

TURNING VISUAL NOISE INTO HARDWARE EFFICIENCY: SYSTEMS OF VIEWER
AND CONTENT AWARE POWER-QUALITY SCALABLE EMBEDDED MEMORIES
WITH ECC-ADAPTATION FOR BIG VIDEOS AND DEEP LEARNING

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Ali Ahmad Haidous

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Electrical and Computer Engineering

July 2021

Fargo, North Dakota

North Dakota State University
Graduate School

Title

TURNING VISUAL NOISE INTO HARDWARE EFFICIENCY:
SYSTEMS OF VIEWER AND CONTENT AWARE POWER-QUALITY
SCALABLE EMBEDDED MEMORIES WITH ECC-ADAPTATION FOR
BIG VIDEOS AND DEEP LEARNING

By

Ali Ahmad Haidous

The Supervisory Committee certifies that this *disquisition* complies with North Dakota
State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Danling Wang

Chair

Jacob Glower

Dharmakeerthi Nawarathna

Sumitha George

Simone Ludwig

Approved:

July 19, 2021

Date

Benjamin D. Braaten

Department Chair

ABSTRACT

Mobile devices, such as smart phones, are being increasingly utilized for watching videos. Video processing requires frequent memory access that consume a significant amount of power due to large data size and intensive computational requirements. This limits battery life and frustrates users. Memory designers are focused on hardware-level power-optimization techniques without consideration of how hardware performance influences viewers' actual experience. The human visual system is limited in its ability to detect subtle degradations in image quality. For example, under conditions of high ambient illumination – such as outdoors in direct sunlight – the veiling luminance (i.e., glare) on the screen of a mobile device can effectively mask imperfections in the image. Under these circumstances, a video can be rendered in lower than full quality without the viewer being able to detect any difference in quality. As a result, the isolation between hardware design and viewer experience significantly increases hardware implementation overhead and power consumption due to overly pessimistic design margins, while integrating the two would have the opposite effect.

In this dissertation, viewer-awareness, content-awareness, and hardware adaptation are integrated to achieve power optimization without degrading video quality, as perceived by users. Specifically, this dissertation will (i) experimentally and mathematically connect viewer experience, ambient illuminance, and memory performance; (ii) develop energy-quality adaptive hardware that can adjust memory usage based on ambient luminance to reduce power usage without impacting viewer experience; (iii) design various mobile video systems to fully evaluate the effectiveness of the developed methodologies; and (iv) provide an overview of bleeding edge related area research then push the boundary further using the novel techniques discussed to achieve optimized quality, silicone area overhead, and power reduction in video memory.

ACKNOWLEDGEMENTS

Bi-smi llāhi ar-raḥmāni ar-raḥīm, “In the name of Allah (God), the Most Gracious, the Most Merciful.” Who has blessed me with opportunity and gave me the health and wherewithal to complete this journey. As He has said in the Quran, “Whatever He wills occurs without resistance, and whatever He does not will, never occurs.”

To first and foremost, Na Gong, I sincerely and with all gratitude thank you. Your encouragement, patience, time, guidance, and knowledge helped me immensely in my research. To Scott Smith, my utmost appreciation for your leadership and mentorship. Your leading questions lead to many great results and answers that evolved my research capabilities. To my lab partners and especially Hritom Das, William Oswald, and Yifu Gong, for their assistance in supporting experiments and simulations. Thank you to my supervisory committee for their assistance in presentations and feedback during my doctoral candidacy: Danling Wang, Jacob Glower, Dharmakeerthi Nawarathna, Sumitha George, and Simone Ludwig.

I am grateful for the National Science Foundation for their grant that made this research possible.

In reference to IEEE copyrighted material, which is used with permission in this dissertation, the IEEE does not endorse any of North Dakota State University's products or services. Internal or personal use of this material is permitted.

DEDICATION

Dedicated to my parents, Mama Huda and Baba Ahmad, who trekked against adversity, sacrificed their wellbeing, and contradicted all odds in America for my siblings and I to pursue our dreams. I will never be able to repay you, but I hope that I can at least make you proud.

TABLE OF CONTENTS

ABSTRACT	iii
ACKNOWLEDGEMENTS	iv
DEDICATION	v
LIST OF TABLES	xii
LIST OF FIGURES	xiii
LIST OF ABBREVIATIONS.....	xvii
CHAPTER 1. INTRODUCTION	1
1.1. Motivation	1
1.2. Viewer-Aware Bit-Truncation	2
1.3. Self-Correcting Memory Throughout Voltage Scaling.....	2
1.4. Smart Dynamic Memory Management in Video Decoder Processes	3
CHAPTER 2. CONTENT-ADAPTIVE MEMORY FOR VIEWER-AWARE ENERGY- QUALITY SCALABLE MOBILE VIDEO SYSTEMS	4
2.1. Introduction	4
2.2. Related Work.....	8
2.3. Influence of Video Content on Viewer’s Experience	9
2.3.1. Mobile Video Memory System	9
2.3.2. Influence of Video Content on Viewer’s Experience in the Presence of Hardware Noise	14
2.3.2.1. Traditional PSNR Metric	14
2.3.2.2. Video Macroblock Variance Analysis	16
2.4. Modeling Process	19
2.4.1. Subjective Testing Procedure for Data Collection	19
2.4.2. Modeling Process	20
2.4.2.1. Decision Tree Model	20

2.4.2.2. Logistic Regression Model	21
2.5. Quality Optimized Bit Truncation Design	23
2.5.1. Quality Optimized Bit Truncation.....	23
2.6. Content-Adaptation Video Memory Design	25
2.7. Experimental Results.....	27
2.7.1. Speed	27
2.7.2. Layout.....	28
2.7.3. Power Savings	29
2.7.4. Video Quality	30
CHAPTER 3. FLEXIBLE LOW COST POWER-EFFICIENT VIDEO MEMORY WITH ECC-ADAPTATION.....	35
3.1. Introduction	35
3.2. State of the Art	38
3.2.1. Video Memory.....	38
3.2.2. Review of Relevant Literature.....	39
3.2.2.1. Video-Specific Memory with Design-Time Fixed Quality:	40
3.2.2.2. Adaptive Memory with Dynamic Power-Quality Management:.....	40
3.3. Proposed Low-Cost ECC Storage Scheme	41
3.3.1. Traditional ECC.....	41
3.3.2. Bit Significance Characteristics of Video Data and Proposed Storage Scheme for Parity Bits	44
3.4. ECC Adaptation Based on Requirements and Failure Rate Based on Voltage.....	46
3.4.1. Failure Characteristics of 6T SRAM.....	48
3.4.2. Errors Injected, Including in Parity Bits.....	49
3.4.3. ECC Under Various Failure Rates	51
3.4.4. Proposed Runtime ECC Adaptation Scheme	52

3.5. Proposed Memory	53
3.5.1. Reusable ECC Encoder for ECC1511 and ECC74	54
3.5.2. Reusable ECC Decoder for ECC1511 and ECC74	55
3.5.3. Correction Unit.....	56
3.5.4. Output MUX.....	57
3.6. Results	57
3.6.1. Timing Diagram	57
3.6.2. Power Efficiency	60
3.6.3. Video Quality	61
3.7. Hardware Implementation for Verification.....	62
3.7.1. Variable Voltage SRAM Test Platform	63
3.7.2. SRAM Error Characterization at Given Voltages	63
3.7.3. Hardware SRAM Testing.....	65
3.7.4. Hardware SRAM Analysis	67
3.8. Comparison with Prior Work	71
3.8.1. Compared to State-of-the-Art Approximate Video Memories.....	71
3.8.2. Compared to State-of-the-Art Adaptive SRAM.....	71
3.8.3. Compared to State-of-the-Art SRAM with traditional ECC	72
3.8.4. Compared to State-of-the-Art Memory with Selective ECC	72
3.8.5. Comparison Summary	73
CHAPTER 4. CONTENT-ADAPTABLE ROI-AWARE VIDEO STORAGE FOR POWER-QUALITY SCALABLE MOBILE STREAMING	74
4.1. Introduction	74
4.2. State of the Art	77
4.2.1. Approximate Video-Specific Memory	77
4.2.2. Viewer-Aware Video Memory	78

4.3. Overview of the Proposed Technique	79
4.3.1. Motivational Example	79
4.3.1.1. Protected ROI	81
4.3.1.2. Power savings vs. Bits Truncated.....	81
4.3.2. Overview of the Proposed Content-Adaptable ROI-Aware Video Storage.....	81
4.3.2.1. ROI Awareness	82
4.3.2.2. Video Content Adaptation	84
4.3.2.3. Truncation Region Extractor	84
4.3.2.4. 3-Bit Truncation.....	85
4.4. Proposed Technique: System Level and Circuit Level Implementation.....	86
4.4.1. System-Level Implementation: Video Streaming Platform	86
4.4.2. Memory Bit Truncation Manager.....	89
4.4.3. H.264 Decoder and MBTM Integration	92
4.4.4. Circuit-Level Implementation of the Proposed Frame Buffer Memory.....	94
4.5. Experimental Methodologies	95
4.5.1. Video Selection	95
4.5.2. Video Frame Quality Metrics.....	96
4.5.3. System-Level and Circuit-Level Implementation	96
4.5.4. Video Quality Evaluation.....	97
4.5.5. Statistical Hypothesis Validation	97
4.6. Experimental Results.....	97
4.6.1. Hardware FPGA System MBTM Overhead.....	97
4.6.2. Circuit-Level Frame Buffer Timing Diagram	98
4.6.3. Circuit-Level Frame Buffer Power Saving Analysis.....	99
4.6.4. Video Visual Quality Comparisons.....	104

4.6.5. Objective Video Quality and Bit Truncation Analysis.....	105
4.6.6. Video-Level Power Saving Analysis.....	106
4.6.7. Statistical Analysis	109
CHAPTER 5. CONCLUSIONS AND FUTURE WORK.....	114
5.1. Chapter 2: Content-Adaptive Memory for Viewer-Aware Energy-Quality Scalable Mobile Video Systems	114
5.2. Chapter 3: Flexible Low Cost Power-Efficient Video Memory with ECC- Adaptation	114
5.3. Chapter 4: Content-Adaptable ROI-Aware Video Storage for Power-Quality Scalable Mobile Streaming	115
CHAPTER 6. REFERENCES	117
6.1. Chapter 2	117
6.2. Chapter 3	122
6.3. Chapter 4	125
APPENDIX A. MACROBLOCK VARIANCE TRUNCATION.....	131
APPENDIX B. ECC 74 AND ECC 1511 ANALYZER	136
APPENDIX C. SRAM TEST PLATFORM SUITE	147
C.1. Raspberry Pi Master Controller.....	147
C.1.1. arduino-slave.py.....	147
C.2. Arduino SRAM Slave Controller	149
C.2.1. arduino_sram_slave.ino	149
C.2.2. SRAM_Slave.h.....	151
C.2.3. SRAM_Slave.cpp	152
C.2.4. I2C.h	154
C.2.5. I2C.cpp.....	155
C.2.6. CY62147GE.h.....	160

C.2.7. GY62147GE.c.....	160
C.2.8. CY62147GE_Defines.h.....	162
C.2.9. Global_Defines.h.....	164
C.3. Arduino Voltage Slave Controller.....	165
C.3.1. arduino_voltage_slave.ino.....	165
C.3.2. Voltage_Slave.h.....	165
C.3.3. Voltage_Slave.cpp.....	166
APPENDIX D. BIT TRUNCATION MANAGER.....	168
D.1. BitTruncationManager IP Core.....	168
D.2. BitTruncationManager Test Bench.....	170

LIST OF TABLES

<u>Table</u>		<u>Page</u>
1.	Video Memories and Their Functionality.....	10
2.	Results of videos with different LSBs truncated in H264 decoder different memories.....	13
3.	Output Quality of Different Videos with Bit Truncation.....	16
4.	Results of ordinal logistic regression.....	23
5.	Traditional ECC74 and ECC1511	41
6.	Impact of Traditional ECC on Video.....	43
7.	Proposed ECC.....	45
8.	Comparison with Prior Work.....	71
9.	Comparison with [11].....	73
10.	Truncation Region GPIO Protocol.....	88
11.	Visual Comparison of Selected Video Frames	103
12.	Selected ROI Videos. Analysis Results	107
13.	Results of NON-ROI Videos	108

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1. Proposed content-adaptive mobile video memory for viewer-aware mobile video systems.....	5
2. Mobile video memory architecture. Block diagram of the mobile video decoding and display process. Different memories are shaded. MB: Macroblock.	10
3. Mobile video memory architecture. Xilinx Zynq 7020 FPGA based system implementation.	12
4. Plain MBs visualization and video output comparison of two videos with varying plain MB % (with 2 LSBs truncated). White: plain MBs.....	18
5. Acceptable truncated bits based on subjective feedback. 1T: 1 LSB truncated; 2T: 2 LSBs truncated; 3T: 3 LSBs truncated.	19
6. Developed decision tree model for bit truncation.	20
7. Average PSNR values of 2,000 YouTube-8M videos using two different truncation techniques.	26
8. Content-adaptive video memory.....	26
9. Timing diagram. DATA7: MSB; DATA0: LSB	28
10. Physical layout design.....	29
11. Power savings.	30
12. Psychological experiment set-up at North Dakota State University Center for Visual and Cognitive Neuroscience.....	31
13. Output quality of video (tag wF6lvdXXwc4): (a) with 3 LSBs truncated using decision tree model and (b) with 2 LSBs truncated using the developed ordinal logistic regression model.	32
14. Video quality testing results using the decision tree model.....	34
15. Mobile video memory architecture for steaming. The reference frame memory and frame buffer are accessed very frequently, and have a profound impact on the system's overall cost and power consumption.....	39
16. Video output quality with traditional ECC. (a) Original frame, (b) ECC74 parity bits stored with PSNR = 27.75 dB, and (c) ECC1511 parity bits stored with PSNR = 8.27 dB.....	43

17.	Encoded video frame (Akiyo) with (a) ECC74 where parity is stored in the LSBs with PSNR = 41.2582 and (b) ECC1511 where parity bits were stored in LSBs with PSNR = 39.8426 dB.	47
18.	Relation between supply voltage (VDD) and SRAM bitcell failure rate in a 45nm CMOS technology.	47
19.	Error map and video quality with the proposed ECC74 under 0.1% faulty memory bitcells: (a) and (b) Error map and original image with 0.1% memory failures injected; (c) and (d): parity bits stored in the LSBs with the proposed ECC74, and corresponding video quality; (e) and (f): Error map and encoded video quality with the proposed ECC74, with the exact same number and position of errors from (a) injected into the memory; (g) and (h): Error map and decoded video quality with the proposed ECC74.	48
20.	Error map and video quality with the proposed ECC1511 under 0.1% faulty memory bitcells: (a) and (b) parity bits stored in the LSBs with the proposed ECC1511, and corresponding video quality; (c) and (d): Error map and encoded video quality with the proposed ECC1511; (e) and (f): Error map and decoded video quality with the proposed ECC1511.	50
21.	ECC adaptation based on failure rate and corresponding PSNR.	51
22.	Proposed Adaptive ECC Memory.	53
23.	ECC Encoder.	54
24.	ECC Decoder.	55
25.	Correction Unit.	56
26.	Output MUX.	57
27.	Timing diagram: purple marked bits are the combination that generated the orange marked parity bits for each operation.	58
28.	Power Comparison.	59
29.	PSNR values of 1,000 videos at 0.1% and 0.9% failure rates.	61
30.	Variable voltage SRAM test platform: (a) Arduino Mega, Arduino Due, Raspberry Pi, Texas Instruments Level Shifters, and Cypress 65nm TSOP-44(II) SRAM based system implementation.	62
31.	Block diagram of the Master and Slave controller interaction to write, read, and verify data on the SRAM, as well as control its supply voltage.	64

32.	(a) SRAM failure characterization map for the entire addressable memory region per bit. Voltage ranges 1.30V down to 1.00V are shown, which demonstrate the per bit error distributions from a 0% to 100% failure rate. The granularity of voltage plotted is 0.01V. Voltages 2.20V down to 1.31V were not shown for clarity, as they resulted in a 0% failure rate; (b) failure rate at each voltage between 1.02V and 1.25V. Failure rates not shown on the graph above 1.25V were nearly 0%, above 1.30V were 0%, and below 1.02V were 100%.	66
33.	SRAM error distribution map at 1.06V, where failure rate is measured as 44.25244%, to show the uniformity of error distribution (black pixels denote error): (a) exaggerated visualization of 16bit by 256K SRAM memory structure (b) Sub-section of (a) to clearly show the error distribution across 128 words: 16bits by 128 (c) Re-organized and re-assembled from (a).....	67
34.	Failure rate and PSNR values at different supply voltages for three memory modes (No ECC, ECC74, and ECC1511).	69
35.	Zoomed in video frame using the proposed memory with different modes (No ECC, ECC74, and ECC1511) at two different supply voltages. The circled regions highlight visual degradation due to slightly increased number of errors.....	70
36.	Proposed content-adaptable ROI-aware low-power video memory.....	75
37.	Observer discernable flaws in the facial region due to a “banding effect” on the face when comparing (a) and (b) caused the overall quality of the frame to become unacceptable at 3 truncated bits (Video tag: wF6lvdXXwc4 from [14]).	80
38.	Proposed Region-Of-Interest and macroblock texture framework.....	82
39.	Akiyo from [28], sample visualized, as generated internal to the proposed method’s frame parsing process. Pink, preserved ROI. Seven possible truncation combinations: 1. Green, Y vector truncation. 2. Blue, U vector truncation. 3. Yellow, V vector truncation. 4. Dark blue, YU vectors truncation. 5. Dark Yellow, UV vector truncation. 6. Dark green, YV vectors truncation. 7. Grey, YUV vectors truncation.	83
40.	H264 video stream demonstration platform hardware system.	87
41.	Mobile video steaming system block diagram.....	88
42.	(a) Encoded frame 175 from Johnny_1280x720_60 video [28]. (b) Visual of areas being truncated. 45 regions total. (c) Output decoded frame. 2,282,496 bits truncated.....	90
43.	Circuit-Level implementation of the proposed frame buffer memory.....	91
44.	Timing diagram of the frame buffer circuit.	99

45.	Hardware FPGA system post- implementation project summary without BTM. (a) On-Chip Power, Total Power: 2.203W. (b) Resource allocation.	101
46.	Hardware FPGA system post-implementation project summary with BTM. (a) On-Chip Power, Total Power: 2.271W. (b) Resource allocation.	102
47.	Power savings (one word) of the frame buffer circuit.	104
48.	Impact of the video content characteristics on the effectiveness of the proposed technique, compared to old technique.	108
49.	Histogram of quality Improvement distributions. Number of data points and P- value shown, between the truncation method in [7] and the proposed method. All distributions are 3-parameter Weibull distributions that fall within a 95% Confidence Interval.	110
50.	Histogram of power savings, measured in percentage improvement, between the truncation method in [7] and the proposed method. All distributions are 3- parameter Weibull distributions that fall within a 95% Confidence Interval.	111
51.	Histogram of PSNR noise increase, between the truncation method in [7] and the proposed method. All distributions are Normal Distributions that fall within a 95% Confidence Interval.	111

LIST OF ABBREVIATIONS

ANN.....	Artificial Neural Network
BIST.....	Built-In Self-Test
CMOS.....	Complementary Metal-Oxide-Semiconductor
CNN.....	Convolutional Neural Network
DPSR.....	Data Pattern Self-Recovery
IoT.....	Internet of Things
LSB.....	Least Significant Bit
MSB.....	Most Significant Bit
MSE.....	Mean Squared Error
MUX.....	Multiplexer
NMOS.....	N-Type Metal-Oxide-Semiconductor
PMOS.....	P-Type Metal-Oxide-Semiconductor
POST.....	Power-On Self-Test
PSNR.....	Peak Signal-To-Noise Ratio
WPSNR.....	Weighted Peak Signal-To-Noise Ratio
RBL.....	Readout Bit Line
RDF.....	Random Dopant Fluctuation
ROI.....	Region-of-Interest
SRAM.....	Static Random Access Memory
SSIM.....	Structural Similarity
V_{dd}	Supply Voltage

CHAPTER 1. INTRODUCTION

Memory in mobile systems is an intrinsic power-consuming vector. Mobile systems consume the most power during video streaming operations. Thus, reducing the memory usage during video streaming operations without sacrificing content delivery becomes one of the most effective and popular methods of power savings realized in mobile systems. This disquisition presents with detailed descriptions several techniques –which are viewer-aware, implement ECC-adaptation, or content aware, – that leverage intended quality degradation in video clinically proven indiscernible by users – which ultimately enable power savings in video memory. These techniques include viewer-aware bit-truncation, self-correcting memory throughout voltage scaling, and smart dynamic memory management in video decoder processes through Region-of-Interest identification via deep learning. This chapter shall introduce these techniques and the motivation.

1.1. Motivation

Users demand increased battery life in their mobile systems. Silicon area reduction plays a key role in reducing power consumption and increasing battery life in mobile systems: as circuitry power demands are lower as a function of reduced silicon area. The current technology trend shows that advancement in battery technology is magnitudes slower than advancement in silicon area. This trend, however, may soon falter to a halt as silicon area nears physical boundaries due to quantum effects. Therefore, researchers are motivated to find other vectors where potential power savings realized. Methods and techniques, such as the ones discussed in this disquisition, thus become popular and desirable.

1.2. Viewer-Aware Bit-Truncation

SRAM and DRAM power consumption is a function of how many bit cells are read or written and refreshed respectively. Both SRAM and DRAM have leakage power. Methods are proposed where if a cell is not utilized, it is turned off and no power is consumed. Memory has a specific data width where each bit in the data width is stored in a memory cell. Binary data significance increases as the bit index increases in the data width. The upper most indexed bit is known as the “Most significant bit” and the lower most indexed bit is known as the “Least significant bit”. Bit-truncation saves power by truncating, or turning off, some number of the least significant bits/cells in the memory; however, the trade-off of truncating data bits is data degradation. Viewer-aware bit-truncation truncates in such a way where the viewer does not notice a difference between the non-truncated and bit-truncated data. This technique works only on data that is not corrupted by bit-truncation, such as decoded video data.

1.3. Self-Correcting Memory Throughout Voltage Scaling

Memory voltage thresholds are determined by memory cell sizes and the lithography process. Then, the specific memory is analyzed at different supply voltages in various environmental conditions and temperatures to establish voltages that result in expected memory behavior. Expected memory behavior is usually behavior where the memory data is written, read, retained for a specified amount of time, and then validated. The results shall fall within an error tolerance range acceptable for the use-case. Memory may encounter conditions outside of what was specified, such as a lower supply voltage. As a result, memory may implement self-correcting parity code, known as Error Correcting Code (ECC), where parity bits are used to determine bit-flip errors. This process is leveraged for data that is not corruptible due to bit-flip errors, such as decoded video data, where instead its quality degrades as a function of bit-flip

errors to reduce power, by intentionally reducing voltage which is also known as, voltage scaling. This voltage scaling technique is utilized along with ECC to create novel and powerful power saving advantages.

1.4. Smart Dynamic Memory Management in Video Decoder Processes

The purpose of hardware and software video decoders is to decompress a video bit stream into frame data at the pixel granularity for display purposes. The uncompressed video is very large in storage footprint; thus, for ease of transfer and to lower its storage footprint, the video is compressed, or as known by the art, encoded. Hardware video decoders utilize many different types of memories during the video decoding process. Some of these internal memories are susceptible to errors that do not corrupt the video frame data, as they store decoded video data, while others are not as they play an important role in the process of decoding. The former memories are leveraged for power savings by systematically and artificially intelligently preserving and truncating memory data regions which the video decoder deems as “Regions-of-Interest” and “Truncation Regions” respectively, based on video content: such as facial features or objects identified via deep learning techniques.

Another smart memory management technique in hardware video decoders is also explored. Before the video bit stream is sent to the video decoder for decoding and display, it is encoded by a video encoder to reduce its storage footprint by orders of magnitudes. During the video encoding process, video data metadata may be collected. The metadata may contain information such as the amount of complexity in each video frame for the decoder to leverage if bit truncation logic for power savings were to be implemented into the video decoder. This process is leveraged along with other smart techniques to dynamically manage memory in video decoders for power savings.

CHAPTER 2. CONTENT-ADAPTIVE MEMORY FOR VIEWER-AWARE ENERGY-QUALITY SCALABLE MOBILE VIDEO SYSTEMS¹

Mobile devices are becoming ever more popular for streaming videos, which account for the majority of all data traffic on the internet. Memory is a critical component in mobile video processing systems, increasingly dominating power consumption. Today, memory designers are still focusing on hardware-level power optimization techniques, which usually come with significant implementation cost (e.g., silicon area overhead or performance penalty). In this chapter, a video content-aware memory technique for power-quality trade-off from viewers' perspectives is proposed. Based on the influence of video macroblock characteristics on viewer experience, two simple and effective models - decision tree and logistic regression – are developed in order to enable hardware adaptation. A novel viewer-aware bit-truncation technique has also been implemented, which minimizes the impact on viewer experience, while introducing energy-quality adaptation to the video storage.

2.1. Introduction

Video is *everywhere* today. According to the recent Cisco Visual Networking Index, Mobile video traffic accounted for 60% of total mobile data in 2016 [1]. It is expected to increase 9-fold between 2016 and 2021 and grow to approximately 78% in 2021, with the continuous evolution of mobile networks and the proliferation of mobile devices [1]. Consequently, video steaming has become one of the most energy-intensive applications on mobile devices. In

¹ The material in this chapter was authored by Jonathon Edstrom, Yifu Gong, Ali Ahmad Haidous, Brittney Humphrey, Mark E. McCourt, Yiwen Xu, Jinhui Wang, and Na Gong. Ali Ahmad Haidous was in charge of the design, development, and implementation of an FPGA based H264 decoder memory architecture bit truncation embedded test system platform for analysis and feasibility validation of the novelties presented in the chapter. Jonathon Edstrom was in charge of data analysis, video quality metrics, and simulation results. Yifu Gong provided the presented SRAM hardware design with power simulation results based on the data analysis and software simulations. Mark E. McCourt, Yiwen Xu, and Jinhui Wang were co-principal investigators. Na Gong was the principal investigator.

particular, during the mobile video steaming process, the frequent memory access contributes to over 92% of the motion compensation energy [2] and 50% of the video decoding consumption [3], the high energy consumption restraints are only expected to increase with the emerging of Ultra-High-Definition (UHD) (e.g., 4K and 8K) videos [4]. Accordingly, enhancing energy efficiency of video memories is of paramount importance to enable efficient mobile video systems, and is also one of the key design considerations to deliver 4K/8K UHD videos to mobile devices.

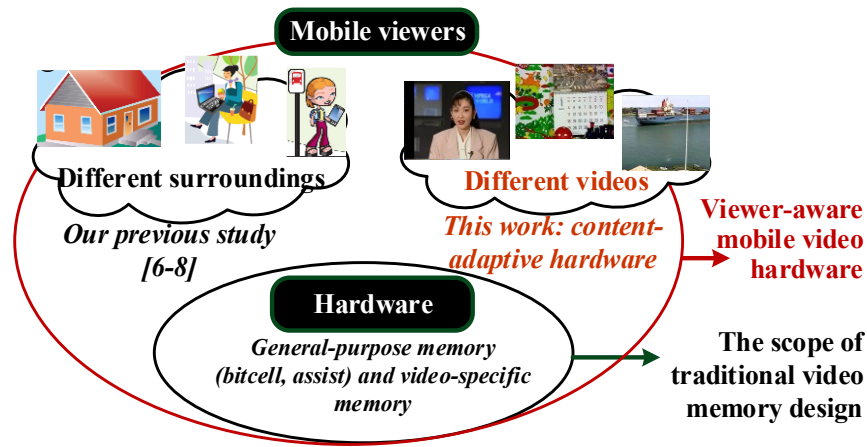


Figure 1. Proposed content-adaptive mobile video memory for viewer-aware mobile video systems.

Designers have extensively exploited memory techniques for power reduction, but traditional memory designs are typically developed based on an objective video output metric such as the peak signal-to-noise ratio (*PSNR*), without dynamic energy-quality adaptation to viewer’s true experience. Such hardware-viewer isolation is mainly due to the following design challenges. First, the existing models to represent viewer’s experience, such as the recently developed human visual system (HVS) model [5], are too conceptual and too complex, to be useful in guiding hardware design. Second, hardware design, particularly memories, usually lack run-time adaptation and therefore new hardware design techniques that enable viewer-aware adaptation need to be explored. Last, but not least, it is challenging for mobile designers to

directly connect hardware design to viewer's experience, which requires professional lab setup, human subject involvement, and psychophysical analysis.

Recently explored was viewer-aware video memory design by investigating the impact of illuminance levels in different viewing surroundings on the viewer's experience [6-8], as illustrated in Figure 1. Specifically, a bit truncation technique was used to introduce memory failures in high noise-tolerance viewing contexts with high luminance levels by adaptively disabling the least significant bits (LSB) of the video data stored in memories. Previous studies [6-8] illustrate a new dimension of power savings for hardware design through the introduction of viewer awareness, but the developed memory lacks adaptation across a wide variety of mobile videos. To enable an optimized trade-off between energy efficiency and video quality, in this chapter, novel energy-quality scalable video memory design technique is proposed that takes into account video content to adjust the energy-quality trade-off according to viewer's experience. Specially, this chapter makes the following contributions.

- A Xilinx Zynq 7020 FPGA based H.264 decoder and display system was developed and based on it, the contribution of different video memories to the output quality has been analyzed. A frame buffer was demonstrated can tolerant significant memory failures, which enables power saving opportunities for hardware design (Section 2.3.1 and Appendix).
- The impact of video content on viewer's experience is studied from the psychological perspective. The correlation characteristics between "banding distortion" to viewers caused by hardware noise and the areas in frames that exhibit low variance among pixel luminance values have the potential to enable content-adaptation opportunities for hardware design were concluded (Section 2.3.2).

- Based on macroblock characteristics analysis and subjective video testing, two models including one decision tree model and one logistic regression model have been developed to enable effective connection of the video content to the hardware design process (Section 2.4).
- Developed a novel viewer-aware bit truncation technique which enables better visual experience while maintaining similar power efficiency. Based on the developed models and viewer-aware bit truncation technique, a content-adaptive video memory design with dynamic energy-quality trade-off is implemented (Section 2.5).
- Finally, a comprehensive suite of simulations on the proposed content-adaptive video memory is performed and the enriched results including performance, layout design, video output quality of various mobile videos, and power efficiency, are discussed (details are shown in Section 2.6).

To the best of the authors' knowledge, the proposed memory has made the first attempt to *exploit viewer's experience and video content* to enable energy-quality adaptive hardware design.

