v), x(x,y) is the pixel value at position indices (x,y) in the residual block, a(u) and a(v) are scaling factor which depends on the indices.

2.2.1 Hardware Optimization

However, detailed cosine calculation is typically not feasible in systems that are time-critical or source-limited due to the nature of floating point operation. Thus it is typical to utilize an integer approximation. A Transform matrix is derived to approximate the floating point operation which is given by $C_{4\times4}$ here[2].

$$C_{4\times4} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & -1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \tag{3}$$

By applying integer arithmetic—through additions, subtractions, and shifts—it simplifies the implementation, allowing the use of efficient hardware components such as adders and shift registers. This integer-based approach ensures the DCT can be computed quickly and with fewer resources, making it suitable for embedded and real-time systems.

2.3 Quantization

The DCT coefficients would need to be quantized before entering entropy encoding. This step is designed to significantly reduce the amount of data needed to be processed by discarding less important frequency information. Though this would result in reduced precision, it is done in a way that is not humanly perceptible. This procedure is outlined in Equation 3.

$$Q(u,v) = \left\lfloor \frac{X(u,v)}{Q_{\text{matrix}}(u,v)} + 0.5 \right\rfloor \tag{4}$$

where Q_{matrix} is the quantization matrix specified by the standard [1]. The division is element-wise, and rounding is done by adding 0.5 before applying the floor function.

2.4 Entropy Encoding

The entropy encoding method chosen here is the Context-Adaptive Variable Length Coding (CAVLC) is chosen from the H.264 standard [1]. The CAVLC is used to compressed the quantized DCT coefficients in a way that adapts to the data characteristics.

- The algorithm starts by scanning the 4x4 matrix in a zigzag pattern shown in Figure 2. This allows the algorithm to rearrange the matrix into a 1D list. This ordering groups the significant (non-zero) coefficient at the beginning while placing the trailing zeros at the end.

- The CAVLC then counts the number of non-zero coefficients and the number of trailing ones in the list. Knowing the position and number of trailing ones enables CAVLC to use shorter codes for common cases, especially in smooth or flat regions with few non-zero coefficients.

- The non-zero coefficients are encoded using a combination of the number and the value of the non-zero coefficient. This format skips the zero values, reducing the data size significantly especially for matrices that have many zeros.

- The algorithm dynamically adjusts its variable-length coding tables based on the local context, such as the number of non-zero coefficients in neighboring blocks. This context adaptation allows the algorithm to assign shorter codes to frequently occurring values, improving compression efficiency.

- Finally, the algorithm assigns shorter variable-length codes to smaller, more frequent coefficient levels while using longer codes for larger values.

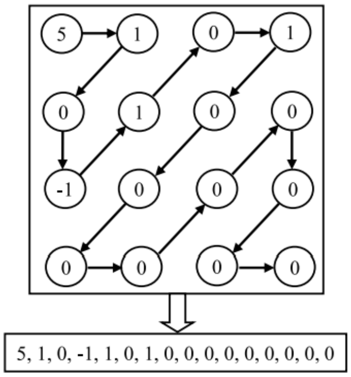


Figure 2: Zigzag ordering for flattening matrix [3]

2.4.1 Hardware Optimization

Typically, CAVLC requires meta information about previous and neighboring encoded macroblocks to achieve the best compression ratio, thus similar to intra-prediction, typical operation would require sequential processing for the segmented macroblocks. However, unlike intra-prediction, where metadata could be substituted for other available data, the required meta information here is not readily available. Therefore, to make the processing and transmission of each macroblock self-contained, to enable parallel pipeline, and to simplify the implementation, it is decided to default all context input to zero, thus dropping the best case compression ratio to 8:1.

3 Technical Overview

The encoder consists of four main components: the data input interface, data output interface, memory controller, and computation unit. For this project, a PC serves as both the data input and output interface, combining them into a single module. However, these interfaces can be reconfigured in practical applications to accommodate specific sensor control and I/O requirements. The FPGA-to-PC interface is implemented using the OpalKelly USB development kit. Data is initially stored in onboard memory, accessible by both the interface and memory controller. Once data is passed to the computation unit, all temporary data and metadata are stored into register cache. The different stages of the computation pipeline incorporate backward controlled stall mechanisms, where the FSM of the current stage could pre-calculate one set of data and pause at the fetch stage for the next iteration while downstream is stalled either by output I/O or algorithm. Additionally, to minimize stalling events, except entropy encoding due to the nature of the algorithm, all stages of the pipeline are designed such that an MB takes 4 cycles to be processed. This way data throughput across processing blocks are matched.