The organization of the chapter is as follows. A review of related video memories is provided in Section 2.2. In Section 2.3, The contributions of video memories and the impact of video content on viewer's experience were studied. In Section 2.4, subjective testing procedures and model development processes were presented. The hardware design, that implements the functionality for power savings through viewer-aware bit truncation, is presented in Section 2.5. The evaluation results are presented in Section 2.6. Finally, this chapter is concluded in Chapter 5.

2.2. Related Work

There is a rich body of literature for power reduction for embedded memories and voltage scaling is particularly effective to reduce the memories' power consumption due to the strong dependency of dynamic and leakage power consumption on supply voltage. However, voltage-scaled SRAMs are susceptible to failures and many techniques have been developed, which mainly fall into the following three aspects: (i) *assist* schemes such as boosted wordline [9], negative bitline [10], and dual-rail supply [11]; (ii) *more-than-6T* bitcells to achieve low voltage operation, such as 8T [12], 9T [13], and 10T [14]; and (iii) *error-correction* techniques such as error correction codes [15] and data remapping [16]. However, the improvements in embedded memory power efficiency are often achieved with significant design complexity, silicon area overhead, and performance penalty for voltage regulators and boosting circuits.

Several recent efforts have investigated application resilience of videos to approximations with “good enough” output and additional power savings. Chang et al. [17] present a hybrid 6T+8T SRAM to achieve quality-power optimization. In [18], a heterogeneous sizing scheme is presented to reduce the failure probability of conventional 6T bitcells. In [19], the correlation between the most-significant-bits (MSBs) of video data was utilized to design a hybrid 8T+10T memory for power savings.

At the same time, alternative metrics for the analyzing videos objectively, including Structural Similarity (*SSIM*) and *PSNR-B*, have recently been shown to outperform the traditional mean squared error (*MSE*) and *PSNR* [32, 33]. While *SSIM* and *PSNR-B* have more meaning in terms of the viewer's perception of a video, the complexity of their calculations makes them less useful to hardware designers when optimizing energy-quality tradeoff.

Very recently, viewer-aware video memory design were investigated by studying the impact of illuminance levels in viewing contexts on the viewer's experience [6-8], where an increased amount of ambient luminance allows for a larger amount of bits to be truncated without noticeable degradation to the viewers. A viewing context-aware SRAM (VCAS) was developed, which introduces memory failures in luminance contexts with high memory failure tolerance. Two low-power techniques - voltage scaling and bit truncation - are explored to implement. Those two techniques were concluded to achieve similar *PSNR* values, but the video quality degradation caused by bit truncation is much less noticeable than that of the voltage scaling technique for the viewers. The hardware design from the previous study has been developed for general videos, although, the video characteristics were observed to significantly influence the viewer's experience [8]. In this chapter, the impact of video content characteristics on viewer's experience to enable video content-adaptive memory with dynamic energy-quality tradeoff was studied.

It is worthy to emphasize that, the proposed content-adaptive video memory as well as viewing luminance-aware video memories [6-8] are orthogonal to existing low-power hardware-level memories and they can be applied simultaneously to optimize power efficiency.

2.3. Influence of Video Content on Viewer's Experience

2.3.1. Mobile Video Memory System

Video streaming has become the most important energy-intensive application used in mobile devices [8]. Figure 2 shows the block diagram of a H.264 video decoding and display system [36]. After parsing compressed bitstream, the inter predictor uses the reconstructed frames stored in the reference frame buffer and the transmitted motion vectors to construct new frames. After the frames are decoded, the display controller sends them from the frame buffer to

the display panel periodically. During this process, multiple memories are needed for storing the intermediate and final results of the frame data, as listed in Table 1.

Table 1. Video Memories and Their Functionality

<i>Video Memories</i>	Size in Bits (Width x Depth)	Memory Functionality
Chroma Level Cb	32 x 8	Stores the blue-difference color space bottom line pixels for up macroblocks
Chroma Level Cr	32 x 8	Stores the red-difference color space bottom line pixels for up macroblocks
Luminosity Level	32 x 8	Stores the luminosity color space bottom line pixels for up macroblocks
Reconstructed Neighboring	32 x 7	Stores neighboring pixels of a luma block after the current macroblock is coded and reconstructed
Prediction Mode	16 x 7	Stores the current macroblock prediction mode for 4x4 blocks
Motion Vector X	64 x 7	Stores the horizontal motion vector prediction calculation of surrounding blocks' motion data
Motion Vector Y	64 x 7	Stores the vertical motion vector prediction calculation of surrounding blocks' motion data
Reference Macroblock	8 x 8	Stores the reference I, SI, P, or SP macroblock used for inter prediction
Frame buffer	64 x 512	Stores the current and previous decoded frames for prediction and display, respectively
Y Display	64 x 8	Stores the luma Y component of the display memory for HDMI output buffer
U Display	64 x 8	Stores the chrominance U component of the display memory for HDMI output buffer
V Display	64 x 8	Stores the chrominance V component of the display memory for HDMI output buffer

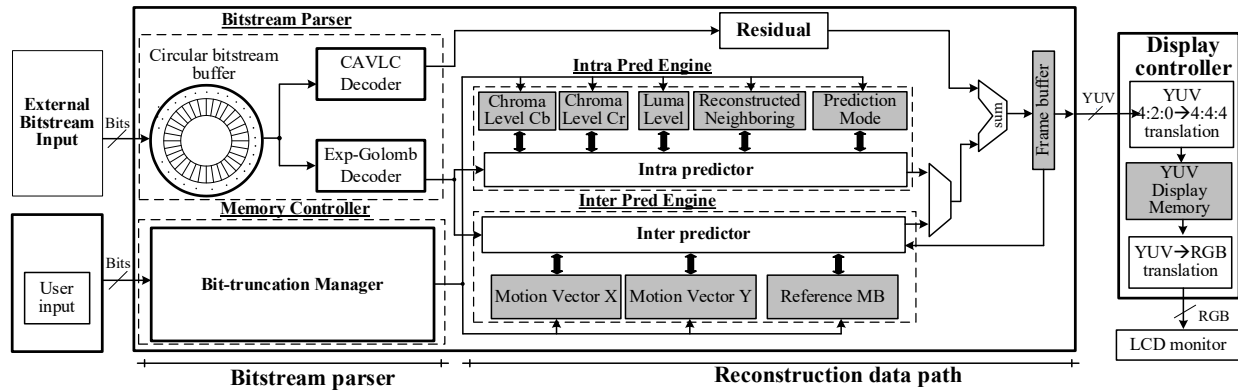


Figure 2. Mobile video memory architecture. Block diagram of the mobile video decoding and display process. Different memories are shaded. MB: Macroblock.

To evaluate the contribution of different memories to the output video quality, a video decoder and display system was developed, as shown in Figure 3. For the memories listed in Table 2, the bit truncation technique [7] was applied to each memory during the video decoding process by disabling least-significant bits (LSBs) [8, 21] and then the output video is captured for quality evaluation. Specifically, LSB truncation starting with one bit with a maximum of five bits

have been applied to each video memory. The encoded bitstream, which resided on an on-board SD card, is decoded using a Xilinx Zynq 7020 FPGA based H.264 decoder. An Arduino-based memory controller is implemented select the memory for truncation as well as the number of truncated LSB which are specified by the user input over a serial interface. A video capture card is utilized to capture the video output over the HDMI output for evaluation. It has been shown that, the frame buffer, the largest memory, can tolerant three truncated LSBs, which provides power saving opportunities for a hardware design. The detailed results are discussed in Table 2

Table 2 lists the results with LSB truncation in different video memories using the video system shown in Figure 3. The standard video sequence *aspen_1080p.y4m* [25], which has a wide range of plain MB percentages across different frames, is used for evaluation. The average plain MB percentage was 20.90%; the maximum and minimum were 50.89% at frame #367 and 3.03% at frame #113, respectively. The video was encoded with the following ffmpeg [37] command:

```
ffmpeg -i aspen_1080p.y4m -profile:v baseline -pixel_format yuv420p -  
level 3.1 -framerate 30 -preset 1 -cavlc 1 -pix_fmt yuv420p  
aspen_1080p.264
```

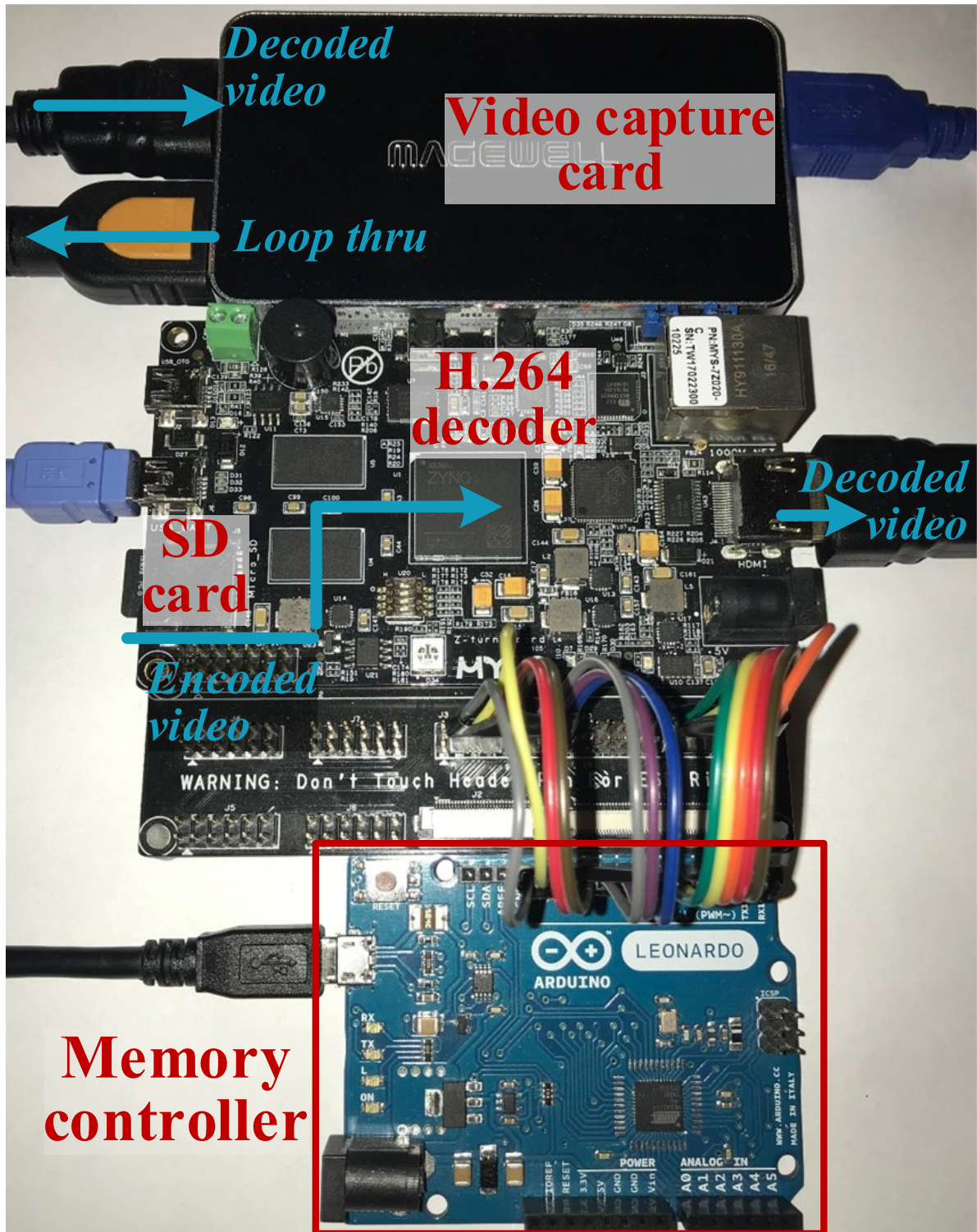


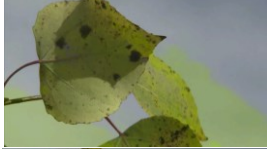














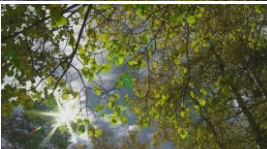
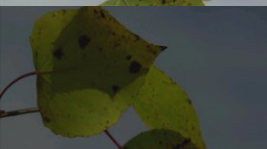



Figure 3. Mobile video memory architecture. Xilinx Zynq 7020 FPGA based system implementation.

Table 2. Results of videos with different LSBs truncated in H264 decoder different memories

<i>Memories</i>	Number of LSBs Truncated	<i>PSNR</i> (Max MB %)	<i>PSNR</i> (Min MB %)	Max Plain MB % Frame	Min Plain MB % Frame
<i>Original video frames without any truncation</i>	-	-	-		
<i>Chroma Level Cb</i>	5 LSBs truncated	43.0 dB	31.1 dB		
<i>Chroma Level Cr</i>	5 LSBs truncated	36.3 dB	27.0 dB		
<i>Luminosity Level</i>	5 LSBs truncated	18.0 dB	16.7 dB		
<i>Prediction Mode</i>	1 LSB truncated	23.0 dB	19.5 dB		
<i>Motion Vector X</i>	1 LSB truncated	29.2 dB	13.5 dB		
<i>Motion Vector Y</i>	1 LSB truncated	29.1 dB	13.3 dB		
Reference Macroblock (MB)	5 LSBs truncated	42.8 dB	32.8 dB		
Frame Buffer	3 LSBs truncated	44 dB	25.5 dB		
<i>YUV Display</i>	2 LSBs truncated in each vector	11.8 dB	13.2 dB		

2.3.2. Influence of Video Content on Viewer's Experience in the Presence of Hardware Noise

Traditionally, hardware designers have used *PSNR* for evaluating video quality, which has been recently shown to be insufficient to demonstrate the viewer's experience [20, 31]. *PSNR* does not encompass the necessary information to hardware designers about viewer's experience, due to the fact that key influencing factors for viewer's experience, such as video content and environment conditions, are not included in *PSNR* [31]. In this chapter, the goal was to find a better method to analyze videos in a quantitative way that will also be useful to hardware researchers. This process is began by using the *PSNR* metric to describe video quality. To continue, new insight was added to the traditional *PSNR* metric with the introduction of *content-aware* information. This new form of information allows us to gracefully scale the video quality with enhanced energy efficiency of hardware.

2.3.2.1. Traditional *PSNR* Metric

The traditional *PSNR* metric is defined as [19]

$$PSNR = 10 \log_{10} \left(\frac{255^2}{MSE} \right) \quad (\text{Equation 1})$$

where the *MSE* is the mean squared error between the original video (*Org*) and the degraded video (*Deg*), expressed as

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [Org(i, j) - Deg(i, j)]^2 \quad (\text{Equation 2})$$

Although *PSNR* is simple for hardware designers to understand, it does not truly capture the effect that errors have on the user's perception of the video. To show the lack of complete information the *PSNR* provides in terms of user perception, the bit truncation technique was applied to two videos and calculate the *PSNR* values for 1 to 4 truncated LSBs of the luma data (i.e. the luminance channel, or Y component in raw YUV videos). The bit truncation technique

was adopted to enable energy-quality adaption, which is due to the following two reasons: (i) bit truncation causes blurring in videos, which is similar to the “*banding distortion*” in the codec-algorithm field, and the video degradation is much less noticeable to viewers as compared to other low-power techniques such as voltage scaling [8] and (ii) the power/energy savings with bit truncation is much more significant than other low power techniques such as voltage scaling [21].

Table 3 shows two videos which were downloaded from Google’s recently released Youtube-8M Dataset [22], which is the largest multi-label video dataset. In this chapter, to maintain a short and consistent size label for all included YouTube video samples, the video tag was used to label each video, which is the last portion of the full URL address¹. As observed in Table 3, using the bit truncation technique, the *PSNR* value is reduced by approximately 7dB, on average, for each additional truncated LSB. Both videos have very similar *PSNR* values with the same number of LSBs truncated, but the visual quality is significantly different. As compared to video #1 (video tag: *EFv2FvnLLao*), the “banding distortion” of video #2 (video tag: *FNlpA4FME-8*) is much more noticeable to the viewers. Accordingly, the traditional video quality metric *PSNR* cannot correlate well with the viewer’s experience and the video-content properties, such as the texture/motion characteristics, significantly affect the viewer’s experience. In this chapter, the video content information was introduced to study the viewer’s experience. Specifically, the recently developed video macroblock (MB) characterization was adapted by analyzing the pixel-luminance values’ variance [23], as described in the next subsection.

Table 3. Output Quality of Different Videos with Bit Truncation

Video output quality with 3 LSBs truncated	# LSBs truncated	PSNR (dB)
	Video #1 (video tag: <i>EFv2FvnLLao</i>)	
	1	52.868
	2	44.433
	3	37.490
	4	30.985
	Video #2 (video tag: <i>FNlpA4FME-8</i>)	
	1	52.741
	2	44.461
	3	37.693
	4	31.154

2.3.2.2. Video Macroblock Variance Analysis

The MB variance analysis is typically conducted during the video pre-processing stage when encoding videos [23, 24]. In the analysis, their defined calculation was adopted for determining whether a given MB is considered either *plain* or *textured*, which avoids introducing significant computational overhead. The calculation is based on the variance of pixel luminance values of a given MB and is defined as [23]

$$\begin{aligned}
 V_{MB} &= \sum_{i=0}^{15} \sum_{j=0}^{15} (P(i, j) - \rho_{MB})^2 \gg 8 \\
 MB &= \begin{cases} \textit{Plain} & \textit{if} (V_{MB} \leq Th_{Low}) \\ \textit{Textured} & \textit{Else} \end{cases} \quad (\text{Equation 3})
 \end{aligned}$$

where ρ_{MB} and V_{MB} are the average luminance and variance of luminance values in a given MB, respectively. The value used for Th_{Low} was 1.25 as was determined in [24] through the use of regression analysis. For these purposes, this Th_{Low} value is an arbitrary number used to define the plain macroblock percentages in the model design process (Section 2.4). This MB characterization can be calculated during the encoding process and transmitted as metadata in the video bit stream. Currently an embedded system implementation was used for calculating this average plain MB calculation. To minimize computational overhead, a single, averaged plain

MB percentage was calculated that represents an entire sample. However, it is possible to calculate a per frame MB percentage for videos that change scenes frequently for dynamic adaptation. Two benchmark videos, *Akiyo* and *News*, were initially retrieved from [25]; these videos contained static backgrounds with a low amount of motion from the reporter(s) in the videos. Both videos displayed low plain macroblock percentages when analyzed. It was obtained further 32 video samples with similar broadcasting characteristics from the Youtube-8M Dataset [22] and calculate the percentages per frame for the *minimum*, *maximum*, *median*, and *average* percentage of each sample video. Figure 4 displays two video samples with similar *PSNR* values but varying plain MB percentages (with 2 LSBs truncated). The distribution of plain MBs and the resulting banding distortion effect are visualized in Figure 4. An important observation is that ***a noticeable relationship exists between the banding distortion and plain MBs***; videos with large amounts of plain MBs, especially where the plain MBs are dense, tend to have decreased visual quality to the viewers. Accordingly, this relationship was utilized to develop a content-adaptive model to predict the number of truncated LSBs for different videos. Specifically, to minimize the computational overhead, the *average* plain MB percentage was used per video frame and focus on low-motion videos with a stationary camera or containing a reporter in the analysis.

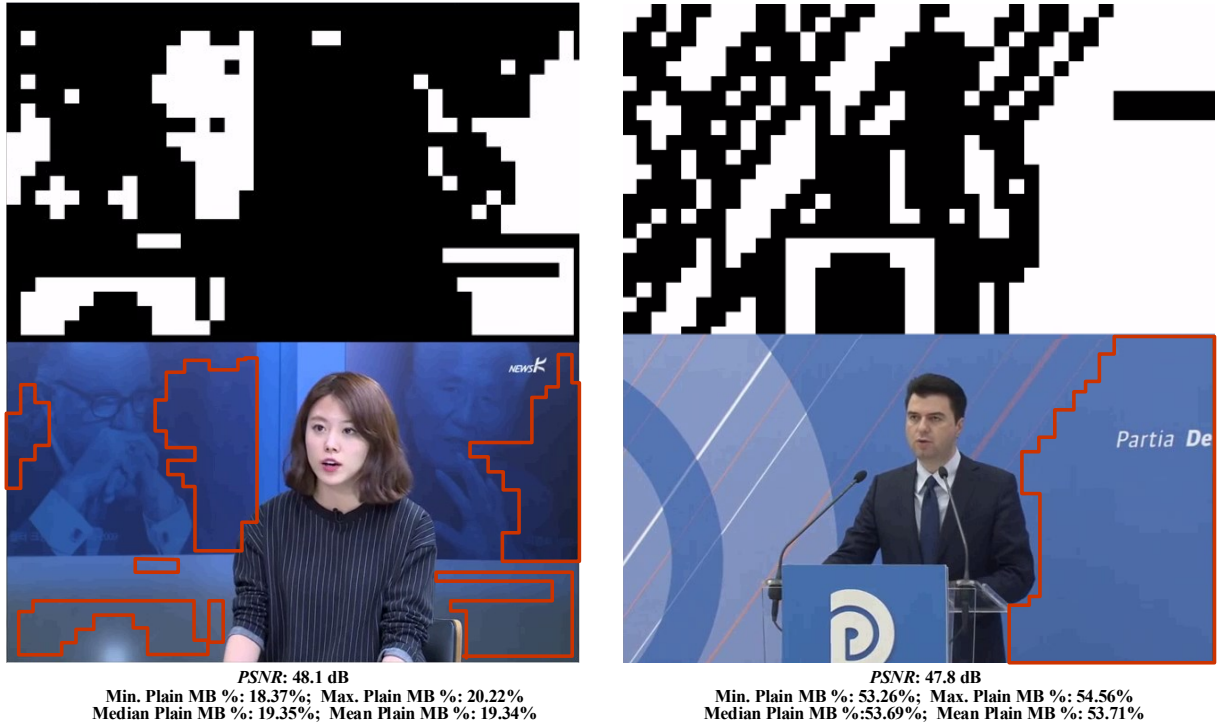


Figure 4. Plain MBs visualization and video output comparison of two videos with varying plain MB % (with 2 LSBs truncated). White: plain MBs.

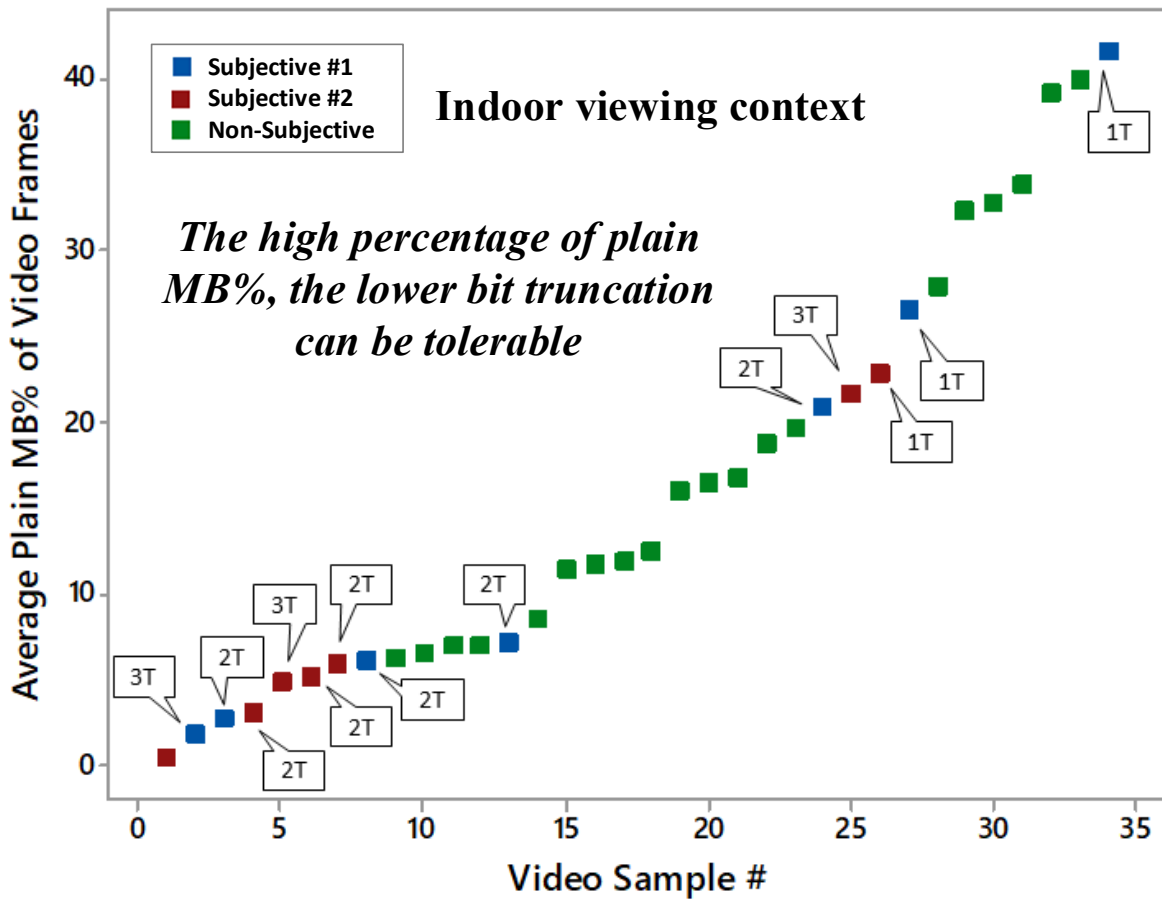


Figure 5. Acceptable truncated bits based on subjective feedback. 1T: 1 LSB truncated; 2T: 2 LSBs truncated; 3T: 3 LSBs truncated.

2.4. Modeling Process

To determine the acceptable number of LSBs to truncate for different videos, subjective video testing was conducted and based on the collected data, two models were developed using decision tree and logistic regression methods. For the initial study, described in this chapter, only considered were the luma (Y) component when truncating LSBs.

2.4.1. Subjective Testing Procedure for Data Collection

Two sets of subjective video studies were conducted to collect viewers' feedback. Within each of the studies designed for subjective analysis of truncation techniques, participants were asked to view multiple versions of the same video. The testing procedure follows guidelines

from the ITU [26] and uses the Degradation Category Rating (DCR) method [20], which is also known as the Double Stimulus Impairment Scale (DSIS). The participants were asked to watch both original video and truncated video and then score from 1 to 5 based on the quality in their opinions (*imperceptible-5, perceptible but not annoying-4, slightly annoying-3, annoying-2, very annoying-1*). An average score of 4.0 or higher was used as the target for acceptable video quality [27]. The first (second) of two studies contained 10 (13) participants who were each asked to view 7 (9) individual videos from the 34 sample videos.

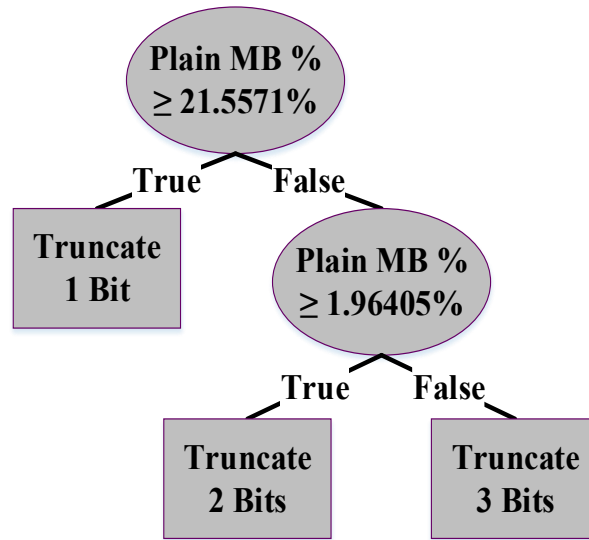


Figure 6. Developed decision tree model for bit truncation.

With these average scores for different amounts of LSBs truncated, the video samples were split into different regions. Based on this, models were developed that connect the average plain MB percentage to number of LSBs that can be truncated.

2.4.2. Modeling Process

2.4.2.1. Decision Tree Model

From the initial subjective studies, the goal was to model the correlations between the calculated average plain MB percentage and the largest amount of LSBs that can be truncated for a given *PSNR* that will maintain an acceptable video quality. Figure 5 displays the video samples

average plain macroblock percentage and how many bits can be truncated based on the minimum acceptable impairment score of 4.0.

From these preliminary results, an inverse relationship was discovered between plain MB percentage and acceptable number of LSBs to truncate. With the knowledge of this relationship and the subjective data gathered from participants, a decision tree model was developed using the Classification Learner tool in MATLAB, as shown in Figure 6. By traversing the tree from the top to the bottom based on the plain MB percentages, the number of truncated LSBs can be obtained for different videos. It is worthy to mention that the majority of videos from the Youtube-8M dataset have plain MB percentages above 1.96405% (see Figure 6) and therefore the number of videos with the decision for 3 LSBs truncation is much less than that of 1 LSB and 2 LSBs truncation.

2.4.2.2. Logistic Regression Model

In the model development process, another widely-applied statistical modeling method was considered: *logistic regression*, in which:

$$\begin{cases} \pi_i = \pi_3 \exp(\beta_{i0} + \beta_{i1}x), & i = 1,2 \\ \pi_3 = \frac{1}{1 + \exp(\beta_{10} + \beta_{11}x) + \exp(\beta_{20} + \beta_{21}x)} \end{cases}, \quad (\text{Equation 4})$$

where $\pi_i := P\{Y = i|x\}$ indicates the probability that the number of truncated LSBs is i for given average plain MB percentage which equals x . Matlab was used to fit the $\hat{\beta}$ coefficients and get $\hat{\beta}_{10} = -1.6636, \hat{\beta}_{11} = 12.7929, \hat{\beta}_{20} = 1.4408, \hat{\beta}_{21} = 1.0497$. However, their corresponding p-values are 0.243, 0.103, 0.111, 0.881, respectively. This implies that all four coefficients are not significant in the regression under a 5% significance level. By observing the data, one can clearly see that this is due to noise.

In addition, notice that, if a user chooses a video as satisfactory which is truncated by k LSBs, then he/she will be satisfied by the same video truncated by k' LSBs where $0 < k' < k$.

The difference between k LSBs and k' LSBs truncation is the energy efficiency that can be enabled; the efficiency is higher for k LSB truncations. To this end, further applied the *ordinal logistic regression*, which yields

$$\ln\left(\frac{\pi_1}{\pi_2+\pi_3}\right) = \beta_{10} + \beta_1 x \quad (\text{Equation 5})$$

$$\ln\left(\frac{\pi_1+\pi_2}{\pi_3}\right) = \beta_{20} + \beta_1 x \quad (\text{Equation 6})$$

Moreover,

$$\pi_1 + \pi_2 + \pi_3 = 1 \quad (\text{Equation 7})$$

Solving (5), (6) and (7), one can get

$$\begin{aligned} \pi_1 &= \frac{\exp(\beta_{10} + \beta_1 x)}{1 + \exp(\beta_{10} + \beta_1 x)} \\ \pi_2 &= \frac{1}{1 + \exp(\beta_{10} + \beta_1 x)} - \frac{1}{1 + \exp(\beta_{20} + \beta_1 x)} \\ \pi_3 &= \frac{1}{1 + \exp(\beta_{20} + \beta_1 x)}. \end{aligned} \quad (\text{Equation 8})$$

Matlab was used to fit the ordinal coefficients and get $\hat{\beta} = [\hat{\beta}_{10}, \hat{\beta}_{20}, \hat{\beta}_1] = [-2.8322, 0.9856, 9.7783]$, with p-values $\mathbf{p} = [0.0039, 0.1710, 0.0156]$, respectively. With this ordinal logistic regression, only β_{20} is not significant under a 5% significance level and the result is much better than the previous case using the standard logistic regression.