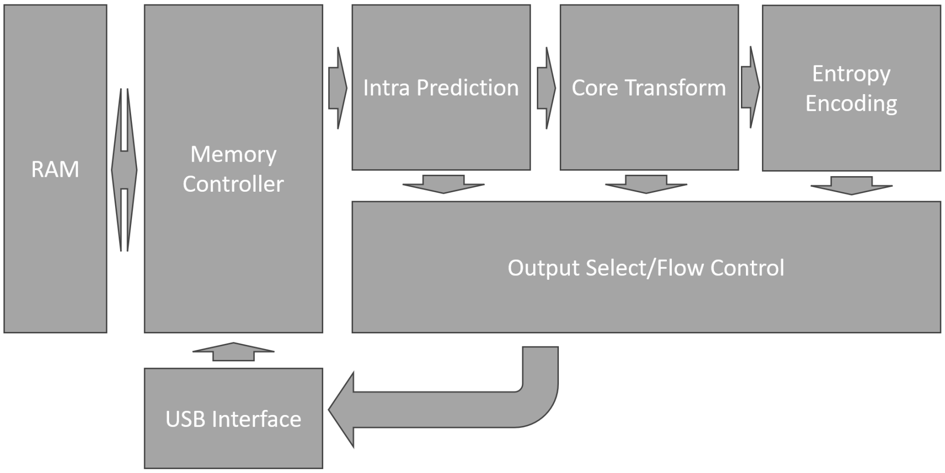


Figure 3: System Architecture

3.1 Interface

The USB interface is developed using the Opalkelly USB 3.0 development kit[4]. To ensure safe data clock domain crossing, incoming and outgoing data are routed through FIFOs implemented with BRAM, chosen for its efficient use of LUT and FF resources. A block-throttled pipeline manages the FIFO, with a ready signal triggering the transmission of a specified data volume. Control signals and status reports are transmitted via asynchronous registers, where clock domain crossings are manually handled using double-flopping. While this technique is typically unsuitable for multi-bit signal synchronization, it is adequate here because the control and status signals are designed to be single-bit and level-sensitive, making cross-bit synchronization unnecessary.

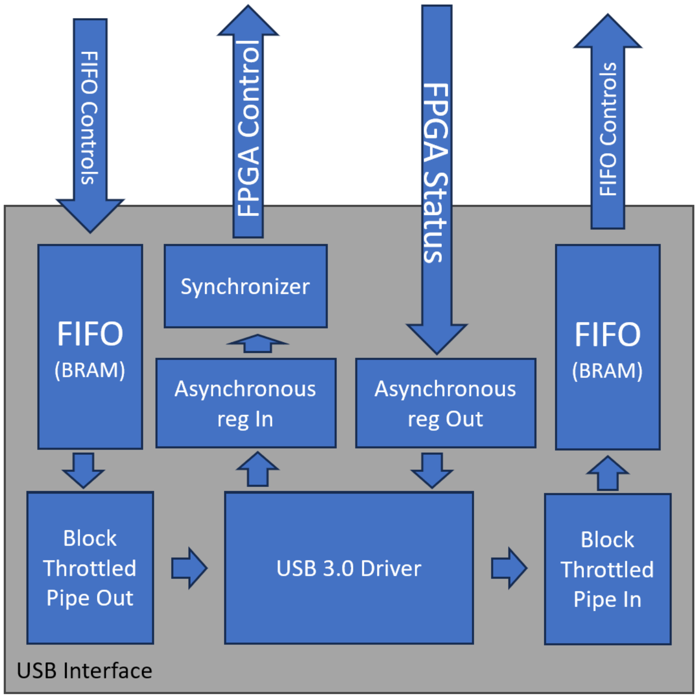


Figure 4: USB interface architecture

3.2 Memory Controller

The primary function of the memory controller is to map the input data stream to specific memory addresses, and retrieve the datastream when requested to the next stage of the pipeline. For each frame, the image (256x256) is received in its entirety before starting processing, with each pixel value encoded in datatype uint8. The data passing in is flattened using scanning order. The integrated Input Parser would distribute the data across the RAM instances in 2x2 blocks per address. Similarly, data are retrieved in 2x2 blocks by the output parser when prompted, which means a full 4x4 macroblock requires 4 transmissions to be extracted, introducing pipeline stall.

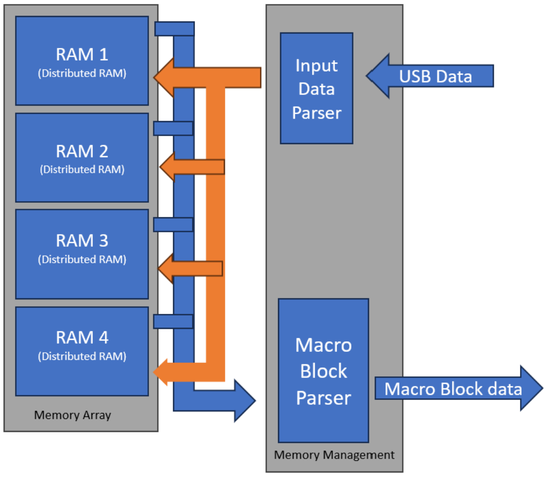


Figure 5: memory controller block diagram

3.2.1 RAM Array

The RAM module consists of 4 ”simple dual-port distributed memory” as defined by Xilinx Vivado distributed memory generator[5], which means that the memory instances have dedicated read and write ports, with independent addressing. Though the pipeline is designed such that simultaneous reading and writing is not a possibility, the independent addressing allows us to prevent accessing to that same register (address) across different procedural blocks, preventing race conditions.

Distributed RAM is selected over Block memory primarily to reduce pipeline stall caused by accessing. Typically, BRAM access would require 2 to 3 clock cycles for output to be stabilized, while Distributed RAM allows single-cycle access when timing closure is satisfied.

Additionally, four distributed RAM instances are used instead of a single centralized RAM instance for several important reasons. First, this design allows parallel access to multiple pixel data, significantly reducing pipeline stalls and improving throughput. Second, it alleviates intra-clock timing closure challenges. Distributed RAM, which is primarily implemented using LUTs and LUTRAM (combinational logic), can experience increased fanout and logical delays with longer addressing paths. These delays can lead to increased slack times and potential timing failures. By using multiple RAM instances, the design minimizes these issues and ensures more efficient operation.