Table 4 lists the ordinal logistic regression results. One can see that there is no decision for 3 LSBs truncation based on the ordinal logistic regression model. This is mainly because very few videos with 3 truncated LSBs are considered acceptable by the participants; also, most of the video testing results with 3 LSBs truncation are considered to be noisy data. When the plain MB percentage (x) is 0.28504 (i.e., 28.504%), $P\{1 \text{ LSB truncated}\} = P\{2 \text{ LSBs truncated}\} = 0.4888$. Accordingly, if $x > 28.504\%$, 1 LSB is truncated; otherwise, 2 LSBs would be truncated. The

developed decision tree model and ordinal logistic regression model only involve very few parameters and the computation time is negligible. The comparison of results between the developed decision tree model and ordinal logistic regression model will be discussed in Section 2.6.

Table 4. Results of ordinal logistic regression

x	P {1 LSB truncated}	P { 2 LSBs truncated }	P { 3 LSBs truncated }	<i>Decision for LSB truncation</i>
0.05	0.0876	0.7261	0.1863	2 LSBs
0.10	0.1354	0.7415	0.1231	2 LSBs
0.15	0.2034	0.7174	0.0793	2 LSBs
0.20	0.2939	0.6560	0.0502	2 LSBs
0.25	0.4043	0.5643	0.0314	2 LSBs
0.28504	0.4888	0.4888	0.0224	2 LSBs
0.30	0.5253	0.4552	0.0195	1 LSB
0.35	0.6434	0.3446	0.0120	1 LSB
0.40	0.7463	0.2463	0.0074	1 LSB
0.45	0.8275	0.1679	0.0046	1 LSB
0.50	0.8866	0.1105	0.0028	1 LSB
0.55	0.9273	0.0710	0.0017	1 LSB
0.6	0.9541	0.0448	0.0011	1 LSB

2.5. Quality Optimized Bit Truncation Design

In this chapter, a new viewer-aware bit-truncation technique was proposed which has less visual quality degradation with the same number of LSBs truncated. Based on the developed bit-truncation technique and models, an energy-quality scalable memory with content adaptation was implemented.

2.5.1. Quality Optimized Bit Truncation

Bit truncation can adjust the video data's bit-depth by disabling LSBs to enable power savings and it has been applied widely in low-power hardware design [8, 21]. In this chapter, viewer-awareness to the hardware-design process was introduced and a new hardware-

implementation scheme was developed for bit truncation with a minimized effect on the viewer's experience.

Suppose that the lowest t LSBs of each luma (Y) byte was truncated. For a given video, the true numerical value for these truncated bits was calculated. However, if all videos in general were deliberated, the true (decimal) value of these truncated t LSBs should be considered a random variable. These truncated t LSBs may express any decimal numbers among $0, 1, 2, \dots, 2^t - 1$, because general prior knowledge that works for all videos does not exist. A crucial question is as follows: *what value should be set/given after the true value of these lowest t bits are truncated?* A natural and intuitive method is to make them all zeros. For example, if the true value of a byte is $10101110(B)$ and three bits are truncated, then the byte's value after truncation is $10101000(B)$. Setting the truncated bits as zeros has been widely adopted by designers [8, 21]. However, in the following proposition, this value is not the best for minimizing the expected mean square error, $E(MSE)$.

Proposition 1. Suppose that the lowest t LSBs of a byte are truncated. Without losing generality, it is assumed that the true value of these bits is evenly distributed. Then, the best value for these t truncated bits, in terms of minimizing $E(MSE)$, is $10 \dots 0(B)$ (with $t - 1$ zeros).

Proof. Let random variable Y indicate the true numerical value which is expressed by the truncated t LSBs. Because Y is evenly distributed, the following probability mass function (pmf) for Y :

$Y =$	0	1	2	...	$2^t - 1$
probability	$1/2^t$	$1/2^t$	$1/2^t$...	$1/2^t$

Let x be the targeted (decimal) value that is set for these truncated LSBs. The goal was to minimize $E(MSE)$:

$$f(x) = \frac{1}{2^t} [(x - 0)^2 + (x - 1)^2 + \dots + (x - (2^t - 1))^2] \quad (\text{Equation 9})$$

Let

$$0 = f'(x) = \frac{1}{2^{t-1}} [x + (x - 1) + \dots + (x - (2^t - 1))] \Rightarrow x = 2^{t-1} - \frac{1}{2} \quad (\text{Equation 10})$$

Because x is an integer, take $x = 2^{t-1} = 10 \dots 0(B)$ (with $t - 1$ zeros).

The significance of Proposition 1 is that it shows the dependence between the value set for the truncated bits and the expected MSE and that it gives the best value, in general. Select 2,000 unique videos randomly, representing 100,000 individual frames, from YouTube-8M [22]. As illustrated in Figure 7, setting the truncated bits to be $10 \dots 0(B)$ (with $t - 1$ zeros) can enable much higher *PSNR* values, thereby providing a better viewing experience for the same videos in the same surroundings.

2.6. Content-Adaptation Video Memory Design

Figure 8 (a) shows the architecture of the proposed viewer-aware dynamic bit-truncation memory with $512 \text{ words} \times 64 \text{ bits}$, which contains 32kb 6T SRAM bit-cells. To enable viewer-aware bit truncation for LSBs, two different bit-line conditioning circuitries are applied to the memory. The normal bit-line conditioning circuitries have a pre-charge unit, write driver, and sense amplifier, and they are connected to the 4 most significant bits (MSBs) in a byte; the remaining bit-lines contain extra circuitry to enable bit truncation, and they are applied for the 4 LSBs in a byte as shown in Figure 8 (b).

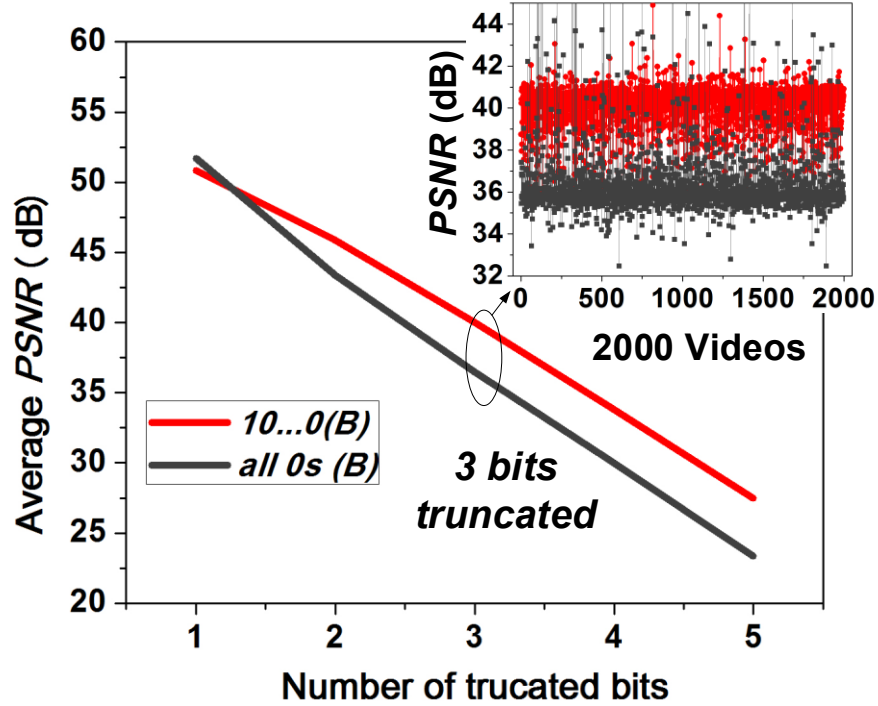
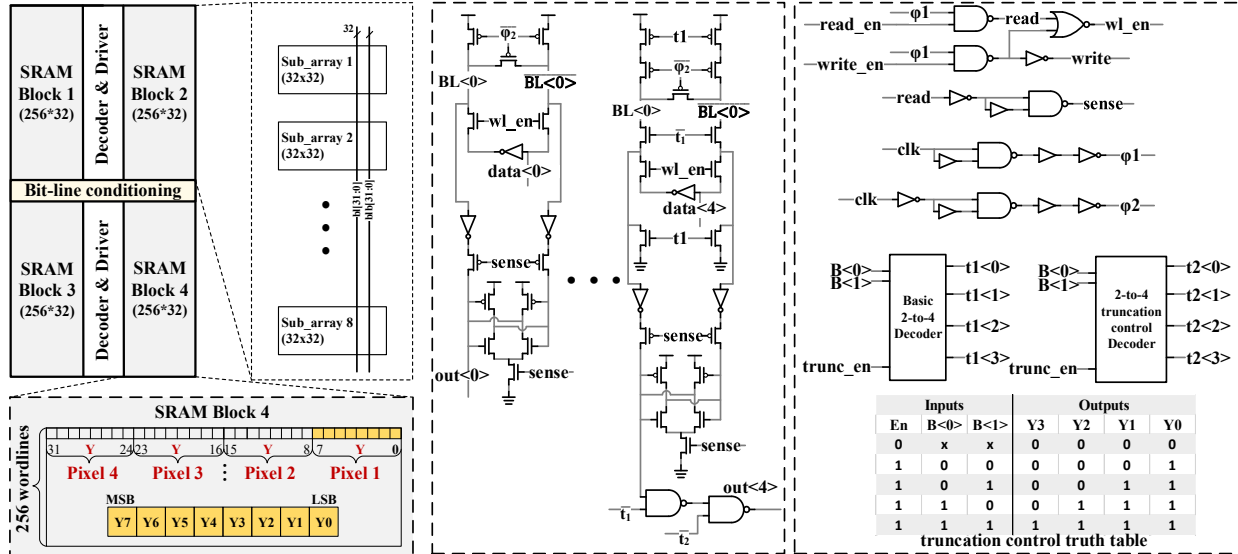


Figure 7. Average PSNR values of 2,000 YouTube-8M videos using two different truncation techniques.



(a) Memory structure (b) Bit-line conditioning circuitry (c) Bit truncation controller

Figure 8. Content-adaptive video memory.

The truncation controller is shown in Figure 8 (c). $\phi 1$ and $\phi 2$ are signals generated from peripheral circuitry based on the clock signal. $\phi 1$ controls read and write operations depending on which period it is in; $\phi 2$ controls the pre-charging circuitry of the memory. The *sense* signal only turns on for a very short time at the end of the reading operation in order to reduce the power consumption during the read operation. The truncation process is controlled by three external signals. *trunc_en* controls whether the truncation function is on, and the other two signals, $B<0>$ and $B<1>$, determine how many bits to truncate. t_1 and t_2 are generated from $B<0>$ and $B<1>$ through two decoders. The decoder for t_1 is a normal 2-to-4 decoder. A special 2-to-4 truncation control decoder is applied for generating t_2 , and the truth table is also shown in Figure 8 (c). When t_1 and t_2 are both **0s**, the normal operations are applied; whenever t_1 is **1**, the pre-charging, writing, and reading operations are suspended; on the basis of t_1 being **1**, if t_2 is **1** then the output will be **0**, otherwise the output will be **1**; the data pattern **01** for t_1 and t_2 will never appear.

The detailed evaluation results including performance, power efficiency, layout, and video quality will be presented in Section 2.6.

2.7. Experimental Results

The proposed memory is implemented based on a 45 nm CMOS technology [28]. In addition to hardware-level implementation and verification, psychological experiments are conducted to test the video output quality from the viewers' perspective.

2.7.1. Speed

Figure 9 shows the timing diagram for the proposed memory. To test the functionality of the memory, the data: $0xe9$, $0xce$, $0x62$, and $0x71$, are written to the addresses: $0x55$, $0xb9$, $0xce$, and $0x15$, respectively, and then read out from the same addresses. For example, during a 3 bit

truncation operation, the values read out are: $0xec$, $0xcc$, $0x64$, and $0x74$, which the last 3 LSBs for these values are **100**(B). The access delay of the reading operation is about 0.5 ns, which is fast enough to deliver the typical mobile video sequences (11MHz for CIF/QCIF and 72MHz for HD720 [29]).

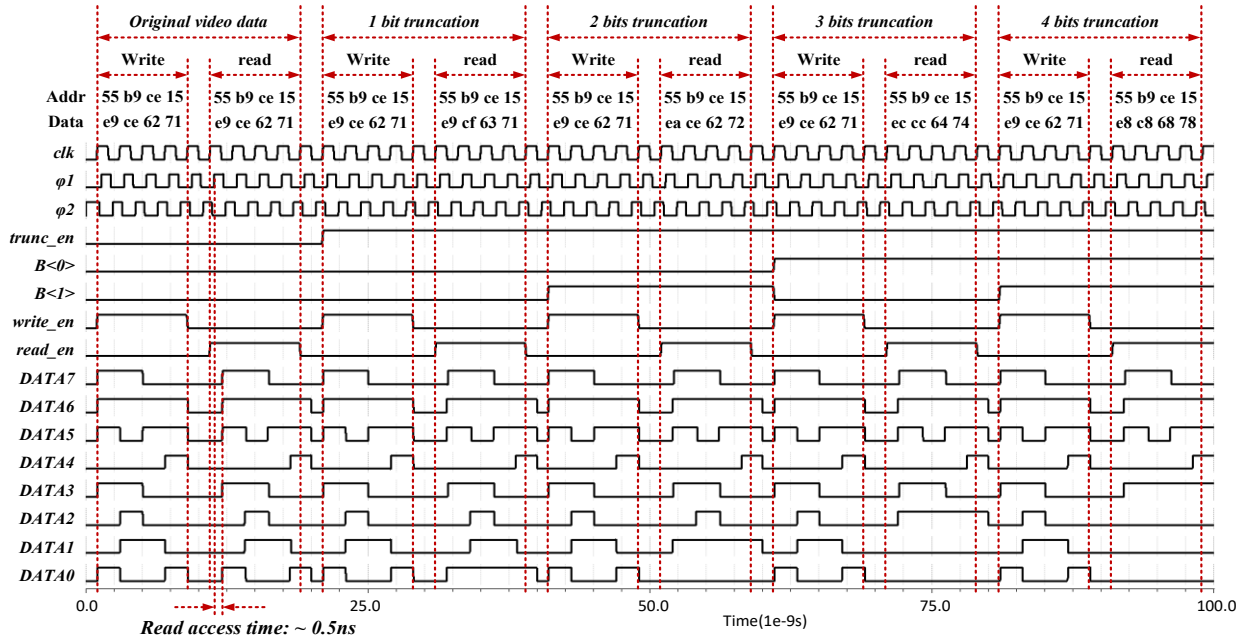


Figure 9. Timing diagram. DATA7: MSB; DATA0: LSB

2.7.2. Layout

The layout design for 512 words \times 64 bits SRAM with viewer-aware bit truncation is shown in Figure 10. Only a few gates are added to the bit-line conditioning circuit to enable the truncation function. Also, after careful design, the decoders for truncation controlling can be fit into the free space of the original layout, without introducing additional overhead. The proposed memory consumes only 0.32% more silicon area as compared to the traditional SRAM, which is negligible.

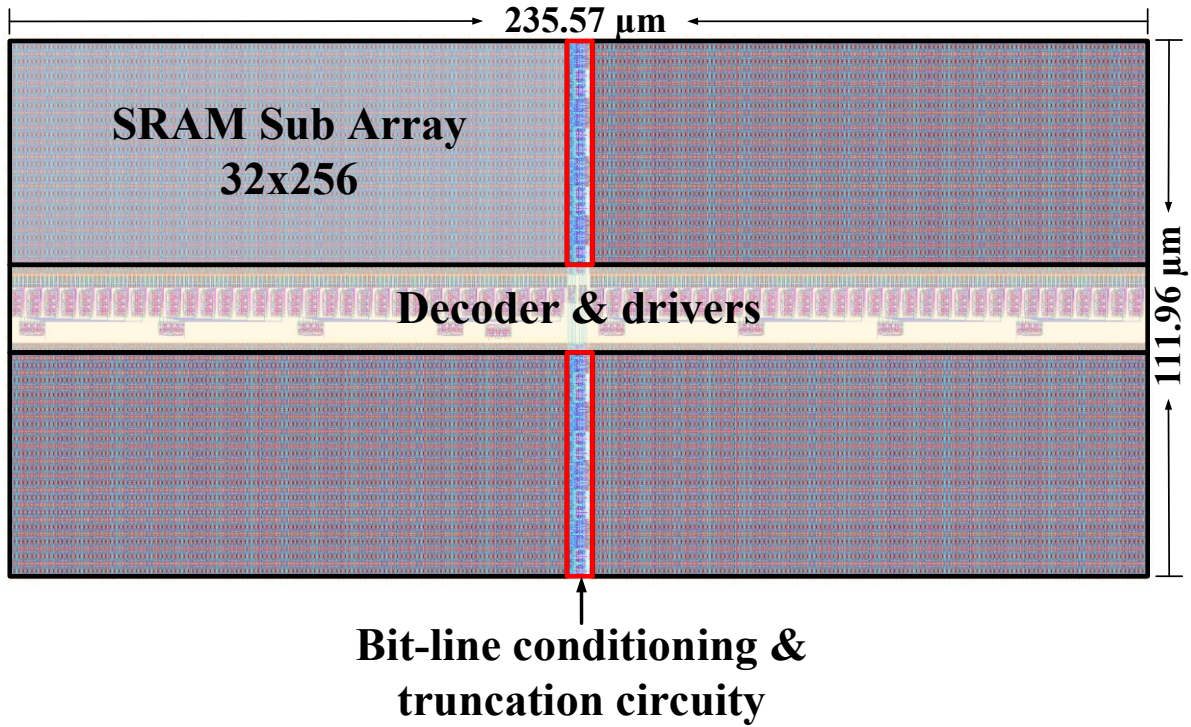


Figure 10. Physical layout design.

2.7.3. Power Savings

Input patterns that cover all data switching possibilities have been tested for the memory. Normal operation, and 1 to 4 LSB truncations, are simulated based on these input patterns, and the power consumption for each scenario is shown in Figure 11. As compared to normal operation, the average power consumption of reading and writing operations for 1 to 4 LSB truncations can enable 13.54%, 20.10%, 26.83%, and 33.31% power savings, respectively.

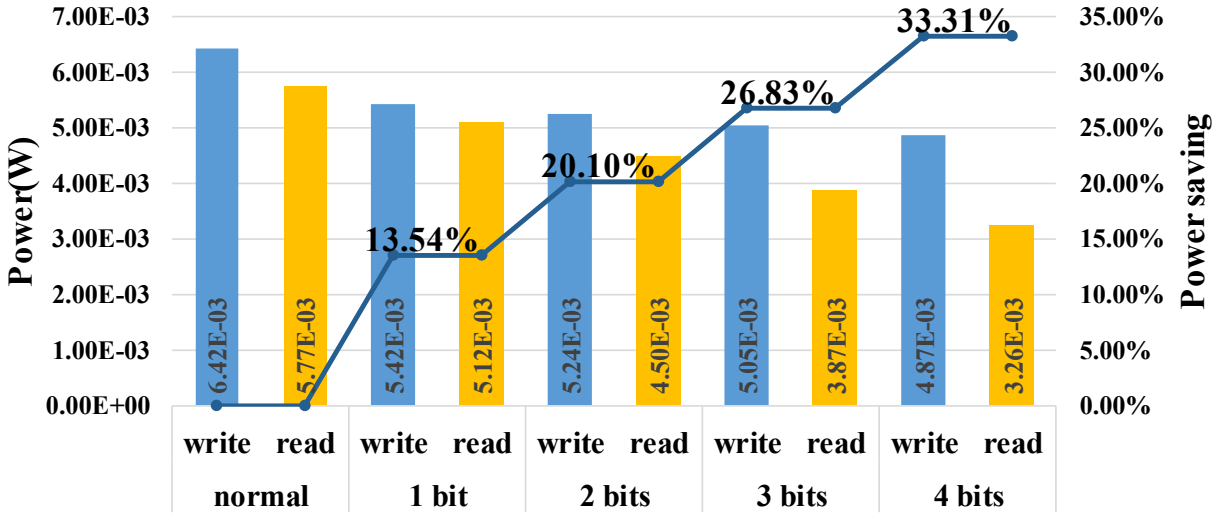


Figure 11. Power savings.

2.7.4. Video Quality

Finally, in order to verify the effectiveness of the technique on the viewer's experience, psychological experiments were conducted at the North Dakota State University Center for Visual and Cognitive Neuroscience. The psychophysical experiment setup is shown in Figure 12. The ambient illumination was provided using a rectangular array of 60 high-intensity LEDs capable of emitting a maximum of 64,000 Lux (Larson Electronics, model LEDP5W-60-D-1227-F5.15). An illumination meter (Extech model 401027) was used to accurately measure the ambient illumination of the phone used for testing, a Samsung Galaxy Note 4. In the experiments, the output of the high-intensity light source was adjusted using neutral-density filters. The luminance level measured by the illumination meter was approximately 811 Lux, which is a typical indoor light level.

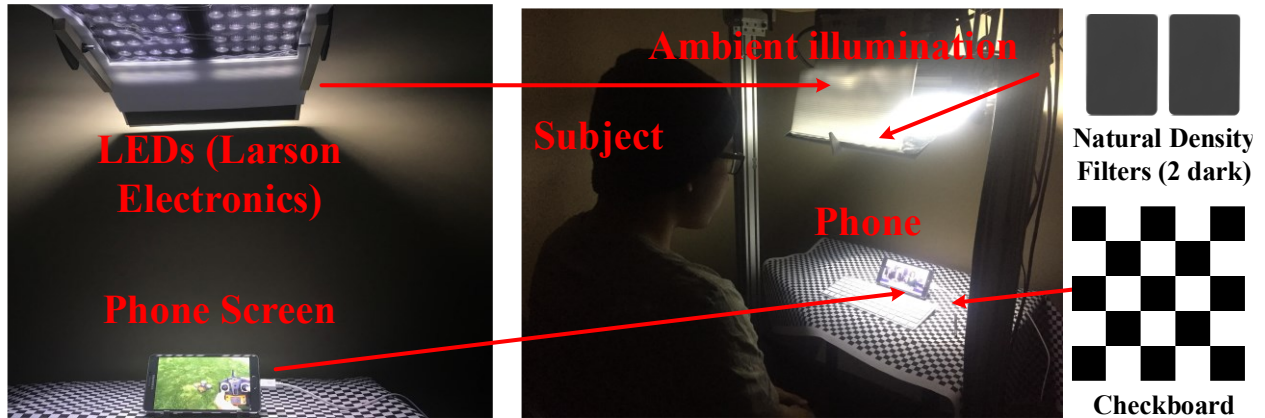


Figure 12. Psychological experiment set-up at North Dakota State University Center for Visual and Cognitive Neuroscience.

To assess the degree to which observers can accept the truncated videos as compared to the reference videos using the developed models, a total of 20 videos were collected: 10 videos that were classified as having a stationary camera and 10 videos containing a reporter. Each video sample was evaluated at a single quality point, encoded using a constant rate factor of 0 (i.e. lossless compression), had a 640×360 resolution, was 10 seconds in length, and was downloaded from [22]. Based on these videos, the average plain MB percentages were calculated, and used the developed models to predict what the expected amount of acceptable LSBs to truncate would be for different videos. Another two versions of each video from the reference were created, one with the predicted amount of acceptable bits to truncate and another with one bit beyond the predicted acceptable amount. Sequences of numbers to represent each video were created and randomized the order they would be presented. During testing, each participant would compare a total of 40 truncated videos to the original, non-truncated version and give their opinion of whether they would consider the video acceptable for viewing on the mobile device.

The testing results for the developed decision tree model are shown in Figure 14. In the analysis, the plain macroblock percentages, the number of bits truncated, and the video quality

metric (VQM) [34] calculation are included for comparison among samples. VQM is one widely used objective video quality metric that has been shown to have a strong correlation to the subjective viewer ratings. When calculating the VQM for each sample, the NTIA General Model was used with Full Reference Calibration, which have been standardized by both the ITU and ANSI [35]. The developed decision tree model works well for nearly all of videos. There was only one video, with tag *wF6lvdXXwc4*, out of 20 videos that was considered to not be acceptable by the vast majority of participants.

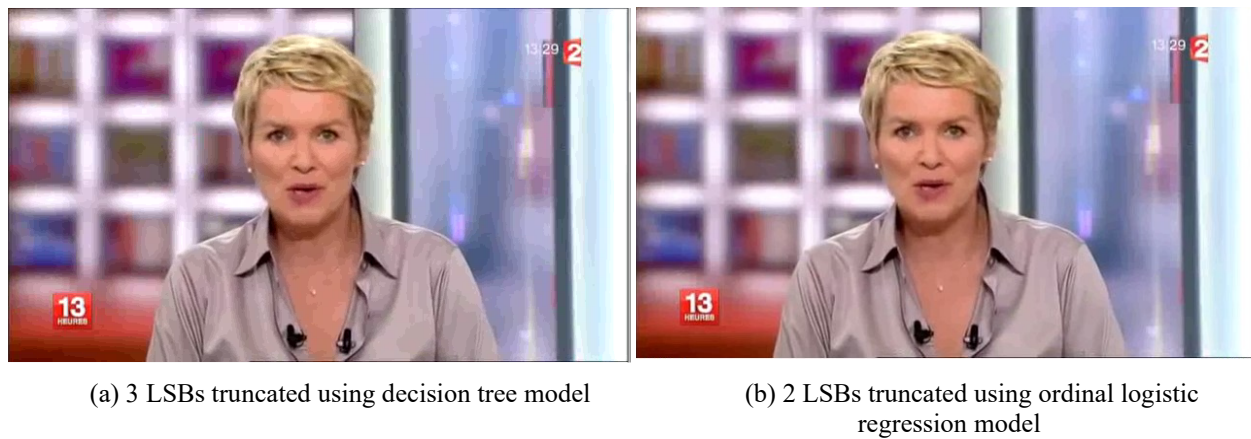


Figure 13. Output quality of video (tag *wF6lvdXXwc4*): (a) with 3 LSBs truncated using decision tree model and (b) with 2 LSBs truncated using the developed ordinal logistic regression model.

As shown in Figure 13 (a), this video displayed banding distortion, caused by bit truncation, appearing on the reporter's face; which is likely the viewer's focus point. Due to this particularly noticeable distortion, viewers were less likely to accept the displayed degradation. All other samples were considered acceptable by the majority of the 15 total participants, with the lowest acceptance rate being 73% for the video with tag *2AQ6rhVhwRc*; another video with banding appearing very close to the viewer's focus point, the kitten playing with a string in the video.

The results were compared further using the ordinal logistic regression to the decision tree model. Those two models achieve the same prediction results for the majority of videos; only 4 out of 20 videos are different. For those 4 videos, decision tree model predicts 3 LSBs truncated, but the ordinal logistic regression model predicts 2 LSBs truncated. One of those 4 videos is the video with tag *wF6lvdXXwc4*; it was the only one that was considered to not be acceptable using the decision tree model. With 2 LSBs truncated predicted by the ordinal logistic regression model, the visual quality is significantly improved, as illustrated in Figure 13 (b). For the other 3 videos (with tags *Lp3HIXOcKCE*, *dgAu_Wsd7Fo*, and *lcVPxLFlq1c*), the visual output with 3 LSBs truncated are acceptable by the majority of participants. Particularly, for the video with tag *dgAu_Wsd7Fo*, all of the participants said it was acceptable. From the above analysis, concluded, that as compared to the decision tree model, the ordinal logistic regression model is a more conservative model which can avoid the worst video quality degradation case, but it may lose energy optimization opportunities for some videos. Another interesting observation that was made during the video testing process is that if the viewer's focus detected is in different videos (e.g., mobile gaze tracker [30]), noticeable degradation in these sensitive areas of videos can be further removed in the future.



Figure 14. Video quality testing results using the decision tree model.

CHAPTER 3. FLEXIBLE LOW COST POWER-EFFICIENT VIDEO MEMORY WITH ECC-ADAPTATION²

In this chapter, a flexible power-efficient video memory is presented that can dynamically adjust the strength of error-correction-code (ECC), thereby enabling power-quality trade-off based on application requirements. Specifically, the bit significance characteristics of video data was utilized to develop a low-cost parity storage scheme that supports both hamming code-74 (ECC74) and hamming code-1511 (ECC1511). Based on this, a flexible memory with three dynamic power-quality adaptation schemes is proposed (i.e., ECC74, ECC1511, and no ECC) to meet different video application requirements. The simulation results in 45nm CMOS technology show that the proposed memory can enable up to 35.37% power savings without a noticeable degradation in video quality, as compared to the conventional design. An integrated ECC encoder/decoder that handles both ECC74 and ECC1511 is designed, which reduces area overhead. To evaluate the effectiveness of the proposed technique, a system-level video storage embedded test platform is developed based on a commercial 65nm SRAM chip, which shows that the proposed technique results in significant supply voltage reduction without noticeable video quality degradation.

3.1. Introduction

With the development of mobile and Internet-of-Things (IoT) technologies, video services today utilize increasingly more battery-powered portable devices, such as cameras,

² The material in this chapter was authored by Hritom Das, Ali Ahmad Haidous, Scott C. Smith, and Na Gong. Ali Ahmad Haidous was in charge of the design, development, and implementation of a variable voltage SRAM test and characterization system platform, based on a three embedded microcontroller setup. He analyzed the feasibility of the technique on real hardware and validated the results. He was also in charge of all simulations and results completed using python. The source code is available in the appendix. Hritom Das was in charge of all circuit design in Cadence, power analysis, and simulation of the proposed ECC memory. Na Gong and Scott Smith were principal and co-principal investigators respectively.

smart phones, unmanned aerial vehicles (UAV), actuators, wearable devices, and various sensors. In addition, the biggest consumer of mobile device data is video, which is projected to continue to increase. According to market research, the amount of data stored on devices will be 4.5 times larger than data stored in data centers, at 5.9 ZB by 2021 [1]. These devices are typically powered by batteries and have small form factors, and therefore have very stringent limitations on energy consumption, computational capability, memory storage, footprint, and cost. Furthermore, video applications consume a large amount of energy due to their large data size and intensive computational requirements. As a typical example, during video-streaming processes, when compressed video bitstreams are received over communication channels, decoded, then displayed on mobile devices, this process consumes considerable power that limits the mobile device's battery life. In particular, major signal-processing units for videos – e.g., motion estimation – require a significant number of calculations and need frequent embedded memory accesses. These embedded memories occupy over 65% of the core area of a video decoder chip [2], and contribute to over 30% of the system's power consumption [3]. Under the resource constraints of IoT devices, video systems, particularly storage systems, are critical to enable various video applications.

This chapter aims to optimize power efficiency of video data storage through adaptive error-correction-code (ECC) schemes. A new adaptive ECC technique is developed, which can effectively select three power-quality tradeoff levels for video applications: hamming code-74 (ECC74), hamming code-1511 (ECC1511), and no ECC. Specifically, this chapter makes the following contributions:

- **New parity storage for video memory:** To minimize parity storage cost, a new parity storage scheme for video memories is developed that utilizes the bit significance

characteristics of video data for both ECC74 and ECC1511. The method utilizes least significant bits (LSBs) to store the parity bits instead of dedicated parity bit storage in addition to data storage, and as a result, avoids additional storage overhead (Section 3.3).

- **Adaptation scheme:** A detailed analyses is ran on the impact of memory failures at low voltages verses video output quality, and then developed an ECC adaptation scheme. Three conditions are concluded: (i) no ECC shall be needed if the memory failure rate is less than 0.01%; (ii) ECC1511 is favored between failure rates of 0.05% and 0.6%; and (iii) ECC74 is favored at failure rates between 0.01% and 0.05%, and above 0.6% (Section 3.4).
- **Memory circuit design:** Based on the developed adaptation scheme, a flexible dynamic energy-quality trade-off video memory circuit was designed and implemented. Moreover, the area overhead was minimized by designing an integrated ECC encoder/decoder that handles both ECC74 and ECC1511 (Section 3.5).
- **Thorough evaluation:** A comprehensive suite of circuit-level simulations was performed on the proposed memory to compare performance, power efficiency, and video quality (Section 3.6).
- **Hardware platform:** Finally, an embedded hardware test platform was developed for video storage to evaluate the effectiveness of the proposed technique using a commercially available SRAM chip (Section 3.7).

While the proposed flexible memory was developed and implemented for video storage, it may also be applicable for other fault-tolerant scenarios. In addition, this chapter focuses on SRAM, which is the most important type of primary embedded memory technology; however, other memory technologies, such as DRAM, flash, or emerging memories, can also benefit from the adaptive ECC scheme.

The organization of the chapter is as follows. A review of low-power video memory designs is provided in Section 3.2; Section 3.3 presents the proposed ECC storage scheme; Section 3.4 discusses the proposed ECC adaptation process; and Section 3.5 presents the designed memory. The evaluation results were discussed in Section 3.6; Section 3.7 presents the developed hardware testing platform and discusses its results; and finally, this chapter is concluded in Chapter 5.

3.2. State of the Art

3.2.1. Video Memory

In the video-streaming process, the original video is compressed, reducing the number of data bits, and then transmitted to mobile devices for decoding over a communication channel. Video processing has become the most important, energy-intensive application used by mobile devices. Figure 15 shows a typical H.264 video decoding and display process. After entropy decoding, inverse quantization (IQ), and inverse transformation (IT), the motion compensator utilizes the previous reconstructed frames, which are stored in the reference frame memory, along with the transmitted motion vectors, to construct new frames. After the frames are decoded, the mobile-display controller periodically sends them from the frame buffer to the display panel. During this process, multiple memories are needed. In particular, the reference frame memory and display memory, which store the decoded video frames, are accessed very frequently; and they have a profound impact on the system's overall cost and power consumption [3, 4]. The proposed adaptive-ECC video memory can be used to implement the reference frame memory and display memory to store the decoded video frames.

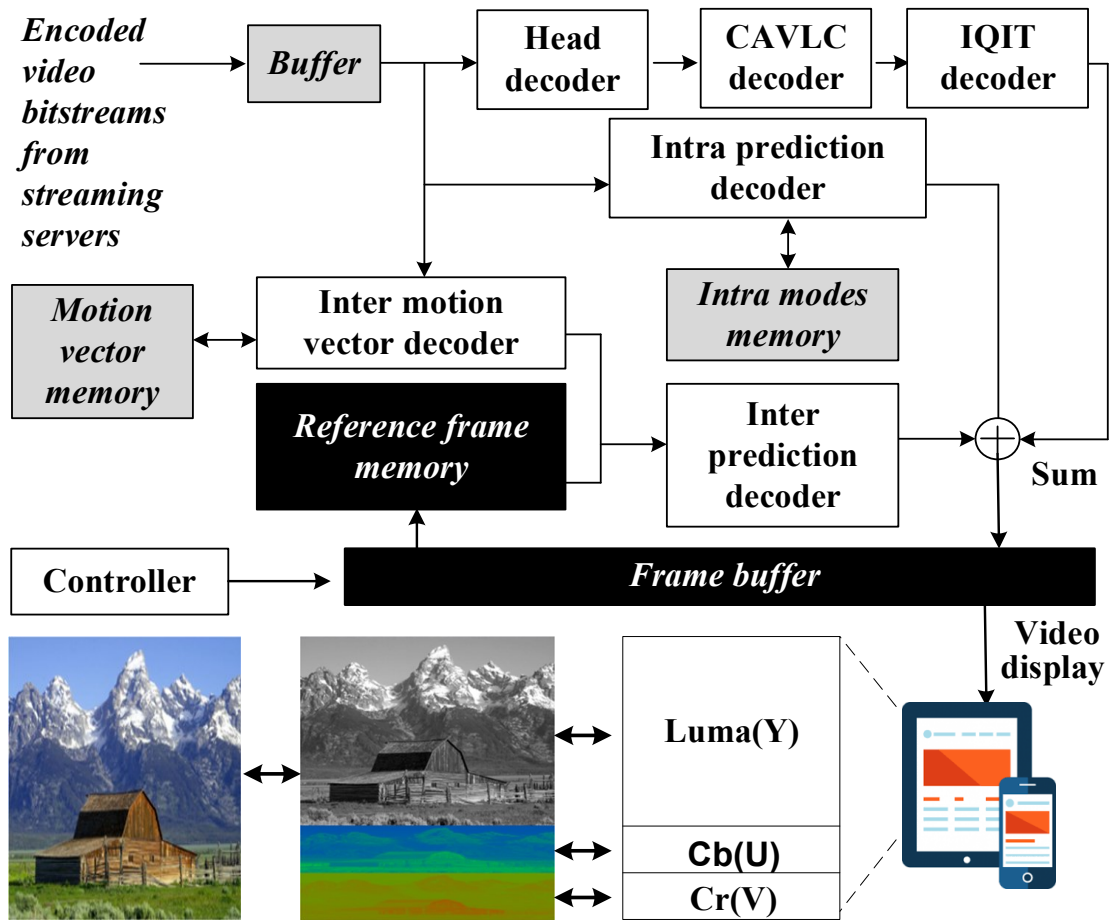


Figure 15. Mobile video memory architecture for steaming. The reference frame memory and frame buffer are accessed very frequently, and have a profound impact on the system’s overall cost and power consumption.

3.2.2. Review of Relevant Literature

Existing as critical hardware-building blocks for today’s approximate computing platforms, video memories show application resilience to approximations with a trade-off between a “good enough” output and additional power savings. State-of-the-art, power-efficient video-specific memory can be broadly classified into two categories: design-time fixed quality or run-time adjustable quality.

3.2.2.1. Video-Specific Memory with Design-Time Fixed Quality:

During the past decade, low-voltage video memories were widely investigated in the literature, and most existing solutions are designs with design-time fixed quality. For example, Chang et al. [5] presented a hybrid 6T+8T SRAM to achieve quality-power optimization. In [6], a heterogeneous sizing scheme was presented to reduce the failure probability of conventional 6T bitcells. In [7], the correlation between most significant bits (MSBs) was utilized to design a hybrid 8T+10T memory for power savings. In [8], advanced data-mining techniques were used to identify useful video data characteristics (e.g., data association) for hardware design. At the same time, several recent works for analyzing the quality of videos, such as viewer experience, have recently been shown to outperform the traditional mean squared error (MSE) and PSNR [9]. Those video-specific memory designs enhanced power efficiency with a reduced implementation cost when compared to general-purpose memories [10]; however, the quality of those designs is fixed during design-time, so they lack run-time adaptation.

3.2.2.2. Adaptive Memory with Dynamic Power-Quality Management:

There were several recent attempts to enable adaptive video memory with dynamic power-quality management. For example, the video memory presented in [11] used the LSBs of video data to store the MSBs' error-correction-code (ECC). In [3], a video content-aware memory technique for power-quality trade-off was developed from viewers' perspectives, based on the influence of video macroblock characteristics on viewer experience. Additionally, in [12], a data-dependent reconfigurable conditional pre-charge (CP) SRAM was designed to utilize statistical dependencies present in the binary values. Furthering these research efforts, this chapter presents a new low-cost adaptive-ECC video memory with dynamic power-quality trade-off. The proposed adaptive-ECC video memory is orthogonal to existing viewer-aware [3] or

data-dependent [12] or negative bitline [11] schemes and therefore can be simultaneously utilized to further optimize power efficiency.

3.3. Proposed Low-Cost ECC Storage Scheme

3.3.1. Traditional ECC

ECC is a very popular technique to enhance the reliability of memory systems [13]. There are various types of ECCs that provide various levels of trade-offs between error correction capability and implementation cost. This chapter utilized the cost-effective hamming code-74 (ECC74) and hamming code-1511 (ECC1511) [11], detailed in Table 5, due to area constraints of the main use-case, video memory. Traditional ECC74 provides protection for 4 message bits, by requiring 3 parity bits to identify a faulty bit location, where each parity bit is generated using 3 message bits. Alternatively, ECC1511 protects 11 message bits with 4 parity bits, where each parity bit is generated using 7 message bits. For ECC74 and ECC1511, only 1 faulty message bit can be detected and corrected. If there are multiple faulty message bits, these two ECC algorithms cannot determine that, and may incorrectly “correct” a message bit.

Table 5. Traditional ECC74 and ECC1511

Traditional ECC74							
2^3	7	6	5	2^2	3	2^1	2^0
N/A	M6	M5	M4	P3	M2	P2	P1
Traditional ECC1511							
2^3	7	6	5	2^2	3	2^1	2^0
P4	M6	M5	M4	P3	M2	P2	P1
2^4	15	14	13	12	11	10	9
N/A	M14	M13	M12	M11	M10	M9	M8

To provide more context for the ECC74 and ECC1511 algorithms, using Table 5 as a visual aid, M0-M14 are message bits and P1-P4 are parity bits. Parity bits are placed on the 2^n ($n=0,1,..$) positions [13]. The calculation of parity bits and error correction bits are based on a specific Hamming code sequence, as expressed below in (Equation 11):

$$\begin{aligned}
P_{1_{74}} &= 3, 5, 7 = M2 \oplus M4 \oplus M6 \\
P_{2_{74}} &= 3, 6, 7 = M2 \oplus M5 \oplus M6 \\
P_{3_{74}} &= 5, 6, 7 = M4 \oplus M5 \oplus M6
\end{aligned}
\tag{Equation 11}$$

The three parity bits for ECC74 were generated by performing XOR (\oplus) operations according to (Equation 11, and utilized even parity. To determine a faulty bit after memory storage for ECC74, the calculation of error correction bits were expressed in Equation 12, where P_N is the previously calculated parity stored then read back, and $P_{N_{74}}$ is the recalculated parity from the read back message data bits:

$$\begin{aligned}
E_{1_{74}} &= P1 \oplus P_{1_{74}} \\
E_{2_{74}} &= P2 \oplus P_{2_{74}} \\
E_{3_{74}} &= P3 \oplus P_{3_{74}}
\end{aligned}
\tag{Equation 12}$$

If the binary to decimal conversion of $(E_{3_{74}}, E_{2_{74}}, E_{1_{74}})$ is evaluated as zero, then there is no error; otherwise, the bit-flip error position is determined by the evaluated decimal number. For example, suppose $E_{3_{74}} = '1'$, $E_{2_{74}} = '1'$, and $E_{1_{74}} = '0'$, then the faulty bit position is 110_2 which corresponds to the 6th bit, M5, having a bit-flip error; hence, M5 is toggled to correct the error. A major disadvantage with traditional ECCs is their significant cost overhead, caused mainly by additional bitcells required for storing parity bits, which results in significant area and power overhead. For ECC74, protecting 4 message bits required 3 parity bits: a 75% silicon area overhead. For ECC1511, protecting 11 message bits required 4 parity bits: a 36% silicon area overhead. Therefore, this excessive overhead eliminated traditional ECCs for data integrity solutions in embedded memory designs, as the use-case required optimized resource allocation. Instead of additional parity bits for message protection, if a use-case was identified where the parity bits can be embedded within the message bits, then the area overhead requirements of ECCs would be eliminated. One such use-case is videos, where replaced legitimate message bits with parity bits, and by doing so, the trade-off is video quality for low-power.

Table 6. Impact of Traditional ECC on Video

Traditional ECC74: Byte 1								
Memory bits	MSB				LSB			
	S7	S6	S5	S4	S3	S2	S1	S0
Data	M7	M6	M5	M4	P3	M2	P2	P1
Traditional ECC1511: Byte 1								
Memory bits	MSB				LSB			
	S7	S6	S5	S4	S3	S2	S1	S0
Data	P4	M6	M5	M4	P3	M2	P2	P1
Traditional ECC1511: Byte 2								
Memory bits	MSB				LSB			
	S15	S14	S13	S12	S11	S10	S9	S8
Data	M15	M14	M13	M12	M11	M10	M9	M8



(a) Original Frame

(b) Encoded ECC74

(c) Encoded ECC1511

Figure 16. Video output quality with traditional ECC. (a) Original frame, (b) ECC74 parity bits stored with PSNR = 27.75 dB, and (c) ECC1511 parity bits stored with PSNR = 8.27 dB.

For example, suppose the traditional ECC scheme in Table 5 was directly applied to the memory system in Table 6. Video pixel data is typically organized in bytes. Here, the popular mobile video format YUV 4:2:0 is considered. Every four pixels will have six bytes, including four luma bytes and two chroma bytes. As shown in Table 6, S0 to S15 are the bit positions for two bytes S0 (Least Significant Bit [LSB]) to S7 (Most Significant Bit [MSB]) were the bit positions of the 1st byte and S8 (LSB) to S15 (MSB) are the bit positions of the 2nd byte of the memory array, respectively. The three parity bits for ECC74 are stored in S3, S1, and S0. The four parity bits for ECC1511 are stored in S7, S3, S1, and S0. Figure 16 (a), (b), and (c)

presented the original video frame, encoded video frame with traditional ECC74, and encoded video frame with traditional ECC1511, respectively.

Peak Signal-to-Noise Ratio (PSNR) is a widely adopted video quality evaluation metric, where a higher PSNR value translated to better video frame quality. Storing the parity bits using the traditional ECC schemes resulted in significant video quality degradation, due in part to video data carrying more quality weight in the MSBs. The PSNR of the encoded frames with traditional ECC74 and ECC1511 are 27.75dB and 8.27dB, respectively. As one observes from Figure 16 (c), ECC1511 encoding resulted in larger video quality loss as opposed to ECC74: due to its 4th parity bit being stored in the MSB of the first byte. Consequently, to ensure the least amount of video quality degradation, video data bit significance characteristics should be considered, such that LSBs are favored for parity storage.

3.3.2. Bit Significance Characteristics of Video Data and Proposed Storage Scheme for Parity Bits

As a typical fault-tolerant application, video data has bit significance characteristics where MSBs have a greater contribution to output quality than LSBs. According to recent literature, the video memory presented in [11] used the LSBs to store the MSBs' parity bits, thereby effectively reducing video quality degradation overhead; this is what was used as a basis for comparison. As a caveat however, only ECC1511 is considered in [11]. In this chapter, a flexible memory was proposed with three dynamic power-quality adaptation schemes to meet the various requirements of video applications, i.e., ECC74, ECC1511, and no ECC. Also, an integrated ECC encoder/decoder was designed that handled both ECC74 and ECC1511, which further reduced area overhead.

The proposed storage scheme is applied for two-byte memory, as illustrated in Table 7. Here, S represents the bit position in the memory array. If ECC1511 is selected, then there would be four parity bits stored in the LSBs (i.e., P1, P2, P3, P4) and eleven protected MSB message bits (i.e., M7 to M2, M15 to M11). Alternatively, if ECC74 is selected, then there would be three parity bits stored in the LSBs (i.e., P1, P2, P3) and four protected MSB message bits (i.e., M6, M7, M14, M15). (Equation 13 is used to calculate the parity bits for the proposed scheme. The symbols in (Equation 13, M , P and (3,5,7,9,11,13,15), indicate the message bit, parity bit, and ECC sequence respectively.

Table 7. Proposed ECC

Proposed ECC74: Byte 1								
Memory bits	S7	S6	S5	S4	S3	S2	S1	S0
ECC sequence	3	6	9	11	13	15	2 ¹	2 ⁰
Data	M7	M6	M5	M4	M3	M2	P2	P1
Proposed ECC74: Byte 2								
Memory bits	S15	S14	S13	S12	S11	S10	S9	S8
ECC sequence	5	7	10	12	14	16	8	2 ²
Data	M15	M14	M13	M12	M11	M10	M9	P3
Proposed ECC 1511: Byte 1								
Memory bits	S7	S6	S5	S4	S3	S2	S1	S0
ECC sequence	3	6	9	11	13	15	2 ¹	2 ⁰
Data	M7	M6	M5	M4	M3	M2	P2	P1
Proposed ECC 1511: Byte 2								
Memory bits	S15	S14	S13	S12	S11	S10	S9	S8
ECC sequence	5	7	10	12	14	16	2 ³	2 ²
Data	M15	M14	M13	M12	M11	M10	P4	P3

$$\begin{aligned}
P_1 &= 3, 5, 7, 9, 11, 13, 15 \\
&= M7 \oplus M15 \oplus M14 \oplus M5 \oplus M4 \oplus M3 \oplus M2 \\
P_2 &= 3, 6, 7, 10, 11, 14, 15 \\
&= M7 \oplus M6 \oplus M14 \oplus M13 \oplus M4 \oplus M11 \oplus M2 \\
P_3 &= 5, 6, 7, 12, 13, 14, 15 \\
&= M15 \oplus M6 \oplus M14 \oplus M12 \oplus M3 \oplus M11 \oplus M2 \\
P_4 &= 9, 10, 11, 12, 13, 14, 15 \\
&= M5 \oplus M13 \oplus M4 \oplus M12 \oplus M3 \oplus M11 \oplus M2
\end{aligned} \tag{Equation 13}$$

Since ECC74 does not require the 4th parity bit, P4, as a result this function is disabled when using ECC74. Furthermore, to reduce circuit area overhead, the encoders and decoders are developed for ECC1511 then reused for ECC74 by only using the green colored bits in (Equation 13 to calculate P1-P3 when ECC 74 is selected. The encoder/decoder design is detailed in Section 3.5.

Figure 17 (a) and (b) present the video output quality using the proposed ECC storage scheme from Table 7. Figure 17 (a) shows the video quality PSNR metric for ECC74 as 41.26dB with 2 parity bits stored in the 2 LSBs of the 1st byte and 1 parity bit stored in the LSB of the 2nd byte. Figure 17 (b) shows the video quality PSNR metric for ECC1511 as 39.84dB with 2 parity bits stored in the 2 LSBs of both the 1st and 2nd bytes. Since ECC1511 needed to sacrifice one extra LSB to store its parity bits, this ECC scheme resulted in a lower PSNR value than ECC74. Hence, the proposed ECC parity storage scheme from Table 7, as shown in Figure 17, which supported both ECC74 and ECC1511, had much better potential to significantly improve video quality compared to the traditional ECC scheme from Table 6, as shown in Figure 16. Based on this parity storage scheme, an adaptive ECC mechanism is proposed to meet various requirements of video applications, as discussed in the next section.

3.4. ECC Adaptation Based on Requirements and Failure Rate Based on Voltage

In this section, an adaptive ECC mechanism is presented, which supports three ECC conditions, no ECC, ECC74, and ECC1511. First, SRAM failure characteristics are studied using a 45 nm CMOS technology. Then, the impact of memory failures on video quality is analyzed, including failures in parity bits. Finally, a memory failure based adaptive ECC mechanism is developed.

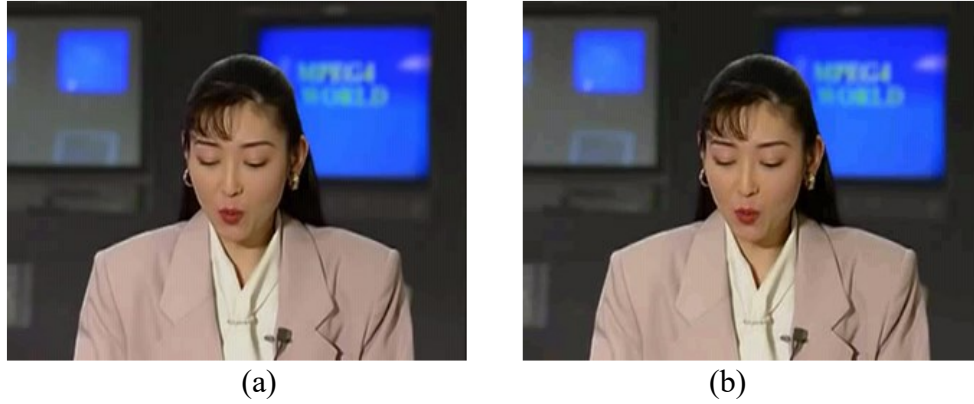


Figure 17. Encoded video frame (Akiyo) with (a) ECC74 where parity is stored in the LSBs with PSNR = 41.2582 and (b) ECC1511 where parity bits were stored in LSBs with PSNR = 39.8426 dB.

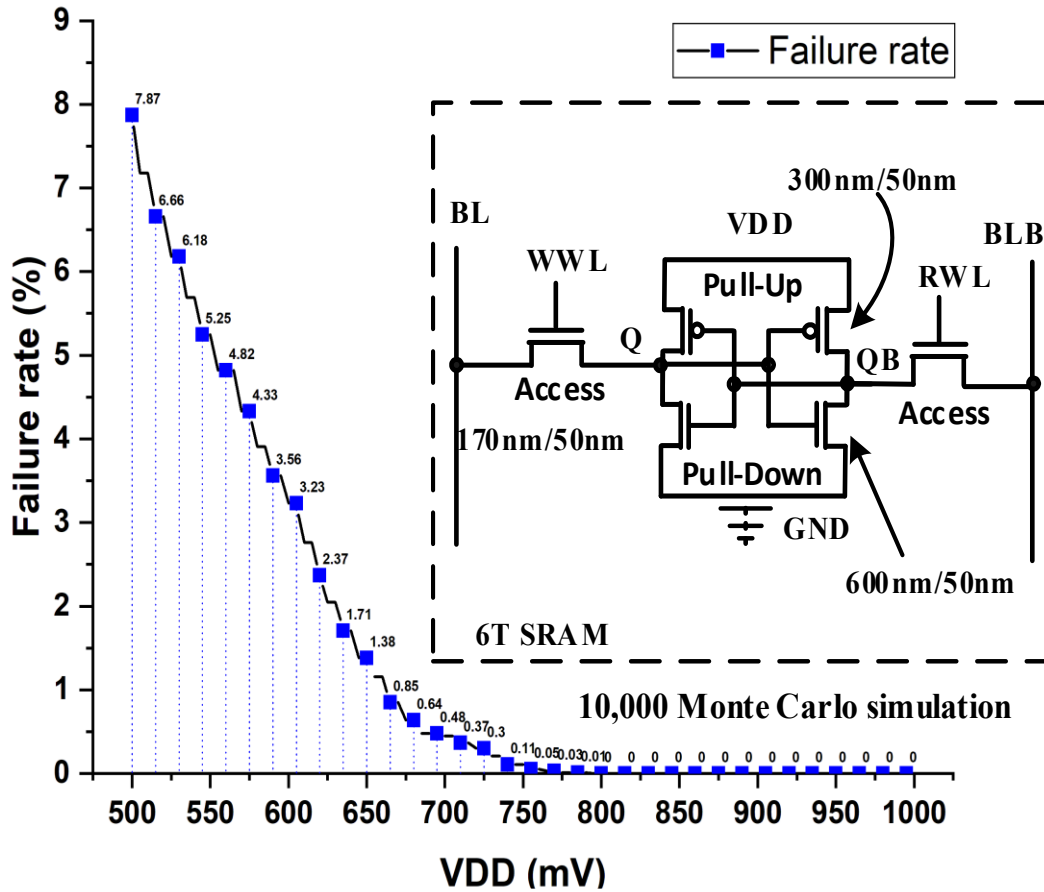


Figure 18. Relation between supply voltage (VDD) and SRAM bitcell failure rate in a 45nm CMOS technology.

3.4.1. Failure Characteristics of 6T SRAM

Figure 18 shows the failure rate of a 45 nm 6T SRAM bitcell at an increasing voltage range between 500mV and the technology’s 1.0V nominal supply voltage, with 5mV increments. The failure rate is measured with 10,000 Monte Carlo simulations at the worst process corner for 6T bitcells, fast NMOS and slow PMOS (FS). As expected, the failure rate increased rapidly as the supply voltage is reduced. The failure rate is about 7.87% at 500mV; and when the supply voltage is scaled-up to 685mV, the failure rate is about 0.48%. There are no errors in the memory system when the supply voltage is 795mV or above. Next, the impact of memory failures on video output quality was studied.

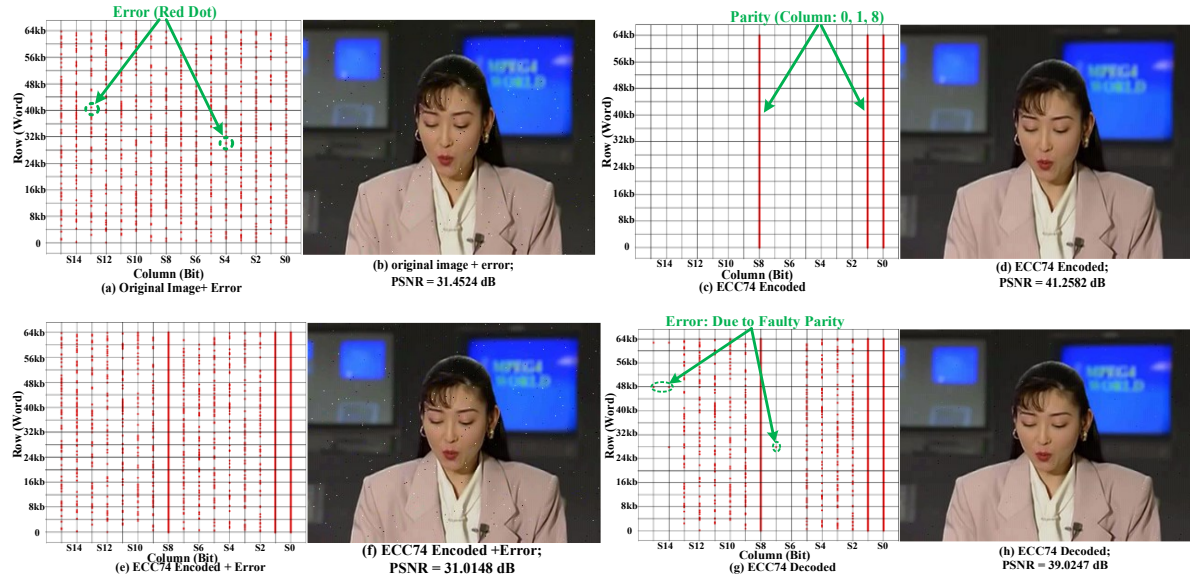


Figure 19. Error map and video quality with the proposed ECC74 under 0.1% faulty memory bitcells: (a) and (b) Error map and original image with 0.1% memory failures injected; (c) and (d): parity bits stored in the LSBs with the proposed ECC74, and corresponding video quality; (e) and (f): Error map and encoded video quality with the proposed ECC74, with the exact same number and position of errors from (a) injected into the memory; (g) and (h): Error map and decoded video quality with the proposed ECC74.

3.4.2. Errors Injected, Including in Parity Bits

First, the impact of memory failures on video quality with failures injected in all bits was analyzed, including parity bits, at a uniform random distribution of 0.1%, on the well-known video sequence – Akiyo.

Figure 19 illustrates the error mapping and quality of a video that is stored in a 65536 word \times 16bit SRAM array, with supply voltage at 665mV, using the proposed ECC74 scheme. Figure 19 (a) and (b) demonstrates the error distribution of the SRAM memory with 0.1% failures injected in the original image. In Figure 19 (a), the dots, two of which are circled in blue as an example, represented the error positions in the SRAM memory array. As can be seen, the memory failures are distributed uniformly in the MSBs and LSBs, and the PSNR is 31.4524dB, as shown in Figure 19 (b). In Figure 19 (c), the parity bits of ECC74 are stored in the LSBs of the SRAM array, based on the proposed parity storage scheme. Specifically, the two LSBs of the 1st byte and one LSB of the 2nd byte are utilized to store the parity bits. The video output quality with ECC encoding is illustrated in Figure 19 (d), having a PSNR of 41.2582dB. After storing the parity bits in the memory, the exact same number and position of errors from Figure 19 (a) are injected into the memory in Figure 19 (e), resulting in a PSNR of 30.0148dB. Finally, after decoding, ECC74 could correct one error in either of the 2 MSBs of every two sequential bytes of the SRAM memory array, as shown in Figure 19 (g), this resulted in a 7.57dB PSNR improvement compared to the original image without ECC74 (i.e., Figure 19 (b) vs. (h)). Note that the error map in Figure 19 (g) shows some new errors, circled in purple, due to injected memory failures in the parity bits, which result in an incorrect ECC correction.

Figure 20 illustrates the error map and the video output quality for ECC1511. It can be seen from Figure 20 (a) that the additional LSB parity bit caused a slight image quality degradation compared to ECC74 (i.e., Figure 20 (b) with a PSNR of 39.8426dB vs. Figure 19 (d) with a PSNR of 41.2582dB). Figure 20 (c) and (d) used the same 0.1% error map used for the previous ECC74 analysis, which resulted in the PSNR being slightly degraded to 30.8653dB, due to the extra parity bit. Figure 20 (e) shows that the decoder circuit corrected most of the injected errors in the 11 protected MSBs, which resulted in a PSNR of 39.2536dB in Figure 20 (f), which is slightly higher than when using ECC74 (i.e., Figure 19 (h)). Hence, even though ECC1511 sacrificed one extra LSB for parity, its stronger error-correction ability improved video quality by correcting more MSB errors.