3.3 Computational Unit

The computational unit is designed to process each macroblock (MB) through a series of compression steps, including prediction, transformation, quantization, and entropy encoding. The unit's main components are the predictor, Discrete Cosine Transform (DCT), quantizer, and entropy encoder. The predictor module is responsible for reducing spatial redundancy by estimating pixel values based on neighboring macroblocks; it includes four prediction modes as individual sub-components—each tailored to specific spatial patterns (such as DC, horizontal, vertical, and planar prediction). A top-level decision-making component selects the optimal prediction mode based on the resulting residuals, aiming to minimize data for encoding. Following prediction, the DCT module converts the residuals from spatial to frequency domain, where the quantizer then reduces data precision to emphasize visually significant information. The entropy encoder compresses the quantized coefficients into a compact bitstream. Throughout the process, the computational unit leverages distributed memory to store temporary data, ensuring quick access and efficient data handling across components.

3.3.1 Intra Prediction

The Intra Prediction module includes three modes: DC, Horizontal, and Vertical. Each mode is implemented independently. The DC mode requires four clock cycles due to critical path delay constraints. To maintain uniformity in data flow and synchronization, the Horizontal and Vertical modes are also designed to process in four cycles, even though they do not have the same delay limitations. This consistent timing ensures seamless integration with downstream modules, simplifying flow control.

Additionally, a finite state machine (FSM) is incorporated to manage data requests and parsing from the memory controller. The FSM also handles storing necessary context data into a cache for future use, optimizing access efficiency. The cache is implemented using registers to enable rapid data retrieval. Furthermore, a residual select module operates based on control signals from the asynchronous controller. It either performs direct routing via manual selection or computes the least total sum over four clock cycles, ensuring efficient processing.

The Mode Select module is integrated within the intra-prediction unit and operates under three distinct modes, controlled directly by the PC. Users can manually select any of the three prediction modes or allow the module to automatically determine the optimal mode. The module's automatic selection is based on a minimum sum comparison, where the summation of residual blocks is processed in a pipeline spanning two clock cycles. An additional clock cycle is used for data sampling (latching), followed by another for sum comparison. This pipelined approach addresses timing closure challenges and aligns the module's throughput with upstream processes, effectively preventing pipeline stalls.

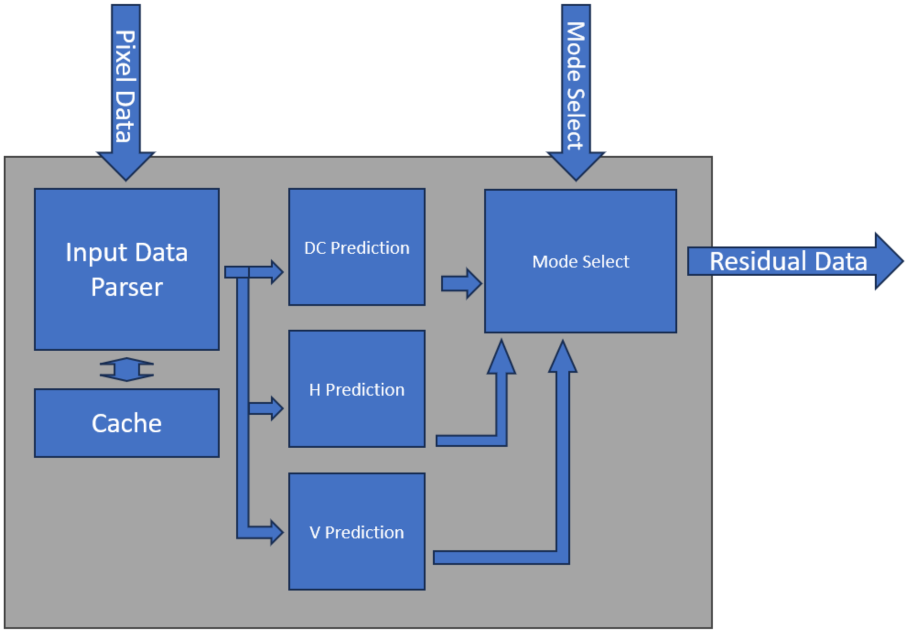


Figure 6: Intra Prediction Module

3.3.2 Core Transform

In the design, since there is no break points, DCT and quantize is grouped together and managed under core transform. The DCT module performs a 4x4 Discrete Cosine transform operation, however, due to hardware limitations, the computation did not strictly follow the mathematical formula. Instead, to avoid doing complicated floating point operations in hardware, an integer approximation is utilized as described by [2]

In practice, the 4x4 Integer DCT is often implemented using the butterfly algorithm, which breaks the 2D transform into two 1D stages—one applied to the rows and another to the columns. This reduces the complexity of direct matrix multiplication. After applying the butterfly algorithm, the resulting frequency coefficients can be quantized for compression. This method balances compression efficiency and computational cost, providing a practical and hardware-friendly approximation of the traditional 4x4 DCT.

Stage 1	**Stage 2**
$Y(0) = X(0) + X(3)$	$V(0) = Y(0) + Y(1)$
$Y(1) = X(1) + X(2)$	$V(2) = Y(0) - Y(1)$
$Y(2) = X(1) - X(2)$	$V(1) = Y(2) + (Y(3) \ll 1)$
$Y(3) = X(0) - X(3)$	$V(3) = Y(3) - (Y(2) \ll 1)$

Table 1: 4x4 forward 1D transform algorithm for H.264[2]

The quantization process in video compression follows the Discrete Cosine Transform (DCT) to reduce the precision of the DCT coefficients, enabling efficient data compression. After the DCT is applied, the resulting frequency coefficients are divided by a quantization matrix and then rounded to the nearest integer. The quantization matrix typically assigns lower precision to higher-frequency components, which often carry less perceptible visual information, thereby reducing their impact on image quality. This process significantly reduces the size of the data, but also introduces some loss of information, which is acceptable in video compression because the human eye is less sensitive to high-frequency details. The quantized coefficients are then further processed through entropy coding techniques, such as CABAC, to achieve high compression ratios.