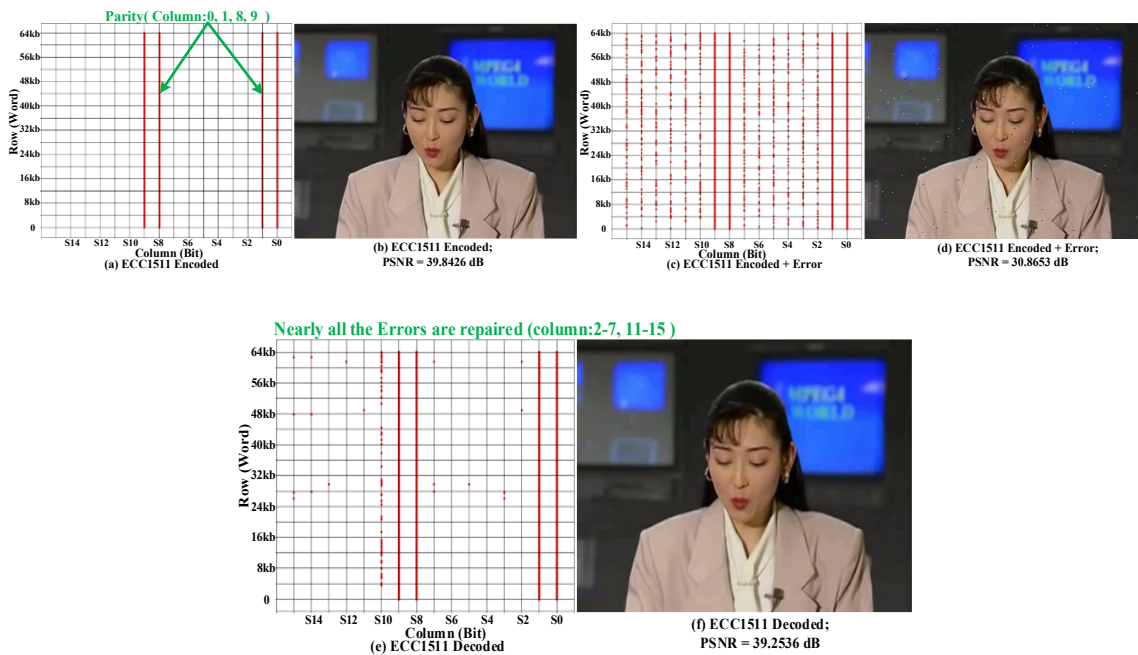


Figure 20. Error map and video quality with the proposed ECC1511 under 0.1% faulty memory bitcells: (a) and (b) parity bits stored in the LSBs with the proposed ECC1511, and corresponding video quality; (c) and (d): Error map and encoded video quality with the proposed ECC1511; (e) and (f): Error map and decoded video quality with the proposed ECC1511.

So far, the proposed ECC scheme was analyzed at a low 0.1% failure rate; in the next sub-section, continued are the analysis of its performance at different failure rates.

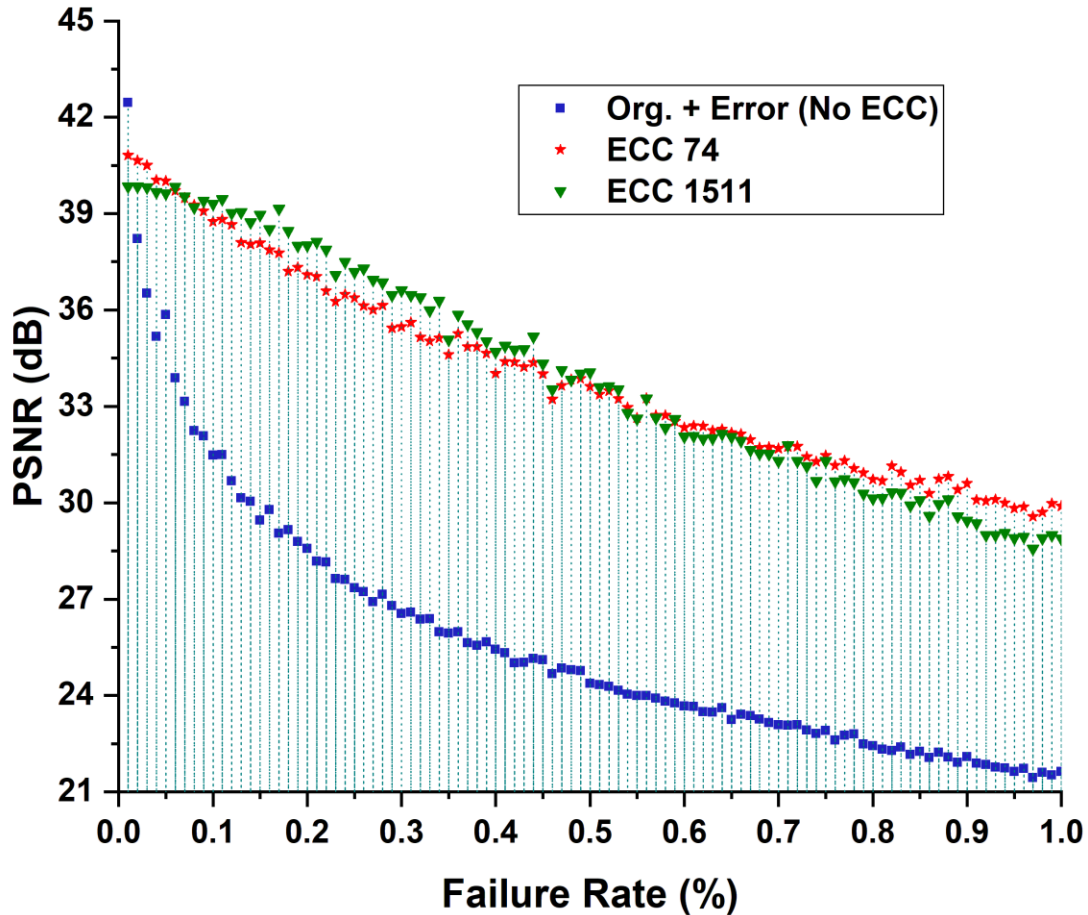


Figure 21. ECC adaptation based on failure rate and corresponding PSNR.

3.4.3. ECC Under Various Failure Rates

To further analyze the performance of the proposed ECC schemes at various failure rates with different videos, 100 videos are randomly selected from YouTube-8M [14] for evaluation. The output quality of the same randomly selected frame from each video is tested for a range of failure rates between 0.01% to 1%, as shown in Figure 21. It can be seen that both ECC schemes significantly increase video quality compared to without ECC, except for when the failure rate is less than or equal to 0.01%, since for very low failure rates, the message bits replaced by parity bits caused more PSNR loss than PSNR gain due to faulty bit corrections. When the memory failure rate is between 0.01% and 0.05%, ECC74 performs best: since the additional message bit replaced by ECC1511's 4th parity bit caused more PSNR loss than PSNR gain due to additional

faulty bit corrections. When the memory failure rate is greater than 0.05% and less than 0.6%, ECC1511 performs best, since the PSNR gained by its increased faulty bit correction outweighs the PSNR loss due to its extra parity bit. However, for failure rates over 0.6%, ECC74 is best due to its reduced possibility of multi-bit errors compared to ECC1511. As expected, video quality degrades as memory failure rate increases, even for the ECC schemes, since they can only correct a single error bit for every 2-bytes of memory. As the failure rate increases, the likelihood of multiple bit errors per 2-bytes also increases: which cannot be corrected with either ECC74 or ECC1511. Hence, the proposed ECC scheme performs best when the error rate is within an acceptable range, where the likelihood of multi-bit errors is small.

Next, a memory failure based ECC scheme is proposed to enable runtime adaptation.

3.4.4. Proposed Runtime ECC Adaptation Scheme

According to [15], video quality is deemed acceptable when PSNR is 30dB or higher. Since ECC74 has a PSNR of 29.88755dB at ~1%, this is within the acceptable range. Hence, if the memory works at its nominal supply voltage or the error rate is lower than 0.01%, no ECC is needed. For failure rates between 0.01% and 0.05%, ECC74 should be selected. As the failure rate continues to increase (between 0.05% and 0.6%), ECC1511 should be utilized due to its stronger error correction ability. And, when the failure rate is above 0.6%, both ECC74 and ECC1511 cannot correct multiple bit errors, but ECC74 should be selected since it has fewer parity bits that can have errors that cause incorrect ECC correction, and therefore performs better. The next section describes the hardware implementation of this proposed runtime ECC adaptation scheme.

3.5. Proposed Memory

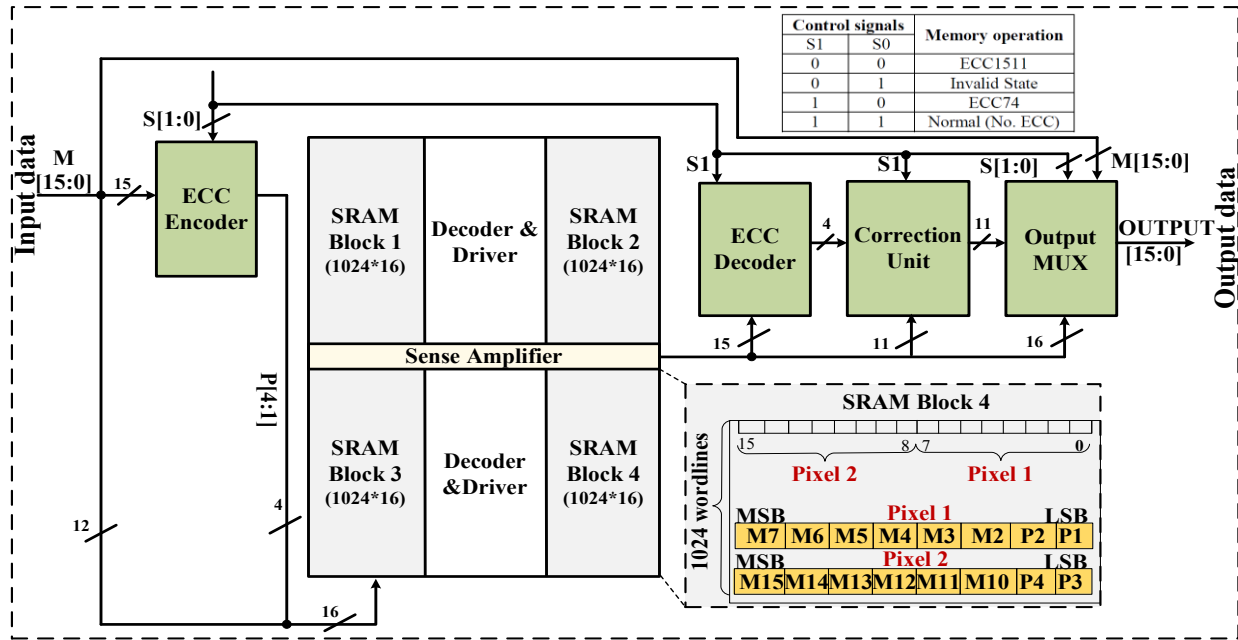


Figure 22. Proposed Adaptive ECC Memory.

Figure 22 presents the architecture of the proposed adaptive ECC memory with its bitcells organized in four 1024×16 bit sub-blocks. Based on the traditional memory structure, ECC Encoder/Decoder, Correction Unit, and Output MUX are needed to enable ECC adaption. For each read/write operation, a 4-to-1 multiplexer with two control signals (S1 and S0) are used to select the correct operation as follows: (i) when S1 and S0 are “00”, ECC1511 is activated; (ii) when S1 and S0 are “10”, ECC74 is selected; and (iii) when S1 and S0 are “11”, no ECC is selected and a normal read/write operation is executed. An S1 and S0 of “01” is invalid and did not occur in a properly operating system; however, if this does occur for some reason, a normal read/write operation without ECC is executed. As shown in Figure 22, the input data [15:0], excluding M10, is provided to the encoder to generate the parity bits or pass the original LSB message data, depending on the control signals, S0 and S1. Then, the data is sent to the memory for storing. During this process, the first two LSBs of both pixels/bytes may be replaced with

parity bits after encoding, depending on which of the 3 ECC schemes is selected (i.e., if ECC74 is selected, M0, M1, and M8 are replaced with P1, P2, and P3, respectively; if ECC1511 is selected, in addition to the M0, M1, and M8 replacements, M9 is also replaced with P4; and if no ECC is selected, then no message bits are replaced). When reading from the memory, the data is sent to the decoder and correction unit circuitry to check for, and correct a faulty bit if needed, respectively. If either ECC scheme is selected, then the Output MUX selects the final output from the Correction Unit; otherwise, it selects the memory output as the final output. To minimize implementation cost, the ECC encoder/decoder are designed to reuse circuitry for both ECC1511 and ECC74, which is discussed next.

3.5.1. Reusable ECC Encoder for ECC1511 and ECC74

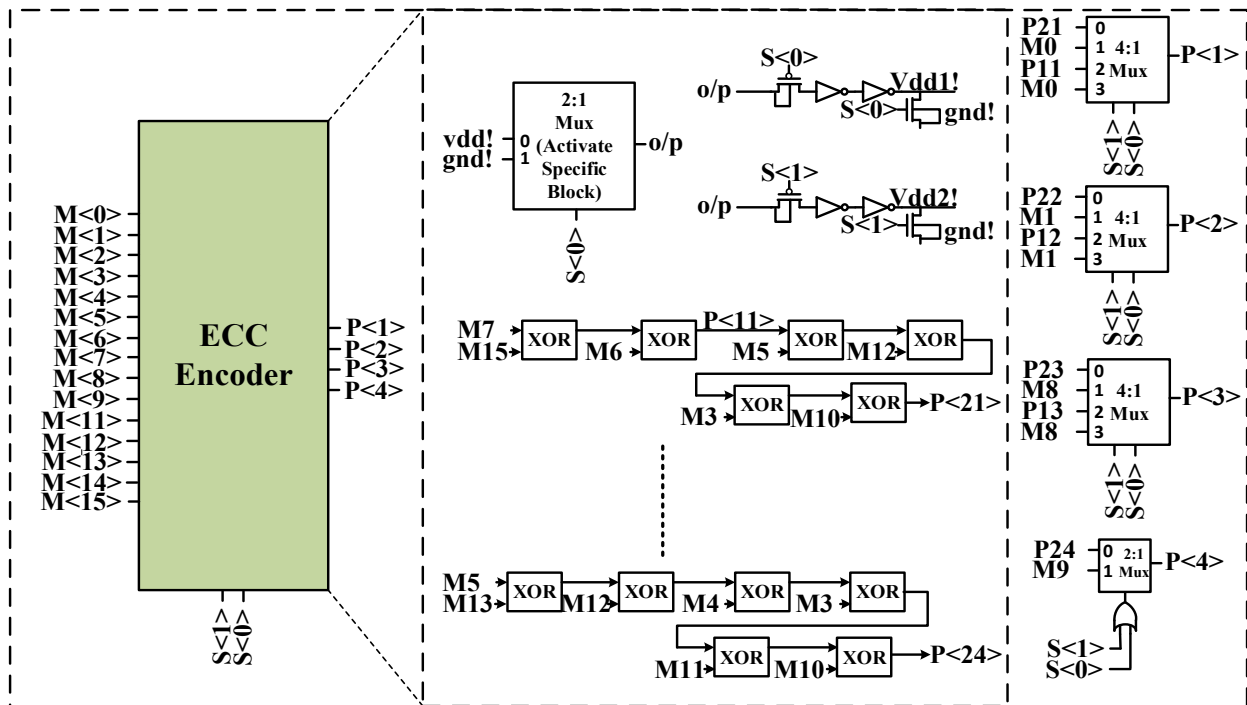


Figure 23. ECC Encoder.

Figure 23 shows the integrated ECC Encoder for ECC1511 and ECC74. There is a 15-bit message input (M[15:0], excluding M10) and two control signals (S0 and S1) for ECC selection; and the encoder generates four parity bits (P[4:1]) for calculation of error bit detection and

correction, only if ECC is selected. If ECC is selected (i.e., $S_0 = 0$), then $V_{dd1!}$ is enabled to supply the ECC74 encoding circuitry (i.e., the first 2 XOR gates in the first 3 XOR chains); if $S[1:0] = "10"$, then $V_{dd2!}$ is also enabled to supply the additional circuitry needed for ECC1511 encoding (i.e., the 4th XOR chain and the rest of the XOR gates in the first 3 XOR chains); otherwise (i.e., $S[1:0] = "01"$ or $"11"$) both $V_{dd1!}$ and $V_{dd2!}$ are kept at ground so that the encoder circuitry is inactive, therefore conserving power, since ECC is not being utilized. The 4 output MUXes then selected which parity bit or original message bit to store in the 2 LSBs of both bytes of memory. As an example, if ECC74 is activated, then P11, P12, P13, and M9 would be stored in P1, P2, P3, and P4, respectively.

3.5.2. Reusable ECC Decoder for ECC1511 and ECC74

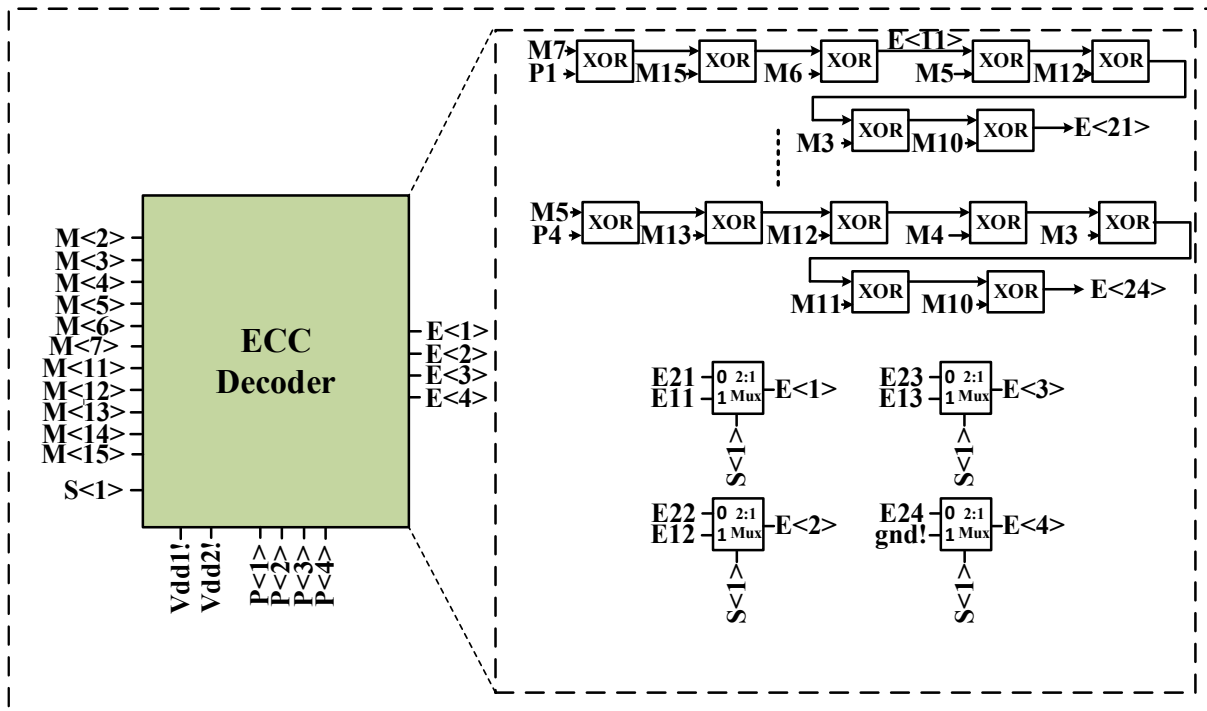


Figure 24. ECC Decoder.

Figure 24 shows the integrated ECC Decoder design. Its input signals include data signals ($M_2 \dots M_7$) and ($M_{11} \dots M_{15}$), parity bits ($P_1 \dots P_4$), and one control signal, S_1 . It generates seven

internal signals (E13, E12, E11 for ECC74 and E24, E23, E22, E21 for ECC1511), which are then grouped into a 4-bit number, E[4:1], which represents the error bit position, if any, in the 2 memory bytes. For example, if E[4:1] = “0111”, then message bit 14 is faulty according to Table 7 (i.e., 7 corresponds to M14), and would be toggled in the subsequent Correction Unit. Similar to the Encoder, Vdd1! is used to supply the XOR gates that generate E13, E12, E11 and the output MUXes that generate E[4:1], and Vdd2! is used to supply the additional XOR gates needed to generate E24, E23, E22, E21, in order to conserve power.

3.5.3. Correction Unit

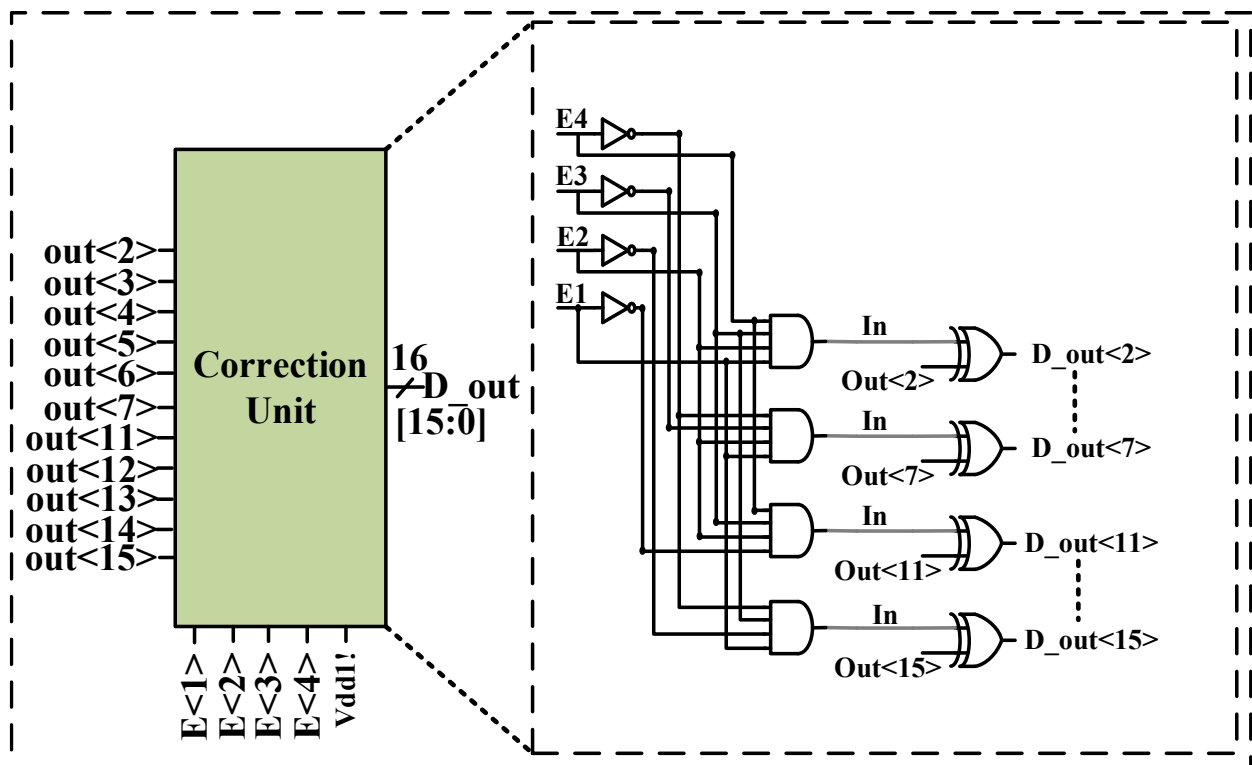


Figure 25. Correction Unit.

Figure 25 presents the error bit correction unit, which flips a message bit identified as faulty by E[4:1]. An active high 4-to-16 decoder is used to select a specific faulty bit location by asserting In, which is input to an XOR gate along with its corresponding message bit, such that the message bit is flipped when In is asserted. For example, the top most In in Figure 25 is

asserted when $E[4:1] = "1111"$, which according to Table 7 corresponded to M2; hence, its corresponding XOR gate input is $Out(2)$, and that XOR gate's output is $D_out(2)$. Note that message bits M10, P1, P2, P3, and P4 can never be corrected; hence, $out[10:8]$ and $out[1:0]$ are passed directly to $D_out[10:8]$ and $D_out[1:0]$, respectively.

3.5.4. Output MUX

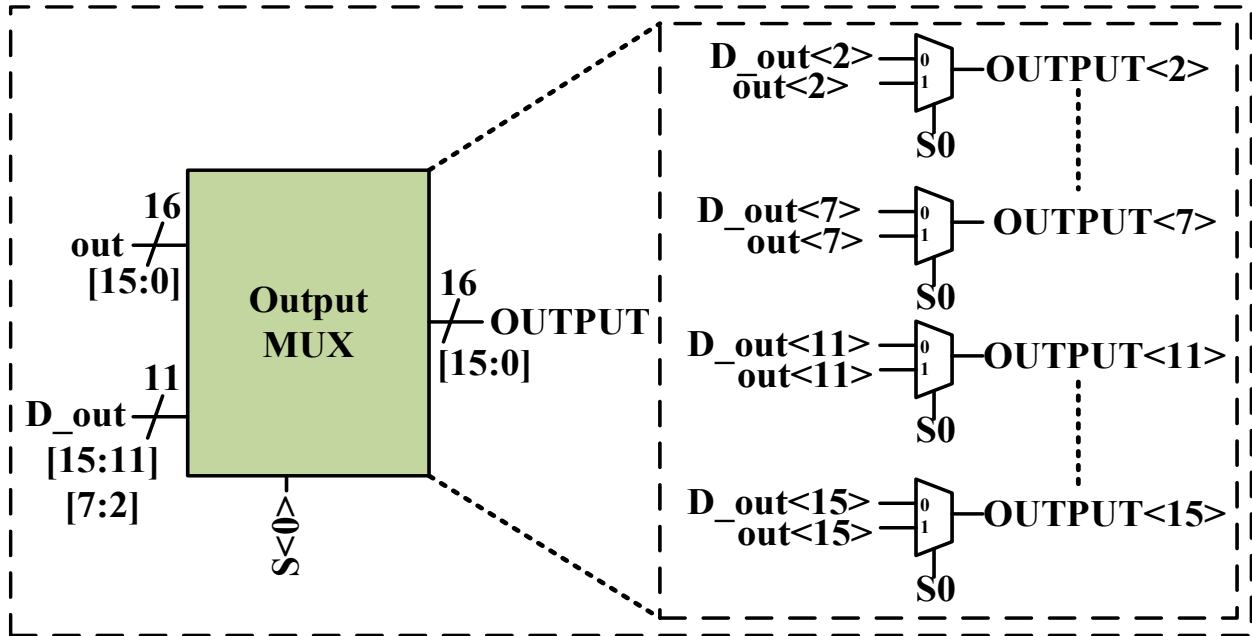


Figure 26. Output MUX.

Figure 26 shows the output MUX, which selects between the potentially ECC corrected bits and the original SRAM bits for bit positions 15-11 and 7-2, depending on whether ECC is selected or not. Bit positions 10-8 and 1-0 are always the original SRAM bits, as mentioned above, so no MUX is required for these.

3.6. Results

3.6.1. Timing Diagram

Figure 27 presents the timing diagram for the proposed memory, showing three segments of simulation waveforms, No ECC (i.e., normal memory operation), ECC74, and ECC1511.

Specifically, Figure 27 shows (a) the input data, (b) the data after encoding, (c) the error correction bit information, and (d) the data after decoding.

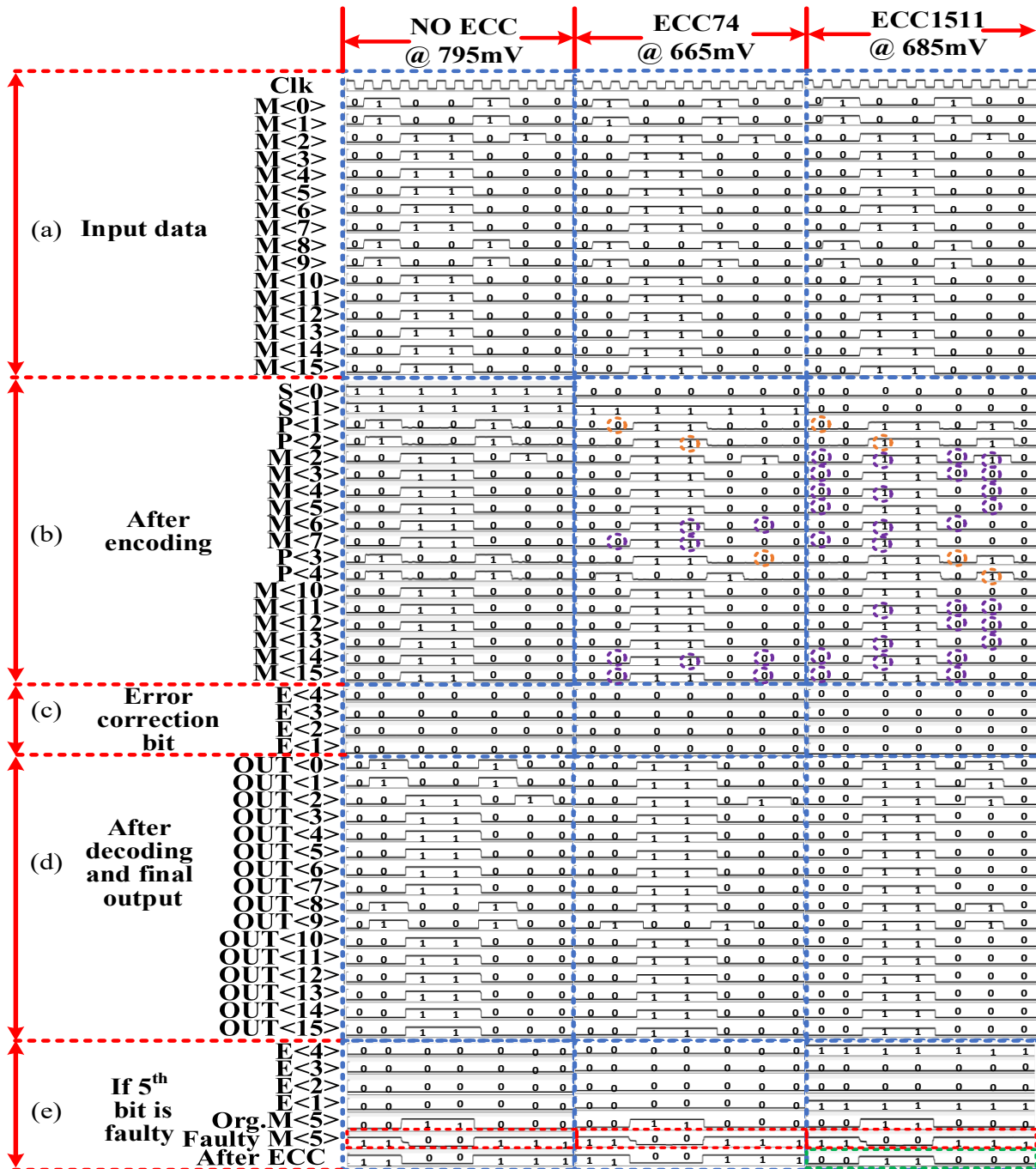


Figure 27. Timing diagram: purple marked bits are the combination that generated the orange marked parity bits for each operation.