3.3.3 Entropy Encoding

The Entropy Encoding unit, which implements a simplified version of CAVLC, is relatively straightforward in design. However, due to the variable-length nature of the algorithm's output, the processing time for each macroblock is unpredictable. This variability can cause FSM runaway conditions in the upstream processor if not properly managed. To address this, a backward-cascading stall signal is implemented. This mechanism allows the upstream processor to process one additional macroblock before stalling at the sampling stage, effectively mitigating timing and synchronization issues.

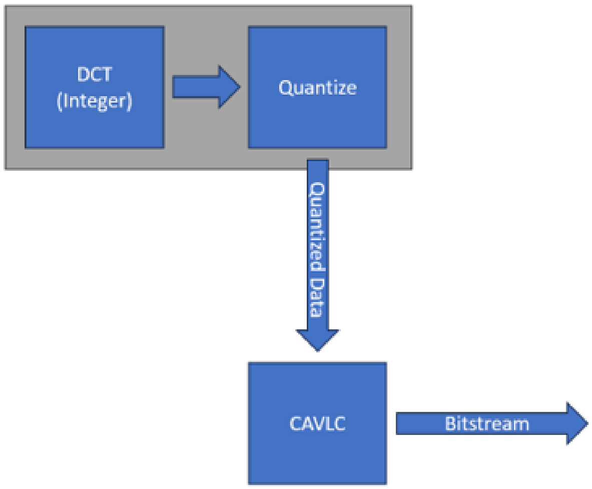


Figure 7: downstream blocks from intra-prediction

3.4 Output Parser

The output parser module is responsible for managing various output modes and ensuring the correct formatting and transfer of data streams. It supports different modes, including raw data dumps for USB and memory verification, intra data dumps for intra-prediction verification, core transform dumps, and the final encoded data stream. Since the USB interface is block-throttled, the module must ensure that data is processed in blocks of 256 or 1024 bytes. To trigger USB transmission, the FIFO buffer must contain this amount of data, so when the upstream processor signals the completion of a frame, the module pads the data with zeros. The module parses incoming data in 16-byte chunks from the intra and quantize modules and splits them into 4-byte segments to match the USB data width.

For entropy encoding, the module processes the raw data byte by byte, concatenating the chunks as needed. If the final 32-byte data chunk is not fully populated, the module zero-pads it to ensure the correct block size for USB transmission. Mode selection in this module is similar to the intra mode selection, as discussed previously. The module also generates and transmits metadata for each macroblock (MB) during entropy encoding, ensuring that each MB is properly encoded and can be decoded accurately. This process guarantees that the data stream is appropriately formatted and meets the specific output requirements for all supported modes.

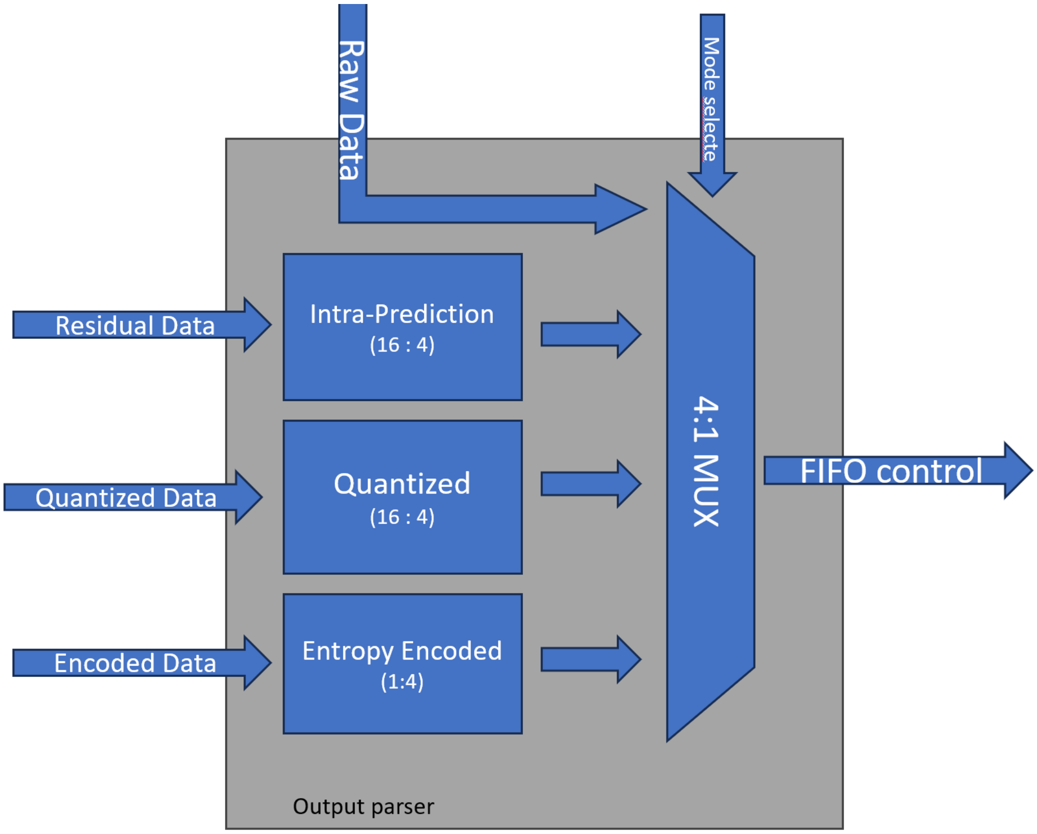


Figure 8: Output Parsar block diagram

4 Experiment and Results

This section provides an overview of the experiment and the results obtained during the project. It covers the setup used for testing, including the hardware configuration of the FPGA and the software testbench implemented in Python. The section also presents the performance results, discussing key aspects such as latency, resource utilization, compression ratio, and image quality. Additionally, it highlights the challenges faced throughout the project, including timing issues, handling asynchronous data, and managing pipeline flow control, along with the solutions implemented to address these challenges.

4.1 Experimental Setup

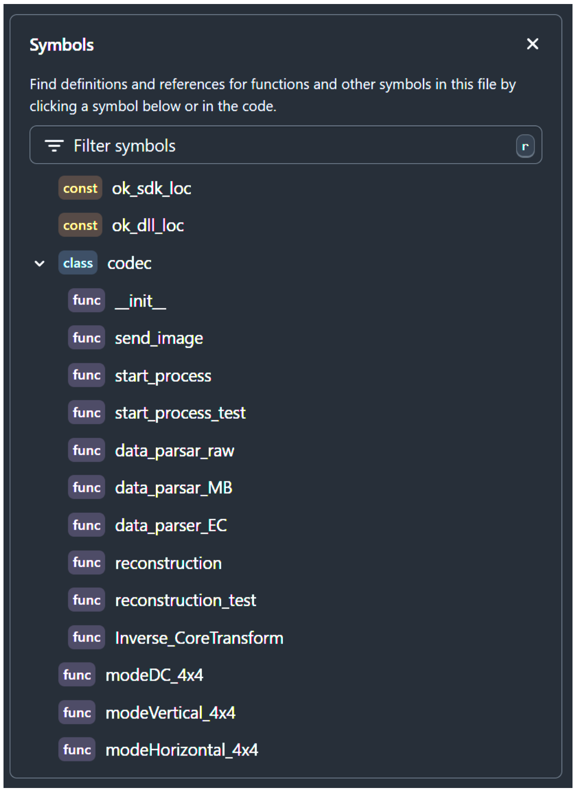
This experiment uses the Xilinx Artix-7 series FPGA to evaluate the hardware design, with a PC functioning as the data input simulator and decoder for the encoded data stream. Python is employed on the PC for its ease of implementation, as the performance of the PC component is not a critical factor in this study, which focuses primarily on hardware.

4.1.1 Software Testbench

The software testbench for this project was implemented on a PC using Python, chosen for its ease of implementation due to the non-critical performance requirements of the PC. The testbench simulates system input and output by converting an image to grayscale, streaming it to the FPGA, and reconstructing the returned data through various stages. To validate the proposed algorithm, an open-source Python library was used, which provided essential decoding functions such as inverse quantization and inverse CAVLC[6], borrowed from the H.264 standard. Although the implementation deviates from H.264 in several ways, these functions ensured compatibility and efficiency.

4.1.2 Hardware FPGA

Due to the limited resources of the Artix-7 FPGA family and the potential timing closure challenges associated with high resource usage, the test unit is configured to instantiate only a single lane of the pipeline. The use of distributed RAM, chosen to



Symbols ✕

Find definitions and references for functions and other symbols in this file by clicking a symbol below or in the code.

Filter symbols

- const ok_sdk_loc
- const ok_dll_loc
- class codec
  - func __init__
  - func send_image
  - func start_process
  - func start_process_test
  - func data_parsar_raw
  - func data_parsar_MB
  - func data_parser_EC
  - func reconstruction
  - func reconstruction_test
  - func Inverse_CoreTransform
- func modeDC_4x4
- func modeVertical_4x4
- func modeHorizontal_4x4

Figure 9: Python testbench structure

minimize pipeline stalls, results in a high report of LUT and LUT RAM utilization. The pipeline itself consumes approximately 4000 LUTs for instantiation. Additionally, the USB interface and memory controller in the system are specific to the project and were co-developed with the software testbench. This integration, while necessary for functionality, inadvertently introduced additional latency and bottlenecks into the system.

4.1.3 Hardware Physical

The experiment is conducted using the Opal Kelly XEM7310-A75 FPGA development board, which, in addition to the FPGA itself, includes a USB 3.0 chipset, hardware drivers, and a software API for seamless integration. Communication between the FPGA and Integrated Logic Analyzer (ILA) is handled through the JTAG connector, while all other data is transmitted via the USB interface. The onboard FPGA is part of the Xilinx Artix-7 family, a fairly low-end FPGA suitable for functional validation. However, due to its limited processing power and resources, it is not ideal for performance benchmarking, and such tests would not yield meaningful results.

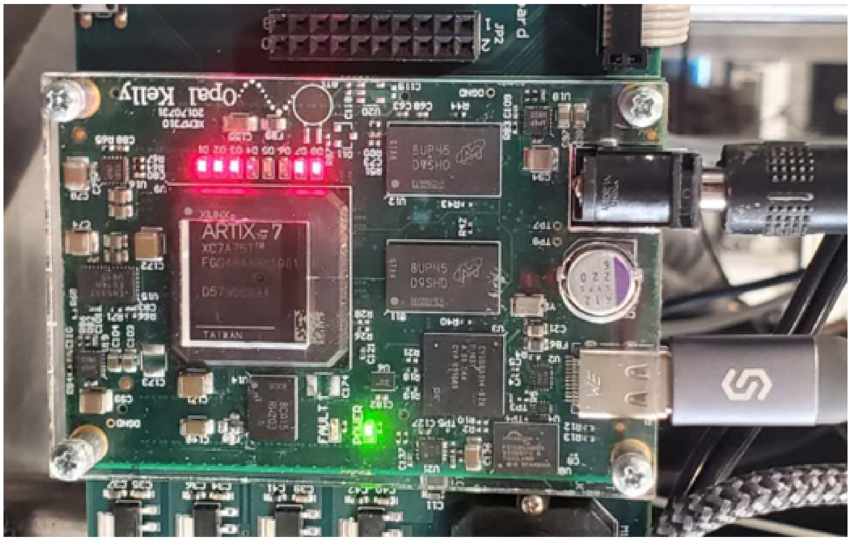


Figure 10: Testbench Hardware

4.2 Performance Result

This section provides an overview of the system's resource utilization, latency, compression ratio, and reconstructed image quality. It outlines key performance metrics and highlights the effectiveness of the design in terms of FPGA resource usage, processing speed, and the quality of the compressed and reconstructed images. The following subsections present detailed analyses of each aspect, focusing on how the design performs under various conditions, the challenges encountered, and the compromises made in hardware implementation.