At first, the input data is applied to the memory and after the ECC encoder, the generated parity bits are sent to the memory block to be stored as the LSBs. As shown in Figure 27 (b), the purple marked bits are the combination that generated parity bits (orange marked) for that specific operation. To calculate parity for ECC1511 and ECC74, seven and three input bits are needed, respectively. Finally, the memory checked the error correction bits E[4:1] in Figure 27 (c), and if all zeros, then no error is detected. Otherwise, the correction unit toggled the faulty bit. Since Figure 27 (c) is always all zeros, there are no faulty bits, such that Figure 27 (b) and (d) are the same.

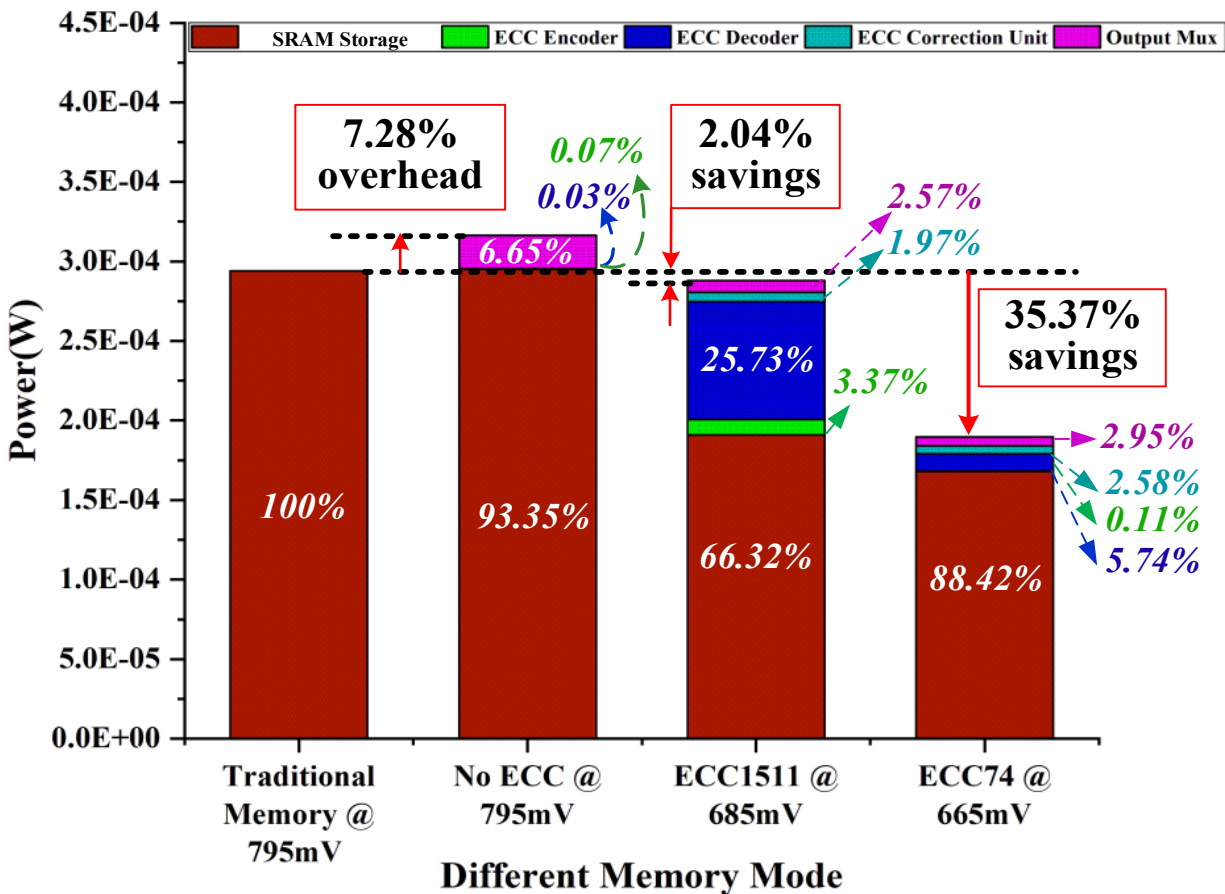


Figure 28. Power Comparison.

Figure 27 (e) illustrates the case when one faulty bit is stored in place of the original message bit, M5. Since neither No ECC nor ECC74 protects M5, their error correction E[4:1] is

all zeros, and their M5 output after ECC is the same as the inserted faulty M5 bit. However, for ECC 1511, $E[4:1] = 10012 = 910$, which corresponds to M5 according to Table 7. Hence, the faulty M5 is flipped which resulted in M5 after ECC being the same as the original M5.

3.6.2. Power Efficiency

For power analysis, the same 45nm CMOS process discussed in Section 3.4 was utilized. For each testcase shown in Figure 28, the average power consumption is measured for writing FF0016 to a random word in a $128 \text{ word} \times 16 \text{ bit}$ memory bank, which is initialized to A5A516, followed immediately by reading FF0016 from the same word, such that all read/write memory operations are equally included (i.e., reading '0' and '1', and writing '0' to '0', '0' to '1', '1' to '0', and '1' to '1'). The baseline design is the traditional SRAM without any of the additional ECC circuitry, operating at 795mV, the lowest voltage that do not induce errors, which requires an average power of $2.94\text{E-}4\text{W}$. The first testcase is the proposed ECC memory operating at 795mV, where No ECC is selected, which shows that the added ECC circuitry only required an additional 7.28% power when not being utilized. The next testcase is the proposed ECC memory operating at 685mV, the lowest voltage where ECC1511 is invoked, which consumed 2.04% less power than the baseline design. The final testcase is the proposed ECC memory operating at 665mV, the lowest voltage where the failure rate would still be less than 1%. In this case, ECC74 is invoked, which resulted in a 35.37% power reduction compared to the baseline design. The tradeoff for this reduction in power is slightly decreased video quality (i.e., PSNR of 34.32dB for the ECC1511 case and 32.28dB for the ECC74 case, both of which are well above the minimally acceptable 30dB).

3.6.3. Video Quality

To evaluate the quality of videos with the proposed method, 1,000 videos with diverse characteristics were selected randomly from YouTube-8M [14], with the simulation results shown in Figure 29. If no ECC is applied with 0.1% error, PSNR ranged between 32dB to 33dB. With ECC1511 enabled, PSNR improved by approximately 23.31%. If no ECC is utilized with 0.9% error, PSNR ranged between 22dB to 23dB, and improved by approximately 32.98% with ECC1511 enabled. Furthermore, applying ECC74 to the 0.9% error case even further increases PSNR improvement, as Section 3.4.3 determined that ECC74 is better than ECC1511 for higher failure rates. Another observation that can be made from Figure 29 is that at 0.9% failure rate, the PSNR values with ECC1511 are below the acceptable 30dB threshold for quality [15]; however, with ECC74, almost all videos maintain acceptable quality.

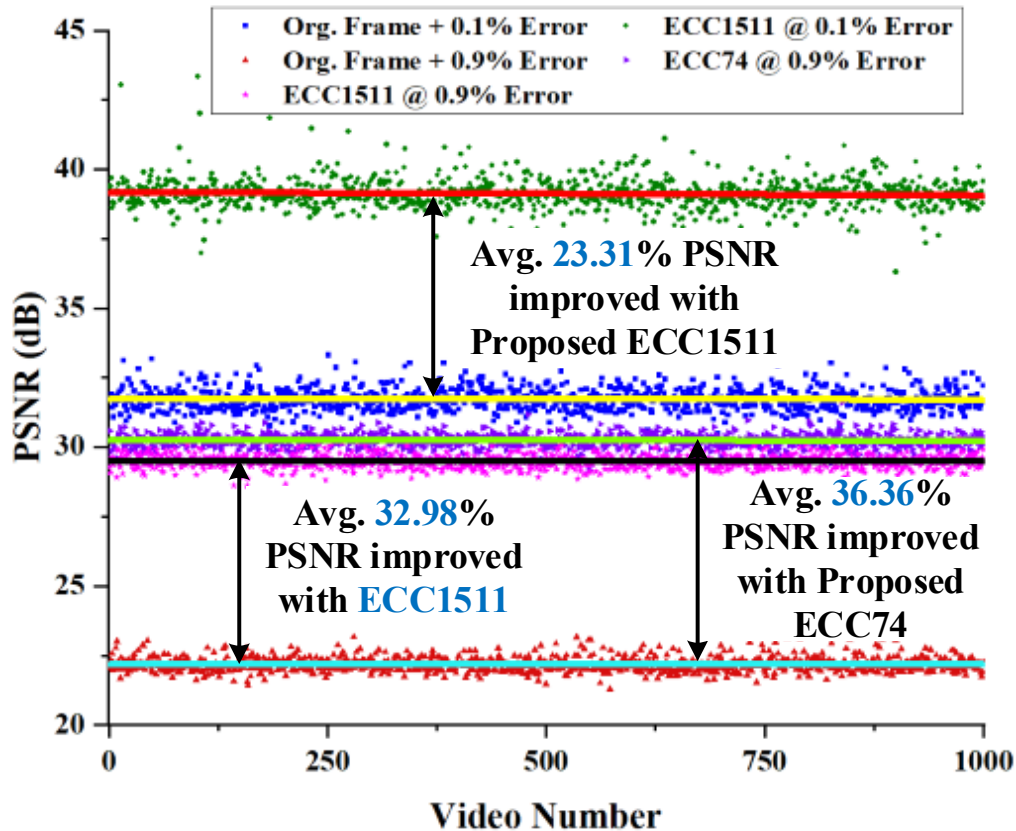


Figure 29. PSNR values of 1,000 videos at 0.1% and 0.9% failure rates.

3.7. Hardware Implementation for Verification

In Sections 3.3 & 3.4, the proposed memory design is presented, simulated, and evaluated. In this section, the same video data is written to a commercial 65nm SRAM chip at various voltages for testing on real hardware, to verify the simulation results.

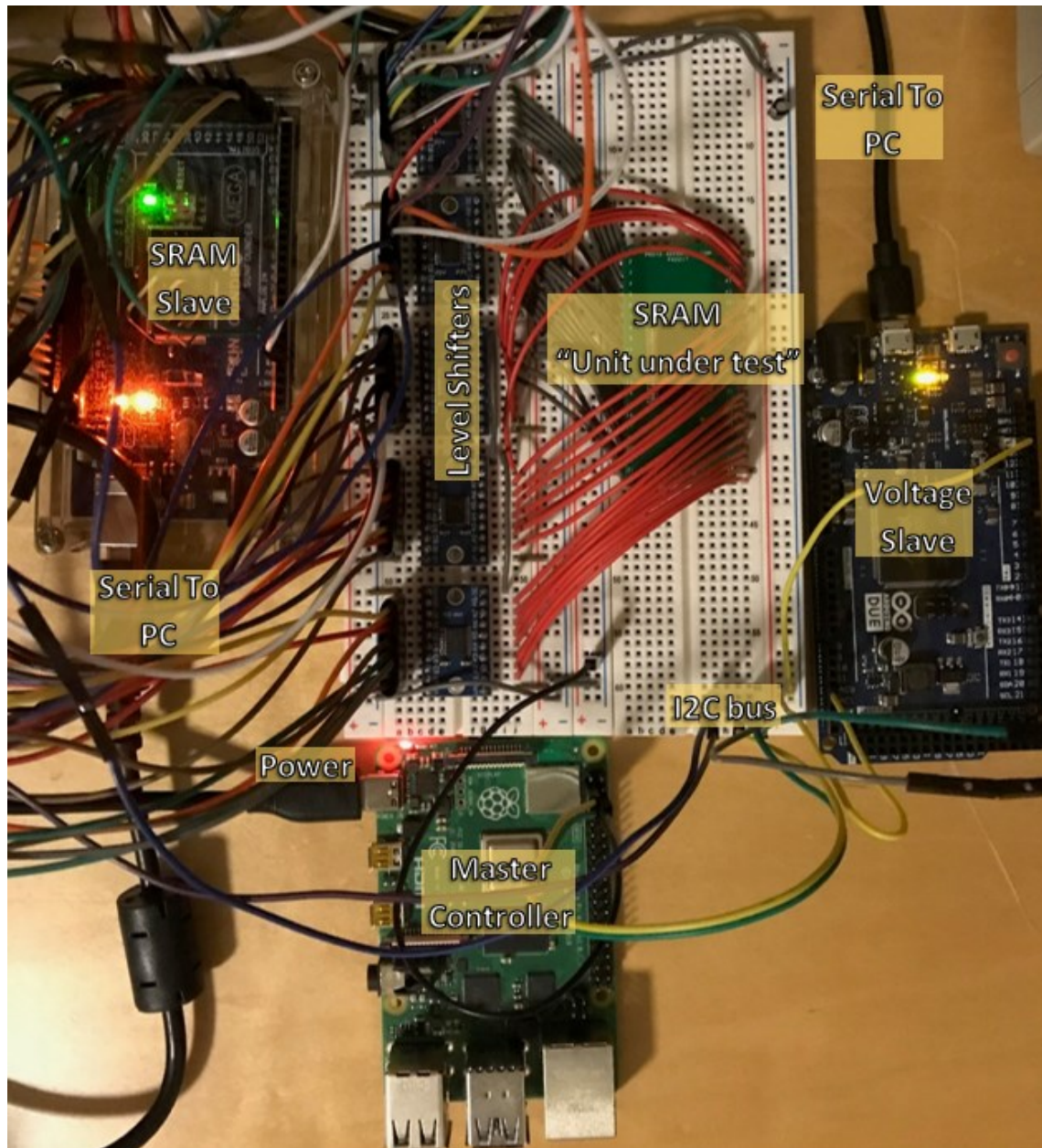


Figure 30. Variable voltage SRAM test platform: (a) Arduino Mega, Arduino Due, Raspberry Pi, Texas Instruments Level Shifters, and Cypress 65nm TSOP-44(II) SRAM based system implementation.

3.7.1. Variable Voltage SRAM Test Platform

The variable voltage SRAM test platform in Figure 30 is implemented as a prototype alternative to taping out a custom IC encompassing the proposed ECC solution. The platform consisted of three microcontroller boards: one master and two slaves, and an SRAM Unit Under Test (UUT). The master controller commanded the two slave controllers, which are used as 1) an SRAM memory controller commanding the SRAM UUT and 2) a variable voltage controller utilizing voltage level shifting ICs. The level shifting ICs are necessary so as to prevent parasitic voltage interference from adversely affecting the results. Using the voltage shifters, all SRAM inputs are voltage controlled at the desired test voltage level. The master controller commanded both the variable voltage controller and the SRAM slave controller via an I2C serial communication interface. The SRAM slave controller translated the master controller's commands into Read and Write operations for the SRAM, thus acting as an SRAM hardware abstraction layer and interface.

The Block diagram in Figure 31 shows the interconnections of the various modules within the system in Figure 30, as well as the various responsibilities of each module. The operation began via user input from a Personal Computer through a Secure Shell terminal to communicate with the master controller to specify which video frame to read/write from/to the SRAM using what specific voltage level. The platform then executes the user input command; and if write is selected, the system first applied one of the 3 ECC schemes (i.e., no ECC, ECC74, or ECC1511), based on the selected voltage level, before writing the data to the SRAM.

3.7.2. SRAM Error Characterization at Given Voltages

The SRAM UUT used is the Cypress CY62146GN 65nm 2.20V CMOS static RAM organized as 256K words by 16 bits [16]. When a bit fails on this SRAM, the failure is a “stuck-

at-one” standard high voltage. The SRAM is characterized to determine the failure rate at each supply voltage, as shown in Figure 32. A test consists of first resetting the chip by writing 1s to the entire addressable region of the SRAM. Then, test data is written as 0s to the entire addressable region of SRAM. After which, the written test data is read back, verified, and analyzed for failures. A test is conducted at the voltage range of 2.20V down to 0.97V, with 0.01V decrements, for a total of 124 tests. The SRAM is switched off before and after each test and erased, so data hysteresis do not affect the results. At the SRAM specification’s recommended voltage of 2.20V, the data read back resulted in a 0% failure rate; on the other hand, starting with voltages less than 1.03V, the data read back resulted in a 100% failure rate. Figure 32 (b) lists the specific failure rates for each voltage between 1.25V and 1.02V, which shows that failure rate increases exponentially as voltage decreases.

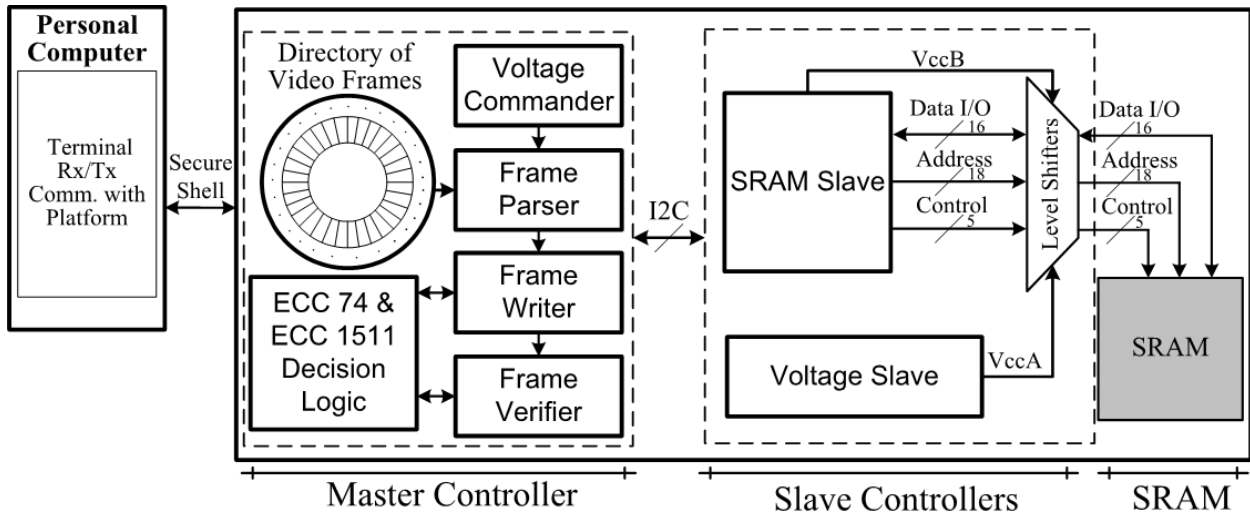


Figure 31. Block diagram of the Master and Slave controller interaction to write, read, and verify data on the SRAM, as well as control its supply voltage.

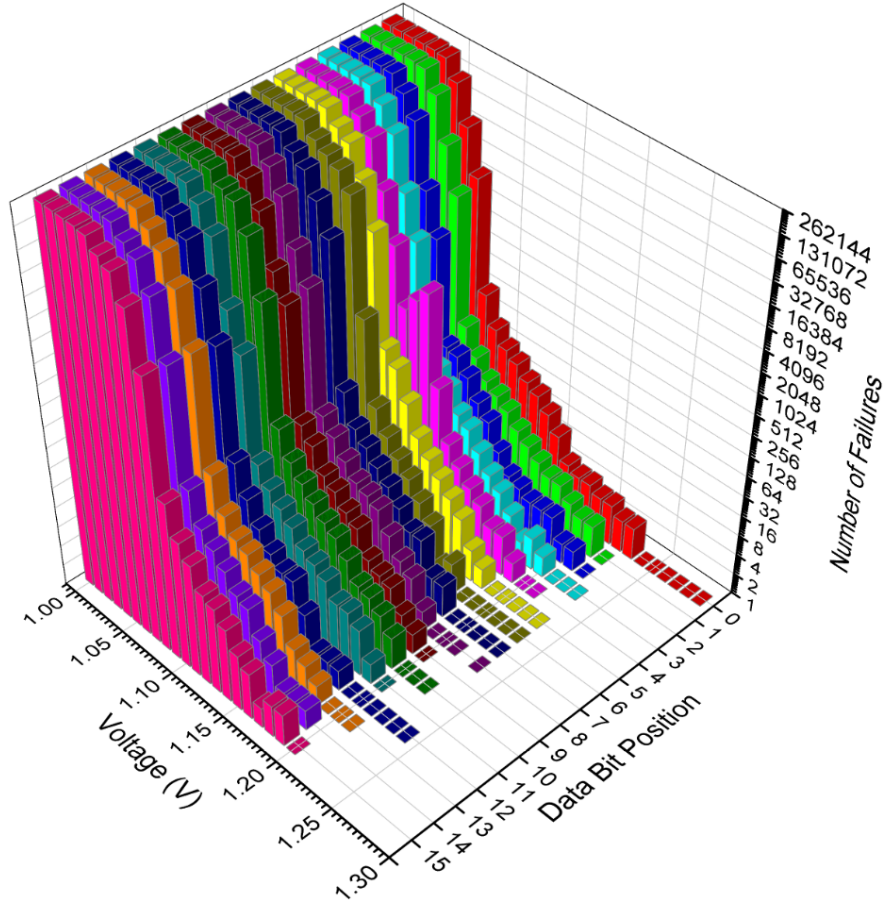
Also, as shown in Figure 32 (a), and better shown in Figure 33 for a specific voltage of 1.06V with a failure rate of 44.25244%, error distribution across the 16-bit data range is observed to be fairly uniform. In Figure 33, all addressable sectors of the SRAM are written and read back, then stored in a 16bit x 256K array in software. Figure 33 (a) provides an

exaggerated visual model of the SRAM; because in reality the image would be far too long and skinny to be accurately displayed in this chapter (i.e., 16 pixels by 262,144 pixels). Figure 33 (b) shows a 128-word sub-section of Figure 33 (a) from address 0x00FFF to 0x0107F, so that the errors in black, where each pixel denotes a bit, can be seen in more granularity. Figure 33 (c) then reassembles the 16 x 256K SRAM from Figure 33 (a) into a more presentable 512 pixel x 512 pixel format, where each pixel corresponds to a single SRAM bit, to demonstrate the uniformity of error distribution.

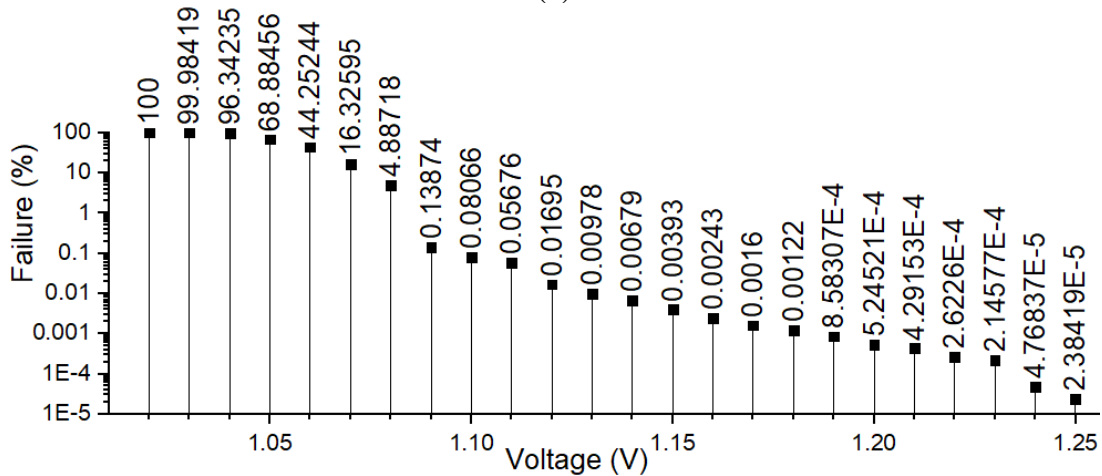
Note that since SRAM failures occur as “stuck-at-one”, failures follow the bit pattern in the data. For example, if data 0xFF are to be written, there would be no failures detected, whereas for data with more zero value bits, more failures would be detected. This is the reason why the failure characterization test data written is all 0s, to detect all failures in the SRAM. From correlating the failure rate data from Figure 32 (b) with the results from Section 3.4.4, the range of voltages where the proposed ECC algorithms would be most beneficial are 1.12V down to 1.09V.

3.7.3. Hardware SRAM Testing

Writing video frames to the SRAM requires the data to be divided into 16-bit chunks to fit into the 16-bit SRAM data width. The frame pixels, however, are 24-bits in width: 8-bits each for Red, Green, and Blue (RGB) vectors. Thus, the pixel data is organized into a sequential flattened array of 16-bit elements to fit into the SRAM. When the frame data is read back from the SRAM, the data is reconstructed into its original frame format. Note that this data reorganization do not affect the proposed method, since the ECC algorithms protect the MSBs of each 2 bytes (16-bits) of data.



(a)



(b)

Figure 32. (a) SRAM failure characterization map for the entire addressable memory region per bit. Voltage ranges 1.30V down to 1.00V are shown, which demonstrate the per bit error distributions from a 0% to 100% failure rate. The granularity of voltage plotted is 0.01V. Voltages 2.20V down to 1.31V were not shown for clarity, as they resulted in a 0% failure rate; (b) failure rate at each voltage between 1.02V and 1.25V. Failure rates not shown on the graph above 1.25V were nearly 0%, above 1.30V were 0%, and below 1.02V were 100%.

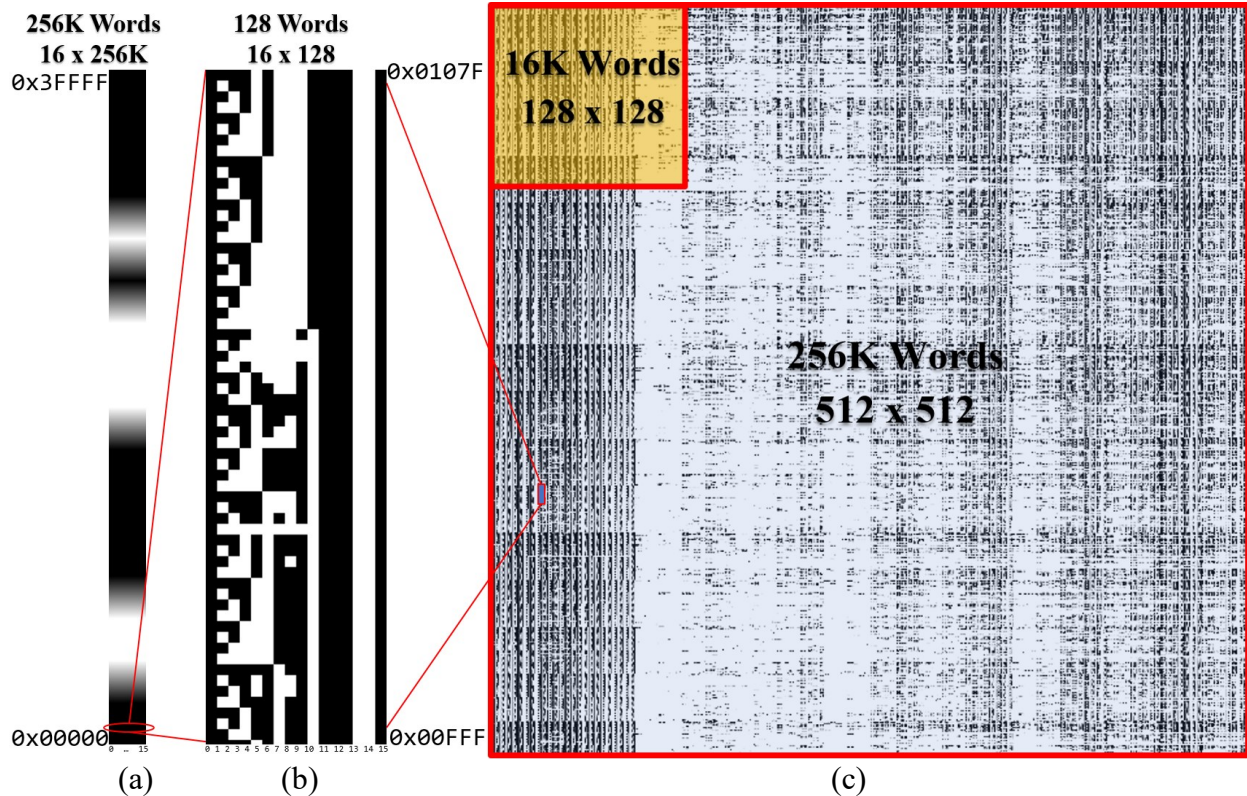


Figure 33. SRAM error distribution map at 1.06V, where failure rate is measured as 44.25244%, to show the uniformity of error distribution (black pixels denote error): (a) exaggerated visualization of 16bit by 256K SRAM memory structure (b) Sub-section of (a) to clearly show the error distribution across 128 words: 16bits by 128 (c) Re-organized and re-assembled from (a).

3.7.4. Hardware SRAM Analysis

Figure 34 details PSNR measurements using No ECC, ECC74, and ECC1511 at supply voltages from 1.13V-1.08V, with 0.01V decrements, which corresponded to failure rates between 0.00978% and 4.88718%, respectively. Samples of the written then read back frames are presented for the reader's observation in Figure 35. Figure 34 shows that for failure percentages below 0.01%, which corresponds to a supply voltage of 1.13V and above, No ECC performs best: as both ECC74 and ECC1511 algorithms decrease quality. For a failure rate of 0.01695%, which corresponds to a supply voltage of 1.12V, ECC74 is the best choice. For a failure rate of 0.05676%, which corresponds to a supply voltage of 1.11V, ECC1511 is the best. And finally, for failure rates 0.08066% and greater, which corresponds to supply voltages of

1.10V and lower, ECC74 is the best choice. The results from these tests correlated similarly to the simulation results from Section 3.4, with the difference being the more limited range where ECC1511 outperforms ECC74. This is because bit errors always manifest as a “stuck-at-1” output for this Cypress SRAM, rather than being a random 0 or 1, such that only 0 bits result in errors, which makes the perceived failure rate less than the actual failure rate.

Figure 35 displays a zoomed view of the decoded video data at 1.11V and 1.09V, each with No ECC, ECC74, and ECC1511 applied, which can be compared to the original frame without error, shown in shown in Figure 16 (a). Each frame is written at the scaled voltage, including parity bits if ECC is enabled, and then read back at the same scaled voltage. For ECC74 and ECC1511, error corrections are computed and reapplied to the frames in a post processing software routine. As shown in Figure 35, the frames with the highest PSNR values demonstrate the best video quality. Specifically, at a voltage of 1.11V, ECC1511 performs the best; alternatively, at a voltage of 1.09V, ECC74 delivers the best quality. It can be observed from Figure 35 that a 1-2 dB PSNR difference enabled by the different ECC schemes can have a significant impact on visual quality. As circled in Figure 35 (b) and (f), there is a clear distortion as compared to (c) and (e), respectively, due to the increased number of errors manifesting themselves as scattered dots in the image. As a result, the proposed memory, with three different power-quality tradeoff levels (i.e., no ECC, ECC74, and ECC1511), yields better image quality compared to only utilizing a single ECC scheme.