4.2.1 Resource Utilization

As previously mentioned, the pipeline itself utilizes approximately 4000 LUTs. However, to produce more meaningful latency results, distributed RAM was employed, which significantly increased the usage of LUTs and LUTRAM. Additionally, a 4096-depth Integrated Logic Analyzer (ILA) was instantiated for status monitoring, consuming a few BRAMs. Despite these additions, the design remains within the resource constraints of the Artix-7 FPGA. The detailed resource utilization report is provided below, highlighting the balance achieved between functionality and resource efficiency.

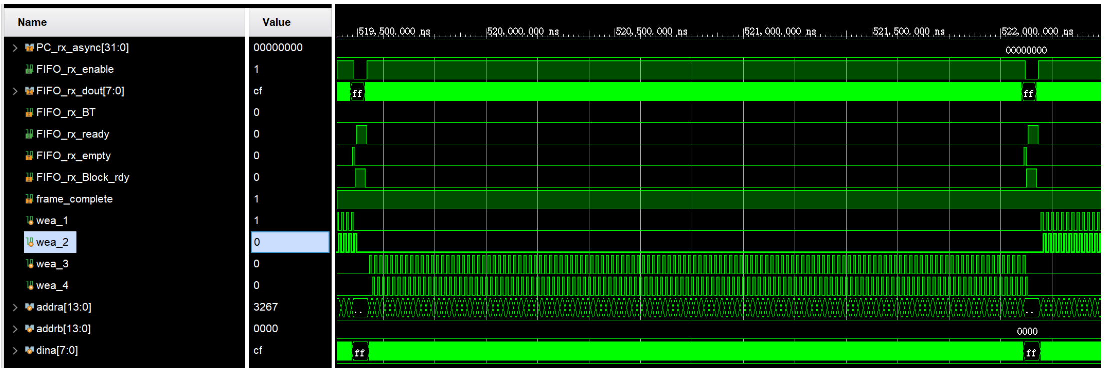
Resource	Utilization	Available
LUT	18382	47200
LUTRAM	12494	19000
FF	4317	94400
BRAM	44	105
IO	51	285
BUFG	4	32
MMCM	1	6

Table 2: Resource Utilization and Availability

This resource allocation demonstrates the effectiveness of the design in leveraging FPGA resources for functional validation while maintaining sufficient margins for potential future optimizations.

4.2.2 Latency

The processing speed of the system is influenced by several factors, including the availability of the USB port, the USB transmission speed, and the resolution of the input image. For testing purposes, a 256 × 256 resolution image was used, along with the Opal Kelly USB 3.0 driver. The USB protocol limits both the image loading and full image processing time to approximately 680 μs per image. A timing diagram for the transmission of an entire row of pixel data is shown below.



Name	Value
PC_rx_async[31:0]	00000000
FIFO_rx_enable	1
FIFO_rx_dout[7:0]	cf
FIFO_rx_BT	0
FIFO_rx_ready	0
FIFO_rx_empty	0
FIFO_rx_Block_rdy	0
frame_complete	1
wea_1	1
wea_2	0
wea_3	0
wea_4	0
addra[13:0]	3267
addrb[13:0]	0000
dina[7:0]	cf

Figure 11: Timing Diagram for transmitting data

Memory access currently constitutes the most significant bottleneck in the pipeline, with USB output occasionally stalling the system. This results in a latency of 1.2 to 1.5 milliseconds for processing and transmitting an image already loaded into the

FPGA. However, as the design is intended to be scalable and modular, these overall latency figures are less critical. More meaningful is the observation that each macroblock requires approximately 20 clock cycles to pass through the computation unit. Since three macroblocks can be processed simultaneously at different stages, running the pipeline at 200 MHz results in a latency of 100 nanoseconds per macroblock and a throughput of 30 MB/μs per lane of the pipeline.

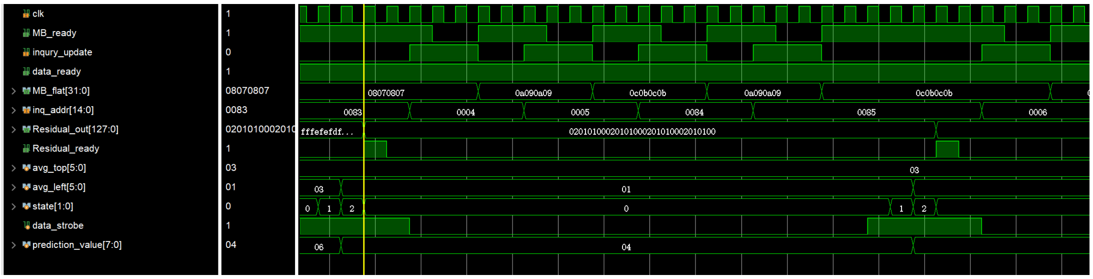
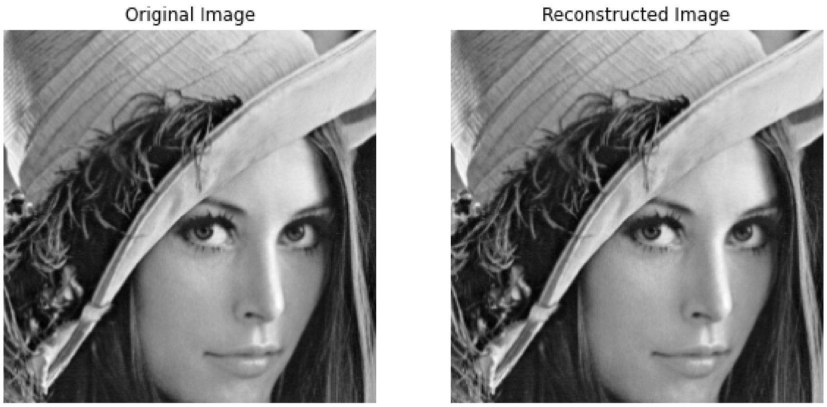


Figure 12: Timing Diagram of intra prediction (without decision) of a macroblock

4.2.3 Reconstructed Image Quality

To validate the proposed algorithm, a variety of standard stock images were used as input data for testing. Observations indicate that the reconstructed images generally retain their visual content, making it difficult for the human eye to discern any loss of information or precision. For example, as shown below with the ”Lenna” stock image, the differences between the original and reconstructed images are virtually indiscernible.


Original Image

Reconstructed Image

Figure 13: Raw and decoded image of stock image ”lenna”

However, since the algorithm uses the original pixel values instead of reconstructed pixels during compression, sharp transitions in the image pose a challenge for accurate encoding. This limitation results in visible artifacts, such as white spots in high-frequency regions. An example of this can be seen in the whiskers of the stock image "Kitten," as shown below.

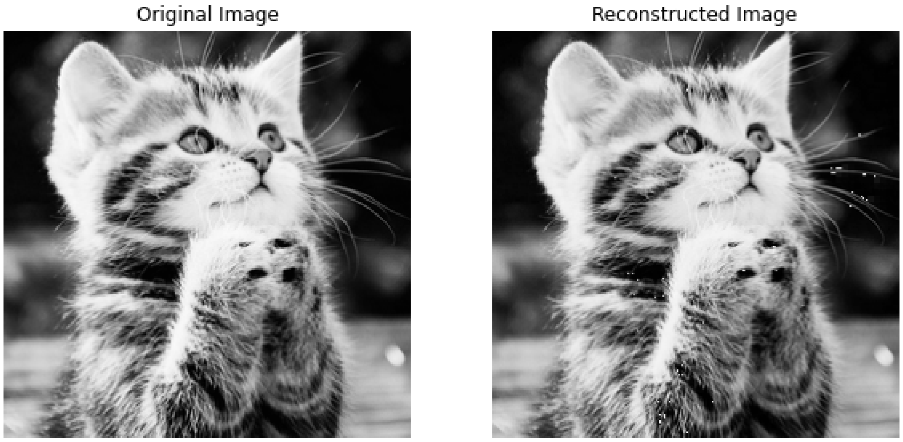


Figure 14: Raw and decoded image of stock image "kitten"

This issue becomes particularly pronounced in areas where the color transitions from dark regions. Various factors contribute to this effect, as demonstrated in the "Peppers" stock image.

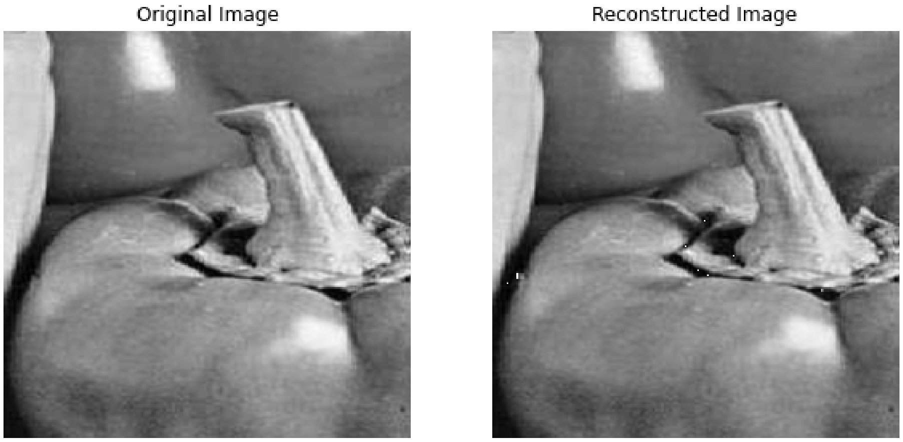


Figure 15: Raw and decoded image of stock image "peppers"

4.2.4 Compression Ratio

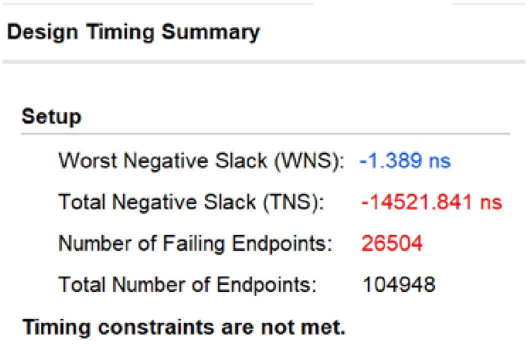
Due to compromises made in the hardware implementation, the system achieves a best-case compression ratio of 8:1, typically observed when compressing uniform color images. On average, the compression ratio hovers around 4:1 to 3:1, with lower ratios of 2:1 to 1:1 occurring in images dominated by unorganized, high-frequency content.