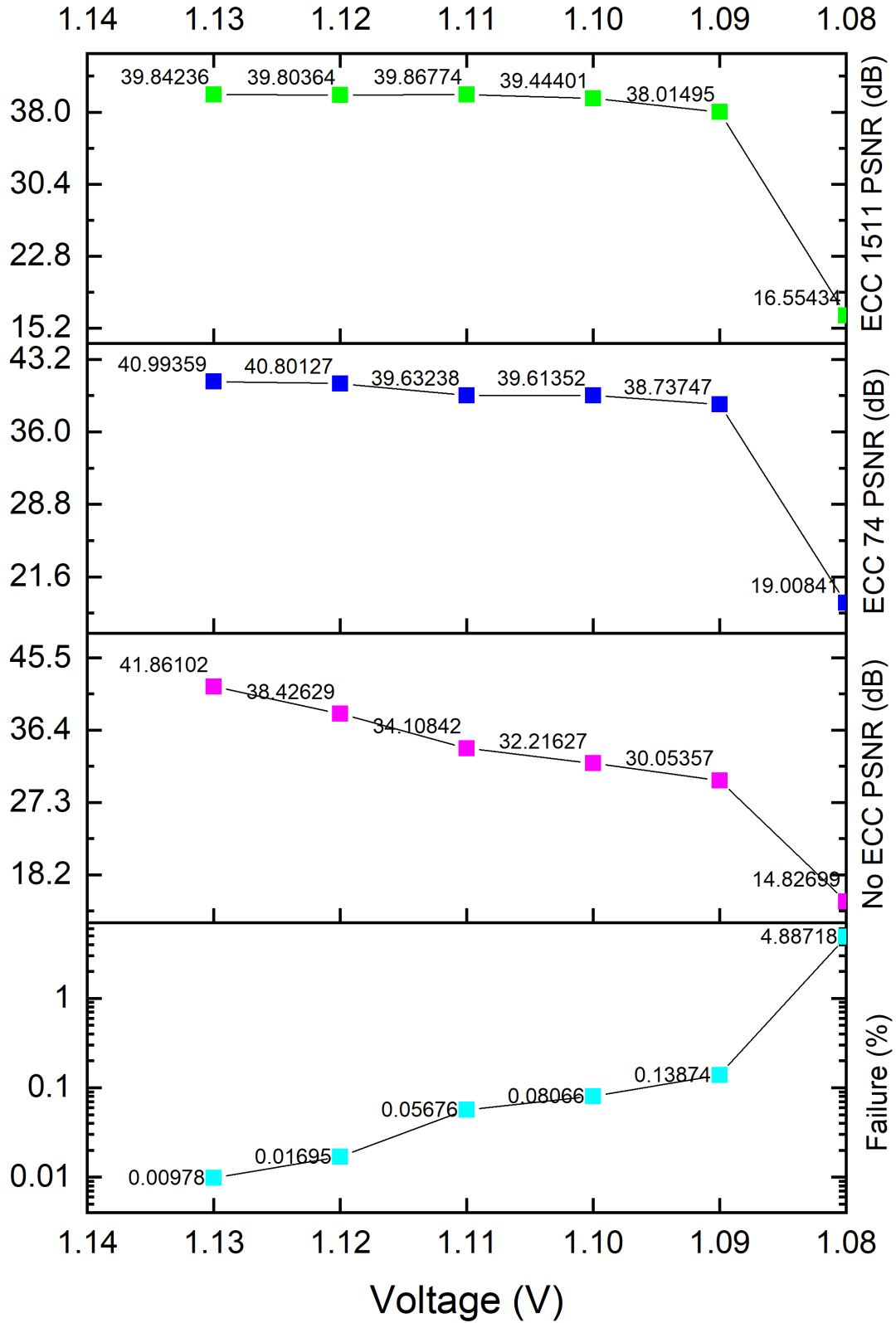
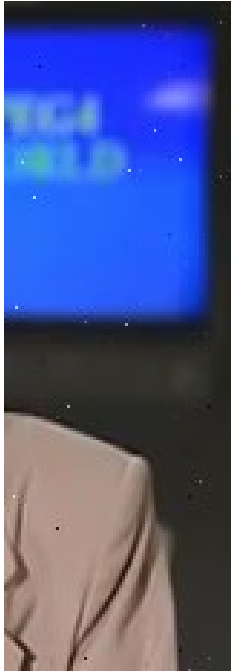
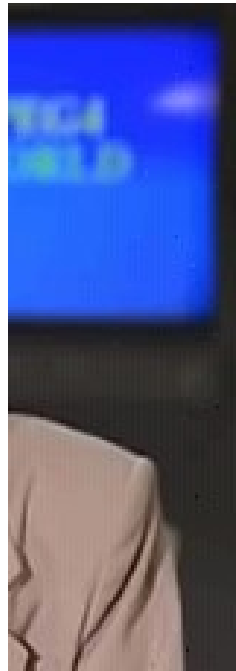


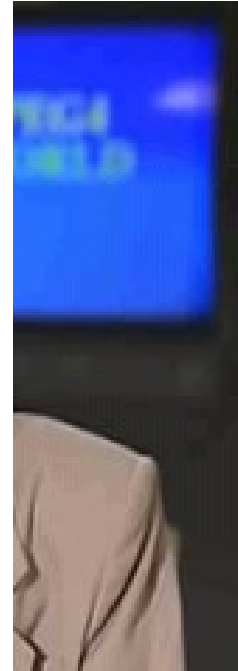
Figure 34. Failure rate and PSNR values at different supply voltages for three memory modes (No ECC, ECC74, and ECC1511).



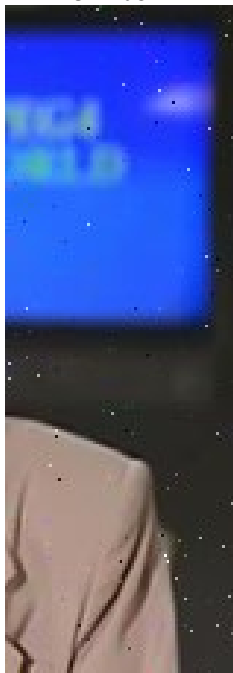
(a) No ECC at 1.11V
PSNR: 34.10842 dB



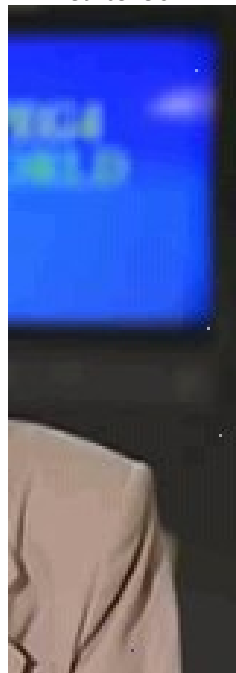
(b) ECC74 at 1.11V
PSNR: 39.63238 dB



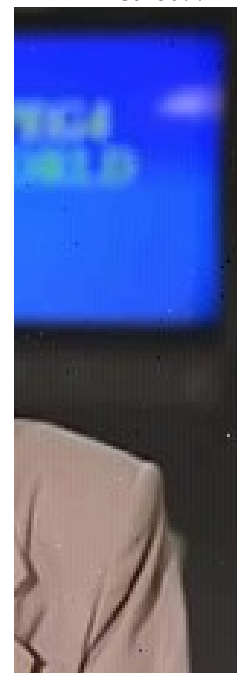
(c) ECC1511 at 1.11V
PSNR: 39.86774 dB



(d) No ECC at 1.09V
PSNR: 30.05357 dB



(e) ECC74 at 1.09V
PSNR: 38.73747 dB



(f) ECC1511 at 1.09V
PSNR: 38.01495 dB

Figure 35. Zoomed in video frame using the proposed memory with different modes (No ECC, ECC74, and ECC1511) at two different supply voltages. The circled regions highlight visual degradation due to slightly increased number of errors.

3.8. Comparison with Prior Work

Table 8 compares this work against state-of-the art video memory designs. As shown, the proposed memory enables run-time quality adaptation without inducing bitcell area overhead, and without requiring multiple supply voltages.

Table 8. Comparison with Prior Work

	Priority SRAM [5]	Heter-sizing SRAM [6]	Split-data-aware SRAM [7]	Content-adaptive SRAM [3]	Data-dependent SRAM [12]	SRAM with traditional ECC [17, 18]	SRAM with hamming (15,11) [11]	Proposed SRAM with ECC Adaptation
PSNR run-time adaptation	No	No	No	Yes	Yes	No	Yes	Yes
Bitcell area overhead (compared to basic 6T)	Yes (6T and 8T)	Yes (Larger 6T)	Yes (8T and 10T)	No	Yes (10T)	No	No	No
Encoder-side modification	No	No	No	Yes	No	No	No	No
Additional column needed	No	No	No	No	No	Yes	No	No
ECC Adaptation	-	-	-	-	-	No	Yes (No ECC and ECC1511)	Yes (No ECC, ECC1511, ECC74)

3.8.1. Compared to State-of-the-Art Approximate Video Memories

The priority-based 6T/8T SRAM [5], heterogeneous sizing SRAM [6], and split-data-aware SRAM [7] were developed to store MSBs in more robust more-than-6T SRAM bitcells (such as 8T, 10T, or upsized 6T) and LSBs in error-prone but area-efficient basic 6T bitcells, thereby leading to a tolerable output quality degradation with power reduction. However, the video quality enabled by those designs is fixed during design-time, so they cannot adapt at run-time to meet requirements of different video applications.

3.8.2. Compared to State-of-the-Art Adaptive SRAM

Recently, two video SRAM designs with run-time adaptation have been presented, data-dependent memory [12] and content-aware memory [3]. The data-dependent SRAM consists of

10T bitcells and associated conditional pre-charge circuitry to adapt to the statistical dependencies present in the binary values. The content-aware SRAM adapts the number of truncated LSBs of video data based on the average plain macroblock percentage of an entire video sample; therefore, a macroblock characterization process is needed in the video encoder side [3]. In the proposed memory, the run-time quality adaptation is enabled without inducing bitcell area overhead or encoder-side modification.

3.8.3. Compared to State-of-the-Art SRAM with traditional ECC

SRAM designs with different ECCs have been developed to implement low-power memories [17, 18]. However, in SRAM with traditional ECCs, either memory capacity needs to be increased or part of the memory's effective capacity has to be sacrificed, in order to store parity bits. As discussed in Section 3.3.1, a 75% and 36% silicon area overhead are introduced by using traditional ECC74 and ECC1511, respectively. Similarly, by using orthogonal Latin square codes discussed in [18], half of the memory capacity is used to store parity bits. In addition to memory space overhead, additional logic for ECC encoding and decoding must be added, which causes significant implementation penalty. The proposed memory design intelligently uses LSBs to store the parity bits, and develops a reusable decoder and encoder to support both ECC74 and ECC1511, thereby avoiding significant area overhead.

3.8.4. Compared to State-of-the-Art Memory with Selective ECC

The SRAM with selective Hamming (15,11) presented in [11] is another recent adaptive memory design that switches between no ECC and ECC1511 based on the quality targets of the applications. It also utilizes LSBs to store parity bits to save silicon area. Specifically, [11] protects 2.75 MSBs per byte using 1 LSB per byte, while the proposed ECC74 protects 2 MSBs per byte using 1.5 LSBs per byte, and the proposed ECC1511 protects 5.5 MSBs per byte using 2

LSBs per byte – neither require any memory storage area overhead. The video output quality of two memory designs at different failure rates using 101 videos is further compared, including the video Akiyo and 1000 randomly selected YouTube-8M videos. The average PSNR values are listed in Table 9, showing that the proposed technique yields higher PSNR values in most cases, compared to [11], with the same zero area storage overhead. Finally, it should be emphasized that the developed negative bitline scheme in [11] could also be added to the proposed technique to further increase power savings.

Table 9. Comparison with [11]

Failure Rate	Average PSNR Values of 101 Videos		Best Scheme
	[11]	Proposed	
0%	Infinite	Infinite	Both with No ECC
0.1%	46.236 dB	43.774 dB	[11] with ECC 1511
0.3%	39.702 dB	39.951 dB	Proposed with ECC 1511
0.5%	36.003 dB	37.065 dB	Proposed with ECC1511
0.7%	33.357 dB	35.064 dB	Proposed with ECC74
0.9%	31.428 dB	33.518 dB	Proposed with ECC74

3.8.5. Comparison Summary

In the developed adaptive memory technique, a new parity storage scheme to support three power-quality tradeoff levels is proposed, hamming code-74 (ECC74), hamming code-1511 (ECC1511), and no ECC. It utilizes LSBs to store the parity bits, thereby avoiding dedicated parity bit storage. Additionally, with ECC protection, it does not need to adopt upsized more-than-6T bitcells to store MSBs. Accordingly, the proposed memory enables run-time quality adaptation with significantly reduced overhead and better video quality, as compared to existing techniques.

CHAPTER 4. CONTENT-ADAPTABLE ROI-AWARE VIDEO STORAGE FOR POWER-QUALITY SCALABLE MOBILE STREAMING³

The demand for mobile video streams is constantly increasing. With this demand comes a need for mobile devices to receive more videos at ever increasing quality. However, due to the large size of video data and intensive computational requirements, video streaming requires frequent memory access that consume a substantial amount of mobile device power; as a result, the battery life of mobile devices is limited. In this chapter, a video content-adaptable Region-of-Interest (ROI)-aware video storage technique is presented that promotes power savings. During the video encoding process on the transmitting server, based on the macroblock variance and ROI characterization, the “macroblocks of interest” are identified and embedded in the encoded bitstream. In the decoding process, a new frame buffer with dynamic power-quality trade-off is presented to adapt to the macroblock characteristics during run-time. Results from the system-level and circuit-level simulations show that the proposed technique enables substantially more truncated bits and significant power savings while delivering similar or better video quality as compared to other state-of-the-art solutions.

4.1. Introduction

Mobile video streaming on YouTube, Vimeo, and Netflix has increased on average 70% per year and will consume approximately 79% of the total internet traffic by 2022 [1]. At the

³ The material in this chapter was authored by Ali Ahmad Haidous, William Oswald, Hritom Das, and Na Gong. Ali Ahmad Haidous was in charge of the theoretical development, design, implementation of the proposed techniques, system level validation, feasibility, experimental methodologies, and experimental results. He invented the system-level novelties and solutions, lead the research direction, created experiments, and developed the hardware system test platform. He was also in charge of all simulations and results completed using python while assisting and mentoring William Oswald on other python code simulations, statistical analysis, and video frame analysis. The source code is available in the appendix. William Oswald was responsible for the statistical analysis, power analysis, and video frame analysis. Hritom Das was in charge of all circuit design in Cadence, power analysis, and simulation of the proposed frame buffer memory. Na Gong contributed to the theoretical development, memory design, video quality metrics, and data analysis and interpretation.

same time, power-efficient video storage has proven to be a very challenging problem to solve. This is due to the large data sizes associated and intensive computational requirements demanding frequent data access. With the advancement of computing technologies, more video streaming services deliver content to battery-powered mobile devices: such as smart phones and Internet-of-Things (IoT). On one hand, these devices would benefit greatly from low-power consumption as this would extend their battery life. On the other hand, the mobile video streaming process – receive, decode, and display of a video bitstream – consumes considerable power and limits the mobile devices’ battery life. For example, with a video decoding chip, embedded memories contribute to over 50% of the decoding power consumption [2]. This use-case is only expected to grow for the next-generation video formats, H.265/HEVC and H.266/VVC, which has 2x-3x greater memory demands when compared to H.264 [3].

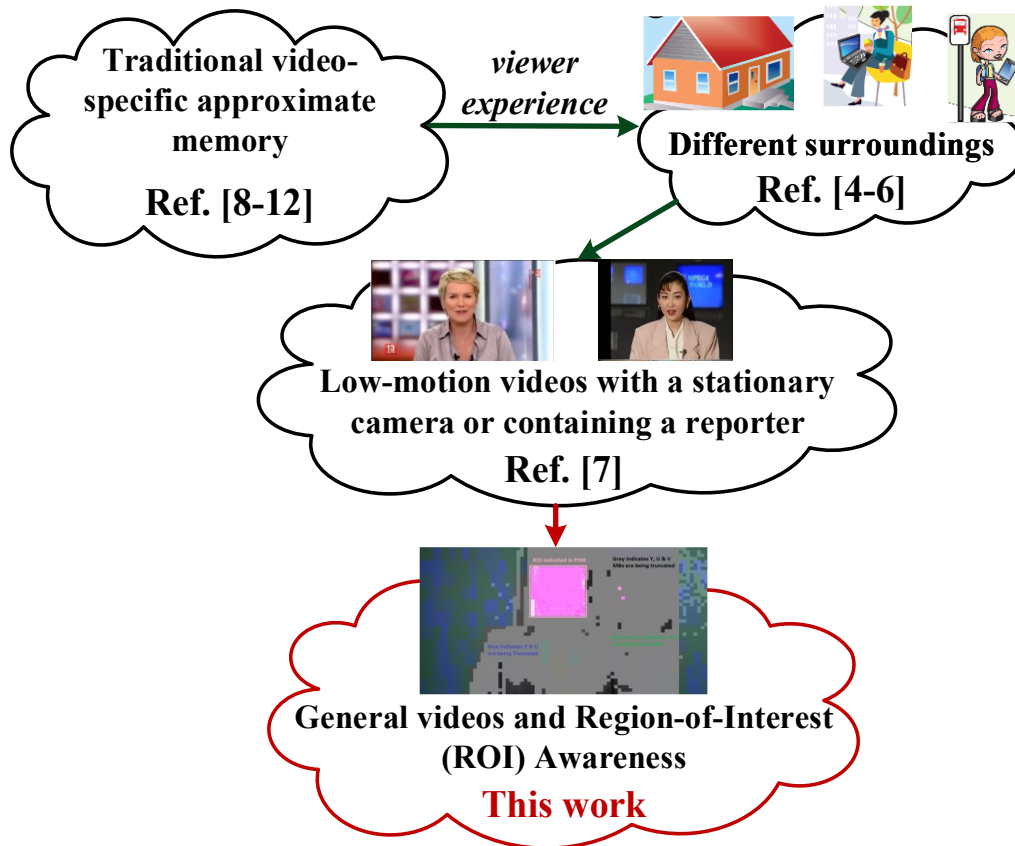


Figure 36. Proposed content-adaptable ROI-aware low-power video memory.

Today's mobile hardware designers, including memory designers, are focusing on hardware-level energy-efficient design techniques in order to accommodate the large amount of video data. However, these design techniques usually come with significant implementation overhead (e.g., silicon area, delay) to solve failure problems in memories. Viewer-aware video memory design was recently explored by investigating the impact of illuminance levels in different viewing surroundings on the viewer's experience [4, 5, 6, 7], as illustrated in Figure 36. The previous studies illustrate a new dimension of power savings for hardware design through the introduction of viewer awareness, but the developed memories lack runtime adaptation across a wide variety of mobile videos. To enable an optimized trade-off between power efficiency and video quality, -, this chapter aims to develop a video content-adaptable Region-of-Interest (ROI)-aware memory for general videos. Specifically, this chapter makes the following contributions:

- An intelligent ROI-aware and content-adaptive framework is proposed to determine video frame regions to preserve (output quality) or truncate bits for power savings. The truncation is applied for all Luma and Chroma video data (i.e., Y, U, and/or V components) (Section 4.3).
- The system-level implementation scheme of the proposed technique is developed and discussed (Sections 4.4.1, 4.4.2, and 4.4.3).
- A low-power low-cost frame buffer with dynamic power-quality trade-off is developed to adapt to the video content (i.e., macroblock characteristics) during run-time (Section 4.4.4).
- A comprehensive suite of simulations on the proposed technique is performed and the enriched results are discussed, including the performance, circuit-level power efficiency,

video-level power efficiency, number of truncated bits, and output quality of various mobile videos (Sections 4.6.1, 4.6.2, 4.6.3, and 4.6.4).

- An extensive statistical analysis demonstrates the effectiveness of the proposed technique in achieving significant bit truncations and power savings as compared to the state-of-the-art, particularly for the videos with medium or high variance (Section 4.6.5).

To the best of the authors' knowledge, this is the first work that seamlessly integrates ROI knowledge, i.e., "macroblocks of interest", into the hardware design process.

The organization of the chapter is as follows. A review of low-power video memory designs is provided in Section 4.2, Section 4.3 presents the macroblock variance and ROI study, and Section 4.4 discusses the proposed technique. The evaluation methodology and results were discussed in Sections 4.5 and 4.6 respectively, and finally, this chapter is concluded in Chapter 5.

4.2. State of the Art

A vast amount of research has been conducted to improve the power efficiency of video data storage. State-of-the-art, power-efficient video memories consist of either approximate memory with application-level information [8, 9, 10, 11, 12] or viewer-aware memories with an awareness of viewer's experience [4, 5, 6, 7]. In this section, some of the existing work related to the proposed technique are briefly reviewed.

4.2.1. Approximate Video-Specific Memory

Researchers have presented various low-power video memory design techniques. Chang et al. [8] presented a hybrid 6T+8T SRAM to achieve quality-power optimization. Gong et al. [9] developed a hybrid 8T+10T memory for power savings based on the correlation between most-significant-bits (MSBs) of video data. In [10], a heterogeneous sizing scheme was presented to reduce the failure probability of conventional 6T bitcells. The video memory presented in [11]

used the Least-Significant-Bits (LSBs) of video data to store the MSBs' error-correction-code (ECC). Kazimirsky et al. [12] developed a hybrid SRAM+DRAM memory to store MSBs in robust SRAM bitcells and LSBs in error-prone DRAM bitcells, leading to a tolerable output quality with power reduction. However, all of those video memory designs were developed based on an objective video output metric such as the peak signal-to-noise ratio (PSNR): without dynamic power-quality adaptation to viewer's true experience

4.2.2. Viewer-Aware Video Memory

Viewer-aware low-power video memory techniques were investigated in [4, 5, 6]: where an increased amount of ambient luminance allows for a larger number of bits to be truncated without noticeable degradation to the viewers. Very recently, the impact of video content characteristics on viewer's experience were studied to enable video content-adaptive memory with dynamic energy-quality tradeoff [7]. However, the technique determined the number of truncated LSBs based on the averaged plain macroblock percentage of an entire video sample; therefore, it was only effective to store low-motion videos with a stationary camera or containing a reporter in a video cast use-case. Additionally, this technique may result in noticeable distortion, e.g., a banding distortion caused by bit truncation, which negatively influenced the viewer's experience.

The common feature of these viewer-aware storage techniques is that the same number of the truncated bits were applied on an entire video. In contrast, the technique proposed in this chapter realizes content adaptation and ROI awareness within each video frame, thereby maximizing the number of truncated bits while maintaining the video quality.

4.3. Overview of the Proposed Technique

This section presents the motivation of the proposed technique that introduces ROI awareness as bit truncation is applied for power savings. Then, the high-level overview of the proposed technique is shown.

4.3.1. Motivational Example

Researchers conducted studies on the human visual system's (HVS) performance and concluded that viewers usually pay more attention to one or a few areas of a video and the region of concentration is called Region-Of-Interest (ROI) [13]. For example, in video conferencing applications, viewers typically pay more attention to the face regions than other areas. In video surveillance, the facial regions are what viewers concentrate most on in consecutive frames. Accordingly, ROIs have higher contribution towards the overall visual quality than other areas. Consequently, if truncation-caused banding distortion appears in ROIs, this will negatively influence a viewer's experience. Figure 37 shows one example. The output quality of the video (Video tag: wF6lvdXXwc4 [14]) using the technique in [7] is shown in Figure 37 (a). Since the banding distortion caused by bit truncation appears on the reporter's face, viewers were less likely to accept the displayed degradation due to this particularly noticeable distortion, as emphasized in [7].



(a) Output quality using [7] (at 3 truncated bits)



(b) Output quality of the proposed technique (at 3 truncated bits)



(c) [7] (left) vs. Proposed technique (right)

Figure 37. Observer discernable flaws in the facial region due to a “banding effect” on the face when comparing (a) and (b) caused the overall quality of the frame to become unacceptable at 3 truncated bits (Video tag: wF6lvdXXwc4 from [14]).

Therefore, the motivation for this work arises from the following two observations:

4.3.1.1. Protected ROI

In a video frame, the distortion in ROIs is more noticeable by viewers. Accordingly, if ROIs can be extracted and protected from truncation, the video quality would be improved from the viewer's perspective (Figure 37(b)). A comparison of the report's face using the technique in [7] and the proposed technique with ROI awareness is shown in Figure 37 (c).

4.3.1.2. Power savings vs. Bits Truncated

There existed a positive correlation between power savings and the number of bits truncated in a video decoder's frame buffer memory [7]. To optimize the power efficiency, it would be beneficial to increase the number of truncated bits in other regions which are not ROIs: the truncation regions.

4.3.2. Overview of the Proposed Content-Adaptable ROI-Aware Video Storage

Figure 38 shows the proposed content-adaptable ROI-aware video storage technique. During the traditional mobile video streaming process, first, from (1) in Figure 38, the mobile device requests a video for display from the cloud. Then, the streaming servers process the requested video by encoding and transmitting the encoded bitstream to the mobile device for decoding and display, (2) in Figure 38. During this process, multiple memories are needed for storing the intermediate and final results of the frame data. In particular, the reference macroblock, frame memory, and display memory, which store the decoded video frames, are accessed very frequently, and they have a profound impact on the system's overall cost and power consumption. The proposed technique extracts ROIs in the cloud server and transmits the truncation region data together with the encoded bitstream to the mobile device, (3) in Figure 38, to further reduce the mobile device's power consumption from computational overhead. The

mobile device hardware video decoder receives the truncation region data and makes memory bit-truncation decisions for greater power savings with less perceived quality loss than [7]. To optimize the truncation decision logic of the mobile device hardware, which further improves power consumption, either no truncation or 3-bit truncation is applied to the truncation regions. Explicitly, the proposed technique is detailed as follows.

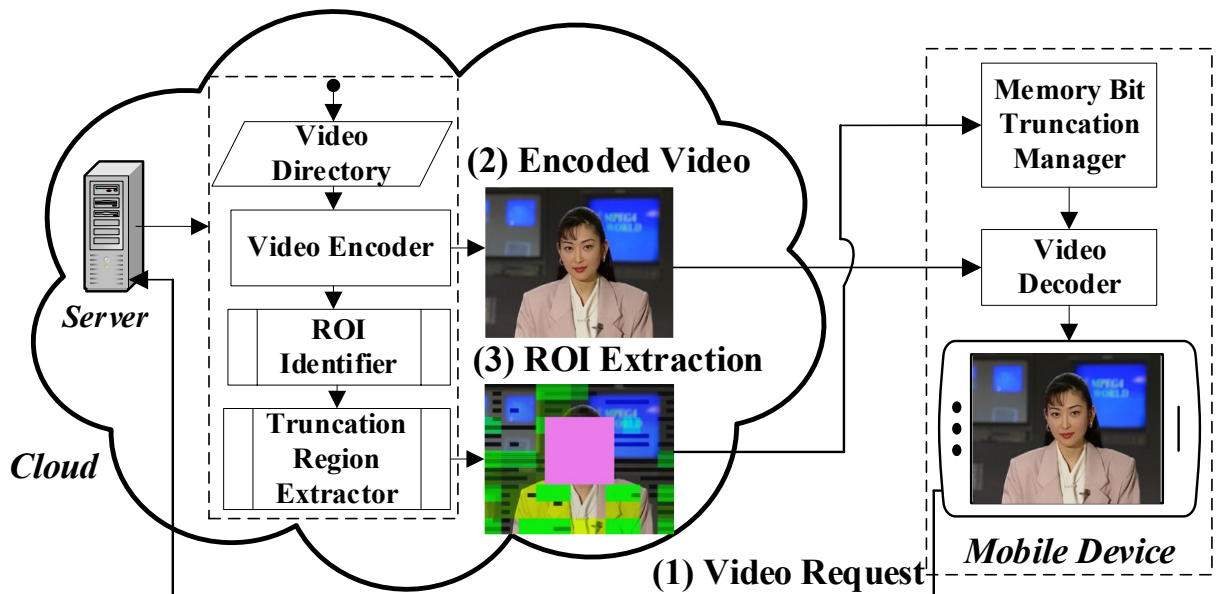


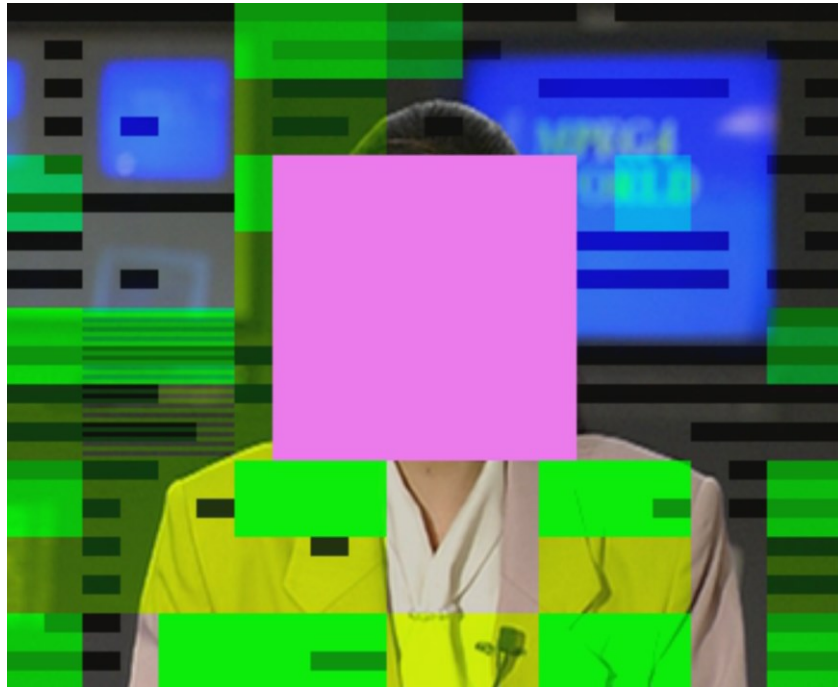
Figure 38. Proposed Region-Of-Interest and macroblock texture framework.

4.3.2.1. ROI Awareness

ROI has been recently applied for different research areas for video system optimization, such as wireless transmission [15], virtual reality (VR) [16], and video summarization [17]. The proposed technique introduces ROI awareness into video storage. Specifically, to minimize the complexity and computational overhead, faces as ROIs in are the focus of analysis which was based on the basic machine learning facial detection OpenCV model [18]. Different algorithms, such as user attention model [13], motion-based models [17], and machine learning models [19], can be applied in future investigations to extract different ROIs, as discussed in Section 4.7.