4.3 Design Challenges

The design process encountered several key challenges, including intra-clock timing closure, inter-clock signal/data crossing, and pipeline flow control. These issues arose due to hardware constraints, integration complexities, and synchronization between components, requiring careful solutions to maintain system performance.

4.3.1 Intra Clock Timing Closure

Our initial design encountered significant negative slack, primarily due to a centralized distributed RAM unit. The read address register experienced substantial net delay (4ns out of the 5ns total available), leading to severe timing failures. We addressed this by implementing several solutions: first, we replaced the single centralized RAM with four distributed RAM units, reducing address length and the number of cells accessing the address, which helped alleviate the net delay. Additionally, we constrained the tool to prioritize this critical net during physical optimization and layout by setting slack priority. Finally, we further reduced slack by adjusting the implementation strategy.



Design Timing Summary

Setup

Worst Negative Slack (WNS):	-1.389 ns
Total Negative Slack (TNS):	-14521.841 ns
Number of Failing Endpoints:	26504
Total Number of Endpoints:	104948

Timing constraints are not met.

Figure 16: timing failure caused by distributed ram addressing

4.3.2 Inter Clock Signal Integrity

Our initial design used asynchronous registers in the USB driver to transmit data, with a separate register serving as the handshaking signal. We employed level-sensitive triggering to sample the data, aiming to minimize the impact of asynchronous transmission. However, this approach proved ineffective, frequently resulting in state machine runaway issues. To resolve this, we switched to a dual-clock FIFO with two-stage synchronization by Xilinx[7], ensuring safe handling of data across clock domains.

4.3.3 Pipeline Flow Control

Initially, it was assumed that flow control would not be an issue, as most processors in the pipeline were flow rate-matched. However, due to the USB driver's inability to read all data and the unpredictable nature of entropy encoding, issues such as state machine runaways, buffer overflows, and data loss arose. To address these challenges, a backward-controlled stalling mechanism was introduced, as described above. In this system, the stall signal is cascaded through the pipeline from the stalling endpoint, with downstream components being stalled by not receiving the ready signal.

5 Conclusion

Matrix compression remains a cornerstone of efficient data handling across various applications, from video streaming to real-time sensor processing. This study successfully demonstrated the design and implementation of a hardware-accelerated compression system leveraging the capabilities of the Xilinx Artix-7 FPGA. By employing a modular approach to video encoding, including key techniques such as intra-prediction, Discrete Cosine Transform (DCT), quantization, and entropy encoding, this project achieved significant advancements in scalability, flexibility, and real-time processing.

The experimental results underscored the system's potential for diverse applications, particularly in scenarios requiring high-throughput and low-latency solutions. While current implementations focused on core functionalities, such as handling macroblock compression and optimizing memory usage, the outcomes align with the goal of enabling efficient data processing in constrained environments. Furthermore, this work highlights how FPGA-based systems can outperform traditional CPU/GPU approaches in terms of energy efficiency and predictable performance, offering promising directions for hardware-accelerated video compression technologies.

6 Future Work

Future work for this project encompasses several promising avenues to enhance both functionality and performance. A key focus is the integration of inter-frame prediction, which would capitalize on temporal redundancies between consecutive frames to achieve significantly higher compression ratios. This addition would expand the encoder's capabilities to handle more complex video content with improved efficiency. Exploring alternative entropy encoding methods, such as Arithmetic Coding or CABAC, could further optimize compression performance by adapting to diverse data characteristics.

Another crucial direction involves implementing parallel processing to enable simultaneous handling of multiple macroblocks. This approach would substantially improve throughput and reduce overall latency, making the system more suitable for real-time applications like live streaming or autonomous systems. Testing the design on higher-end FPGA platforms, such as Xilinx UltraScale+ or Intel Stratix, could provide valuable insights into scalability and performance improvements, particularly for larger resolutions and higher frame rates.

Additionally, integrating advanced memory architectures and addressing pipeline synchronization challenges could further streamline data flow, reducing bottlenecks and enhancing system stability. By pursuing these enhancements, the system could evolve into a robust, adaptable solution for next-generation video compression demands, establishing a competitive edge in the hardware-accelerated data processing landscape.

References

[1] I. T. Union, “Recommendation itu-t h.264: Advanced video coding for generic audiovisual services,” International Telecommunication Union, Tech. Rep., 2003, version 1. [Online]. Available: https://www.itu.int/rec/T-REC-H.264

[2] A. T. Bunji Antoinette Ringnyu, “Implementation of forward 8x8 integer dct for h.264/avc frext,” *The Eurasia Proceedings of Science, Technology, Engineering Mathematics (EPSTEM)*, vol. 1, pp. 353–358, 2017.

[3] G.-L. J. G.-L. J. e. a. Fuentes-Alventosa, A., “Cavlcu: an efficient gpu-based implementation of cavlc,” *J Supercomput 78*, vol. 7556–7590 (2022), 2022. [Online]. Available: https://link.springer.com/article/10.1007/s11227-021-04183-8#citeas

[4] O. K. Incorporated, *FrontPanel HDL Interface*, n.d., accessed: December 8, 2024. [Online]. Available: https://docs.opalkelly.com/fpsdk/frontpanel-hdl/

[5] AMD, “Distributed Memory Generator User Guide: DS322,” 2024. [Online]. Available: https://docs.amd.com/v/u/en-US/dist_mem_gen_ds322

[6] Wattery, “Pycodec,” 2019. [Online]. Available: https://github.com/Watterry/PyCodec/tree/master

[7] AMD, “FIFO Generator User Guide: DS317,” 2024. [Online]. Available: https://docs.amd.com/v/u/en-US/fifo_generator_ds317