(a) Original Akiyo Frame (for reference)



(b) Visualized ROI Sample

Figure 39. Akiyo from [28], sample visualized, as generated internal to the proposed method's frame parsing process. Pink, preserved ROI. Seven possible truncation combinations: 1. Green, Y vector truncation. 2. Blue, U vector truncation. 3. Yellow, V vector truncation. 4. Dark blue, YU vectors truncation. 5. Dark Yellow, UV vector truncation. 6. Dark green, YV vectors truncation. 7. Grey, YUV vectors truncation.

4.3.2.2. Video Content Adaptation

After the ROIs to preserve are detected and captured by the framework ROI Identifier, it then searches for regions of low variance measured by the percentage of plain macroblocks (MBs). Specifically, a MB defines an area of 16x16 pixels within a frame. An attribute associated with MBs is how “Textured or Plain” they are. A Plain MB is one in which the variance of intensity within the MB is less than or equal to the threshold value. It has been concluded in [7] that textured MBs are less susceptible to bit-truncation. The pre-established method from [20] is adopted for determining the variance in a MB.

$$V_{MB} = \sum_{i=0}^{15} \sum_{j=0}^{15} (P(i, j) - \rho_{MB})^2 \gg 8 \quad (\text{Equation 14})$$

$$MB = \begin{cases} \text{Plain, if } (V_{MB} \leq Th_{low}) \\ \text{Textured, Else} \end{cases} \quad (\text{Equation 15})$$

Equation 14 and Equation 15, where ρ_{MB} is the average brightness within the MB, V_{MB} is the texture variance within the MB, and traditionally, Th_{low} is defined as a value of 1.25 [21].

4.3.2.3. Truncation Region Extractor

After ROIs are identified on the server, a truncation region extractor encodes the truncation region data using a proprietary protocol per frame and transmits in synchronization with the encoded video transmission to the mobile device. The truncation region data is decoded onboard the mobile device’s hardware video decoder in a novel Memory Bit Truncation Manager (MBTM) hardware unit: which truncates a novel frame buffer memory through the use of unique control YUV truncation signals. The video decoding and bit truncation processes occur in lockstep.

4.3.2.4. 3-Bit Truncation

Truncation is performed in the YUV (Y'CbCr) color space [22], inferring that any truncation is done to the YUV color values. The memory designed in [7] truncated 1, 2, or 3 bits in the Least Significant Bits (LSBs) of the Y vector of all frames within an entire video as a blanket truncation. The proposed technique will enable a different amount of truncated bits for each region within each frame within an entire video. To minimize the implementation overhead, only 3-bit truncation is adopted in the new frame buffer, which will be discussed in Section 4.4.4. Meanwhile, the proposed technique can identify bit-truncation for each Y, U, and V vector of the frame separately for each truncation region in each frame, instead of only truncating the Y vector as a blanket truncation across the entire video as the existing techniques [4, 5, 7]. Furthermore, the proposed technique is expected to enable additional bit truncations as compared to existing techniques. Also, to minimize the video quality degradation caused by bit truncation, the developed frame buffer truncates three LSBs to the optimal value "100" [7], instead of truncating the values to "000".

Figure 39 shows the Akiyo video sample using the proposed technique. The extracted preserved ROI region is highlighted in pink. All truncation regions within a frame are identified, including the following seven possible truncation combinations: (1) Green, Y vector truncation; (2) Blue, U vector truncation; (3) Yellow, V vector truncation; (4) Dark blue, YU vectors truncation; (5) Dark Yellow, UV vector truncation; (6) Dark green, YV vectors truncation; and (7) Grey, YUV vectors truncation. Each of these combinations would be encoded in the truncation region data for the MBTM to generate control signals for memory bit truncation in the video decoding process.

To conclude, the proposed technique truncates the chroma sub samples within each frame as well as the luminosity: Y, U, and V vectors. Previous research only targeted luminosity, Y, of a video for truncation, while chroma samples were disregarded for the entire video. Also, the technique preserves ROIs that impact viewer perception most, while enabling greater truncation for each Y, U, and V vector for the truncation regions with textured MBs. Accordingly, the proposed technique will realize a greater number of truncation while preserving visual quality. The system-level and circuit-level implementations of the proposed technique will be discussed in Section 4.4.

4.4. Proposed Technique: System Level and Circuit Level Implementation

This section presents the system-level and circuit-level implementation of the proposed technique.

4.4.1. System-Level Implementation: Video Streaming Platform

Figure 40 shows the developed system-level video streaming platform. As shown, a Raspberry Pi [23] microcontroller was used to serve as a video streaming server with which a mobile device would communicate and retrieve video data. Also, a Z-Turn 7020 [24] board was utilized and synthesized an H.264 video decoder into the on-board Xilinx Zynq 7020 Field Programmable Gate Array (FPGA) which would operate as a mobile device. Finally, the decoded video data was captured via a Magewell [25] HDMI Video Capture & Display Device.

The corresponding block diagram for Figure 40 is illustrated in Figure 41. The video streaming process is kicked-off by a command from the mobile device to the server to retrieve an encoded H.264 video stream over Secure Copy Protocol (SCP) [26]. The mobile device sends the initial kick-off command to the server over a serial terminal on a PC interfaced with the mobile device over USB.

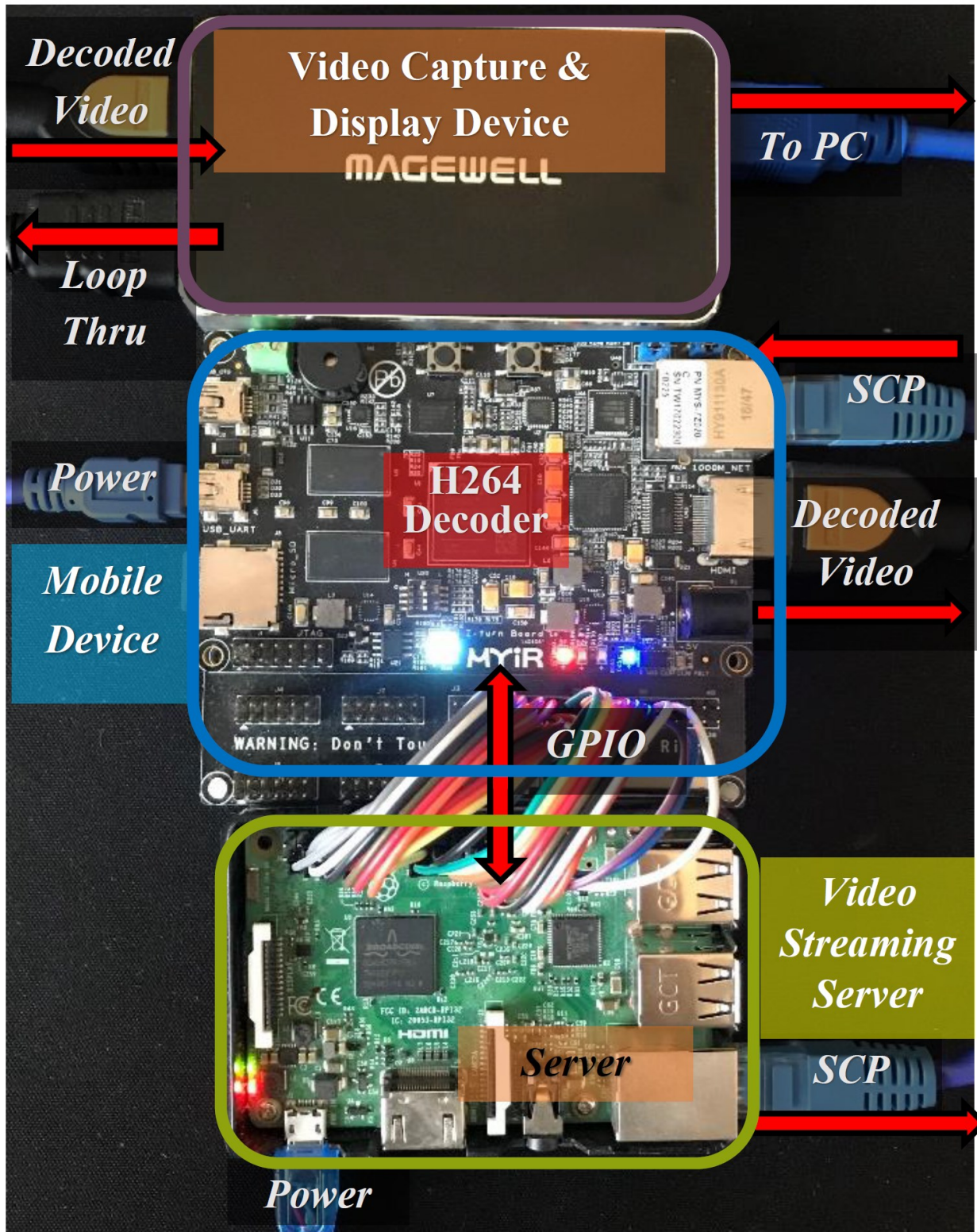


Figure 40. H264 video stream demonstration platform hardware system.

The server then processes the video stream requested by the mobile device by both transmitting an H.264 encoded format of the video stream over SCP to the mobile device and parsing the frames for truncation region information.

Table 10. Truncation Region GPIO Protocol

(a) SERVER-TO-MOBILE DEVICE

Index 0	Index 1	Index 2	Index 3	Index 4	...	Index N+1	Index N+2	Index N+3
Frame Number	Number of Regions	YUV ₁ Truncation	(X ₁₁ ,Y ₁₁)	(X ₁₂ ,Y ₁₂)	...	YUV _N Truncation	(X _{N1} ,Y _{N1})	(X _{N2} ,Y _{N2})
22 bits	16 bits	3 bits	22 bits	22 bits	...	3 bits	22 bits	22 bits

(b) MOBILE DEVICE-TO-SERVER

Index 0	Index 1
Frame Number Request	Send Frame Flag
22 bits	1 bit

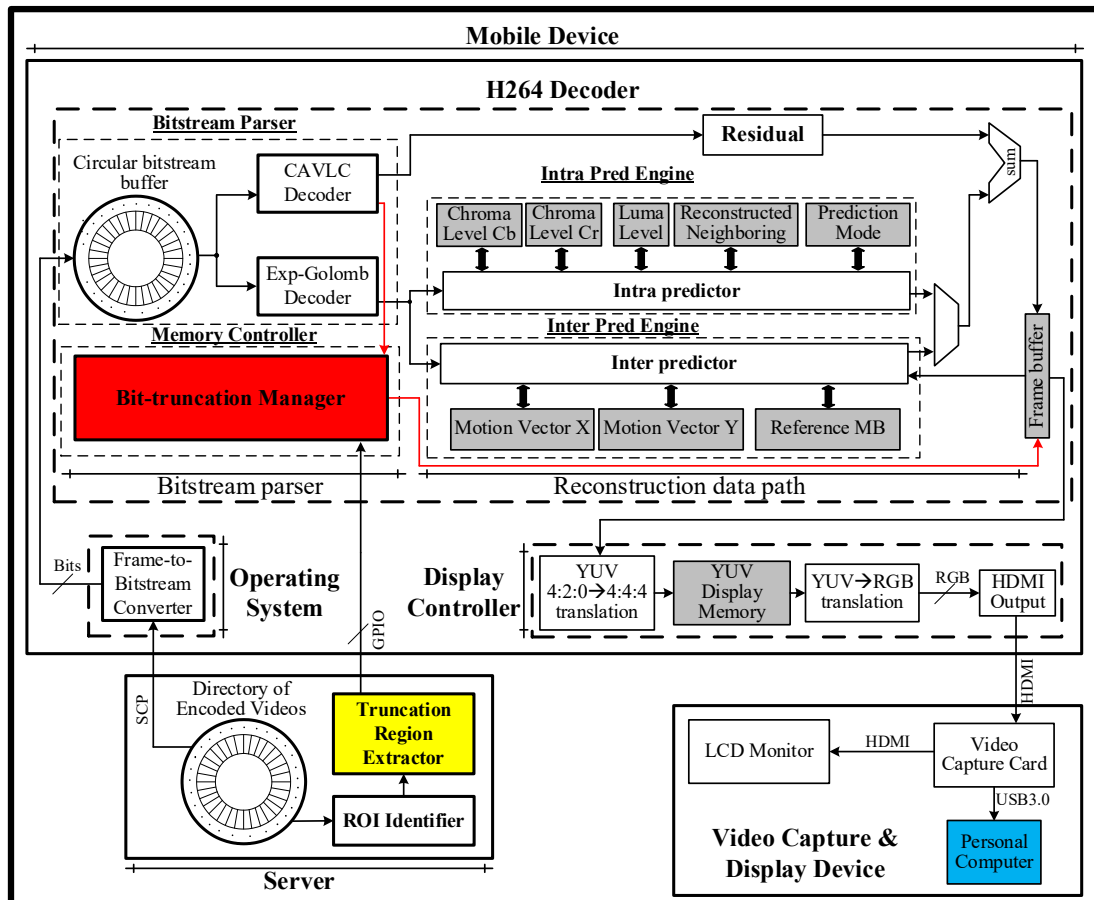


Figure 41. Mobile video steaming system block diagram.

After the video frame is parsed on the server, the truncation region information is transmitted over GPIO per frame. In the developed system, the protocol is defined in Table 10. Only the truncation region information of the frames that would be truncated is transmitted. The preserved ROI information will not be transmitted as these regions are identified prior to the transmission on the server and preserved. As listed in Table 10, the first index, index 0, denotes the current frame number parsed. The second index, index 1, denotes the number of truncation regions to truncate. Then the next indices denote the first three YUV truncation signal bits plus two sets of XY coordinates denoting the left top and right bottom corners of rectangles grouping the affected truncation region. These three indices repeat for each region called out by the “Number of Regions”, index 1. The GPIO interface data width bit size of the developed system is 22-bits per index. The 22-bit distribution is to account for a maximum of 211 x 211 pixel addressing – a max resolution of $1,920 \times 1,080$ – totaling 22 bits. There is an additional 2 handshaking bits between the server and mobile device to denote data reception confirmation in-order to transmit the next index.

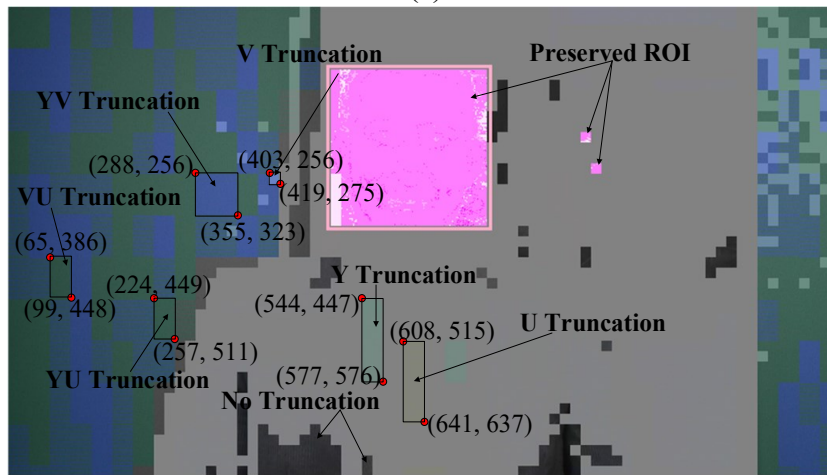
This truncation region information will be transmitted to a MBTM for processing in the mobile device side, as discussed in Section 4.4.2. The MBTM will generate control signals for the frame buffer memory, thereby determining which sub-pixels – from Y, U, and/or V – shall be truncated for each frame written to the frame buffer memory, which will be detailed in Section 4.4.4. Finally, the decoded and bit-truncated frame is output over HDMI from the mobile Device and captured by the Video Capture & Display Device.

4.4.2. Memory Bit Truncation Manager

The MBTM implemented into the H.264 decoder parses the protocol data that is transmitted by the server’s Truncation Region Extractor. The flow is broken down as follows.



(a)



(b)



(c)

Figure 42. (a) Encoded frame 175 from Johnny_1280x720_60 video [28]. (b) Visual of areas being truncated. 45 regions total. (c) Output decoded frame. 2,282,496 bits truncated.

First, from Figure 42 (a), the encoded frame is transmitted via SCP to the mobile device.

Figure 42 (b) illustrates the truncation regions determined to be bit-truncation capable on a sub-

frame vector level: Y vector, U vector, and V vector each encompassing all the sub-frames summing to a frame. From Figure 42 (b), the gray areas denote the truncation regions determined to be bit-truncation capable for all Y, U, and V vectors. The areas in boxes are regions where only 1 or 2 vectors were determined to be bit-truncation capable. Two coordinates, top-left and bottom-right, are highlighted in Figure 42 (b) for each of these regions to show how the truncation region data was used to determine the regions to truncate using the protocol in Table 10. A total of 61 regions to truncate are shown in Figure 42 (b). Figure 42 (c) shows the resultant frame after Figure 42 (a) is decoded using the identified truncation region information. As shown, the preserved ROI around the face, pink region from (b), is not truncated to avoid visual quality degradation. The frame is decoded normally, but when it is written into the frame buffer, the transmitted truncation region information is used to control the T_Y, T_U, and T_V control inputs to truncate the frame buffer memory as it is written. These control inputs are provided to the proposed frame buffer in Figure 43, which will be discussed in detail in Section 4.4.4.

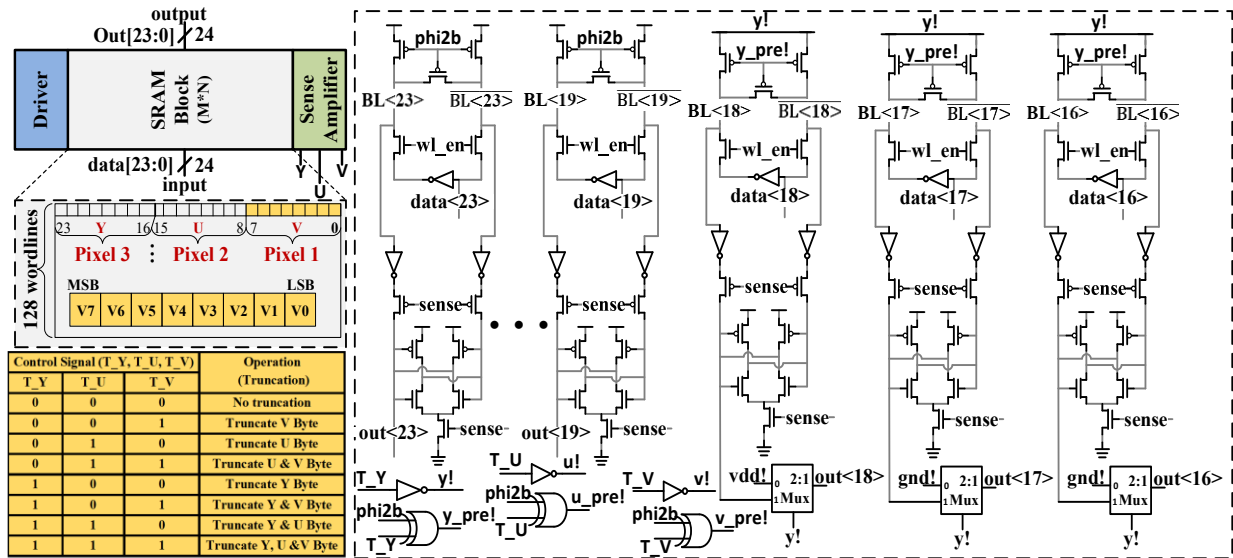


Figure 43. Circuit-Level implementation of the proposed frame buffer memory.

4.4.3. H.264 Decoder and MBTM Integration

A H.264 video decoder is implemented based on the Open Source Osenlogic OSD10 decoder IP [27]. This decoder was capable of decoding baseline profile level 3.1 encoded bitstreams. The slice types supported were I-Slice, SI-Slice, P-Slice, and SP-Slice [28]. The entropy coding profile supported was Context-Adaptive Variable-Length Coding (CAVLC). The decoder took an H264 Network Abstraction Layer (NAL) bitstream and output YUV 4:2:0.

During the NAL bitstream parsing process, the bitstream is parsed into raw bytes of syntax elements from the Raw Byte Sequence Payload (RBSP). Within the RBSP, therein lies the slice layers. Ignoring the Sequence Parameter Set (SPS) and the Picture Parameter Set (PPS), the Instantaneous Decoder Refresh Access Unit (IDR Slice(s)) and the slice layer includes all slice headers and slice data for the frames that shall be truncated using the MBTM. H.264/AVC defines a frame as an array of luma samples and two corresponding arrays of chroma samples: denoted as YUV.

Specifically, the slice header includes the parameters `first_mb_in_slice`, which indicates the position of the first macroblock in the slice data, and `frame_num`, which represents the order in which a video decoder shall decode the encoded frames. This is not the same as the display order or Picture Order Count (POC), which is the order in which the frames are displayed. The `frame_num` parameter is used to determine which frames during the decoding process would be susceptible to YUV bit-truncation by the MBTM and the `first_mb_in_slice` is used to determine the starting coordinates of the macroblocks susceptible to bit-truncation. The slice data included all the macroblocks of the slice.

After the MBTM determined that a frame would be truncated, through a conditional match between the frame number parameter from Table 10 (a) and `frame_num`, a running count

of the current macroblock index was kept track of internally to the MBTM from the slice data starting with the index of `first_mb_in_slice`. After the MBTM determined that a macroblock would be truncated, through a conditional match of the running macroblock index and the truncation region given by the two indices from Table 10 (a) that indicate the rectangular region which YUV truncation would be applied, the MBTM passes through the YUV truncation signal, from Table 10 (a), to the frame buffer which would result in the macroblock being truncated to the desired amount. An internal signal denoting the number of macroblocks truncated in the frame is then incremented. After all the macroblocks desired to be truncated in the frame are truncated, denoted by the number of ROI parameter from Table 10 (a), then from Table 10 (b), the Send Frame Flag is set then reset by the MBTM over GPIO to signal the next frame information to truncate. From Table 10 (b), the Frame Number Request index is used to fetch any frame index truncation information for macroblocks that required multiple frames for prediction. This process is repeated until the end of the NAL bitstream.

The trade-off with utilizing the BTM is the additional GPIO parallel bitstream overhead required to truncate the macroblocks in each frame. Each frame parsed had an absolute worst case overhead of approximately 380,738 additional bits to transmit using the protocol from Table 10. This worst case is calculated assuming every macroblock with 16×16 pixels in a maximum resolution of $1,920 \times 1,080$ would be truncated differently per frame in a video. On average, however, the number of additional bits transmitted per frame is 1,200, because the maximum resolution of each frame is 1920×1080 and the truncation regions are combined to encompass a greater area in the video to save on bits transmitted: on average 50 truncation regions per frame. With a 1920×1080 video at 30 frames per second progressive (1080p 30fps) or a 1280×720 videos at 60 frames per second progressive (720p 60fps), i.e. 5,000 kbps bit rate, the worst case

percentage overhead would be 7.62% with an average of 0.02% per frame. The protocol utilized is one of the simplest methods to implement the proposed technique.

4.4.4. Circuit-Level Implementation of the Proposed Frame Buffer Memory

During the video decoding process, multiple memories are needed. In particular, the frame buffer memory is accessed very frequently and it has a profound influence on the system's overall cost and power consumption [7]. In this chapter, a new frame buffer is designed, and the circuit-level implementation is shown in Figure 8. Specifically, the logic in the truth table highlighted in yellow was designed to be supported by the MBTM. Here, T_Y, T_U, and T_V are utilized to truncate Y, U, and V byte from the word. Each word consists of a Y, U, and V byte. During the Write Enable (WE) phase of the frame buffer memory access, if either control line of T_Y, T_U, and / or T_V are asserted, the memory would truncate the 3-LSB of the optimal asserted vector as "100" [7].

The proposed frame buffer has M words and each word consists of N bits. To evaluate the functionality and measure average power consumption of this proposed circuitry, a 128-word by 24-bits memory array is designed. Here, input and output pins are denoted as data[23:0] and out[23:0] respectively. Bits 23-16 are named Y byte, bits 15-8 are named U byte, and bits 7-0 are named V byte. The memory implemented had a driver and sense amplifier for writing and reading data. These enabled bit truncation according to T_Y, T_U, and T_V control signal activation. If T_Y, T_U, and T_V are all de-asserted as logic '0', then the frame buffer would operate as a traditional memory device where the sense amplifier would operate with a supply voltage (VDD) and pre-charge signal phi2b. When the T_Y signal is asserted as logic '1', the peripheral circuitry would generate two signals: y! which is the inverted value of T_Y and y_pre! which is inverted value of the pre-charge enable signal. These two signals are used to control the

sense amplifier for the Y byte's 3-LSBs, thereby enabling truncation. During this process, the VDD for this sense amplifier remains grounded and the pre-charge signal would be reactivated. As a result, the power consumption of this portion of circuitry will be reduced as compared to the normal operation. During the read back operation, the 3-LSBs are generated as "100" through use of three 2:1 multiplexers in-place of regular of data output. When the bit truncation is asserted, these multiplexers would select "100" through control signals $y!$, $u!$, or $v!$. Otherwise, these multiplexers would pass normal readout data values. In addition, the VDD of all the 3 LSBs of each byte are also controlled by the corresponding control signals $y!$, $u!$, and $v!$. During the truncation, VDD for LSBs can be powered off to save power consumption and multiplexers will select "100" as the output data, thereby achieving low-power operation. The detailed timing diagram and power efficiency of the proposed memory will be discussed in Section 4.6.

4.5. Experimental Methodologies

This section discusses the metrics, methods, and strategies used to evaluate the effectiveness of the proposed technique. The testing and analysis setup used to generate the experimental results is also discussed.

4.5.1. Video Selection

To verify the effectiveness of the proposed technique, 74 videos with diverse characteristics were selected from the YouTube 8M dataset [14], YouTube UGC dataset [29], and Xiph.org Video Test Media [30]. As shown in Table 12, those videos have different resolutions (e.g. 288×352 , 1280×720 , and 1920×1080) and different MB variance characteristics (low, medium, and high). Of those videos, 60 videos contain facial features to enable ROI preservation using the proposed technique. All videos were converted to the YUV 4:2:0 chroma subsampling standard for ease of bit-truncation. A detailed statistical analysis

shows that the selected videos are representative of the full population of videos in general, which will be discussed in Section 4.6.5.

4.5.2. Video Frame Quality Metrics

The traditional video quality metric PSNR is applied widely to evaluate video quality, but it fails to incorporate the importance of ROI. This is because this metric weights all pixels of the video equally, regardless of user awareness. For this reason, another video quality metric – Weighted Peak Signal to Noise Ratio (WPSNR) – is used in this chapter to evaluate the quality of videos with ROI [22], which is defined as [31]:

$$\text{WPSNR} = 10 \log_{10}(255^2 / D_{frame}) \quad (\text{Equation 16})$$

$$D_{frame} = \alpha * \text{MSE}(f, f') + (1 - \alpha) * \text{MSE}(f, f') \quad (\text{Equation 17})$$

where MSE stands for the Mean Squared Error between the original frame and after truncation while α (alpha) is defined as the weight that the ROI would have. The α value will be a constant value of 0.9 following the previous research in [22]. In the analysis, PSNR was used for videos without ROI and WPSNR is used for videos with ROI.

4.5.3. System-Level and Circuit-Level Implementation

The hardware system platform from Figure 40 implemented an H264 decoder synthesized into a Xilinx Zynq XC7Z010 FPGA fabric. The H264 decoder IP Core was designed using the Xilinx Vivado 2019.2 [32] software design suite. This same decoder is modified to include a MBTM. The FPGA was commanded via an ARM Cortex-A9 Processor running on a Linux Operating System through a custom baseband driver.

The circuit-level frame buffer is implemented using a 45nm CMOS technology[33]. The supply voltage is 1.0V. The memory size is 128 words at 24 bits per word.

4.5.4. Video Quality Evaluation

All selected videos were analyzed using an in-house custom software tool. The tool operated in the following three-step process: (i) Load one original video frame from memory; (ii) Apply both the method in [7] and the proposed method to the original frame and generate the truncated frame using each method; and (iii) Compare the frames generated against the original frame and calculate the PSNR and WPSNR values. With data points collected on a per-frame basis, the average PSNR and WPSNR of each video stream was calculated and compared.

4.5.5. Statistical Hypothesis Validation

From the proposed method, a hypothesis was conjectured: that the differences between the method in [7] and the proposed method follow a Normal, or near-Normal distribution. This should hold true for both PSNR and WPSNR. To support this hypothesis, a goodness of fit regression test was performed to determine if the data falls within the probability plot of a Normal or Weibull distribution. The intention behind this analysis was to identify patterns in the output videos that serve to estimate the differences in quality and noise for any given input video. If the data follows this hypothesis, this would suggest that the sample set of videos is of adequate size and as a result, no more videos would need to be tested.

4.6. Experimental Results

4.6.1. Hardware FPGA System MBTM Overhead

Figure 45 and Figure 46 show the post-implementation project summaries of the baseline H264 decoder and the H264 decoder modified to include an MBTM. When comparing both figures, one observes that the Lookup Table (LUT) overhead, which is the additional logic gates required for the proposed design over the baseline, was 204 LUTs or a 0.38% increase in area. The I/O, which was used for the server-to-mobile device interface, increased by 37, or 29.6%.

The power consumption of the modified decoder also increased by 0.068 watts: most of which was attributed to the increased number of I/O. Finally, the Worst Negative Slack (WNS) increased by 0.011ns, which was within acceptable tolerance for this system as any positive value means that the critical path passes timing constraints. Overall, this additional overhead was tolerable when compared against the benefits in power savings and quality improvements achieved using the proposed technique.

4.6.2. Circuit-Level Frame Buffer Timing Diagram

The proposed frame buffer is shown in Figure 43 and the simulation timing diagram is shown in Figure 44. In this waveform, ϕ_{2b} , T_Y , $y!$, and $y_pre!$ denote the pre-charge (for untruncated bits), bit truncation enable for Y byte, power supply for truncated bitcell's (last 3 LSBs of Y byte), and pre-charge deactivated signal for truncated bit cells, respectively. T_U and T_V controlled the bit truncation for U and V bytes respectively. Write and read enable signals initiated the write and read operations for the memory accordingly. Data [23:0] were the three bytes of each word of the proposed memory buffer. Here, blue to red lines stand for "don't care" regions. The red lines denote where the rising clock edge was initiated for write and read operations. Finally, the green lines denote that the write and read operations were enabled. All 8 truncation permutations and traditional read and write operations were presented in the timing diagram as an exhaustive simulation of the frame buffer circuit.

It should be noted that, if the bit truncations were initiated, then 3 LSBs were truncated from the selected byte/bytes based on the control signals T_Y , T_U , and T_V . During the read operations, the 3-LSBs of the truncated bytes would output "100" bits through the utilization of 2:1 multiplexers instead of being read from memory to save power.

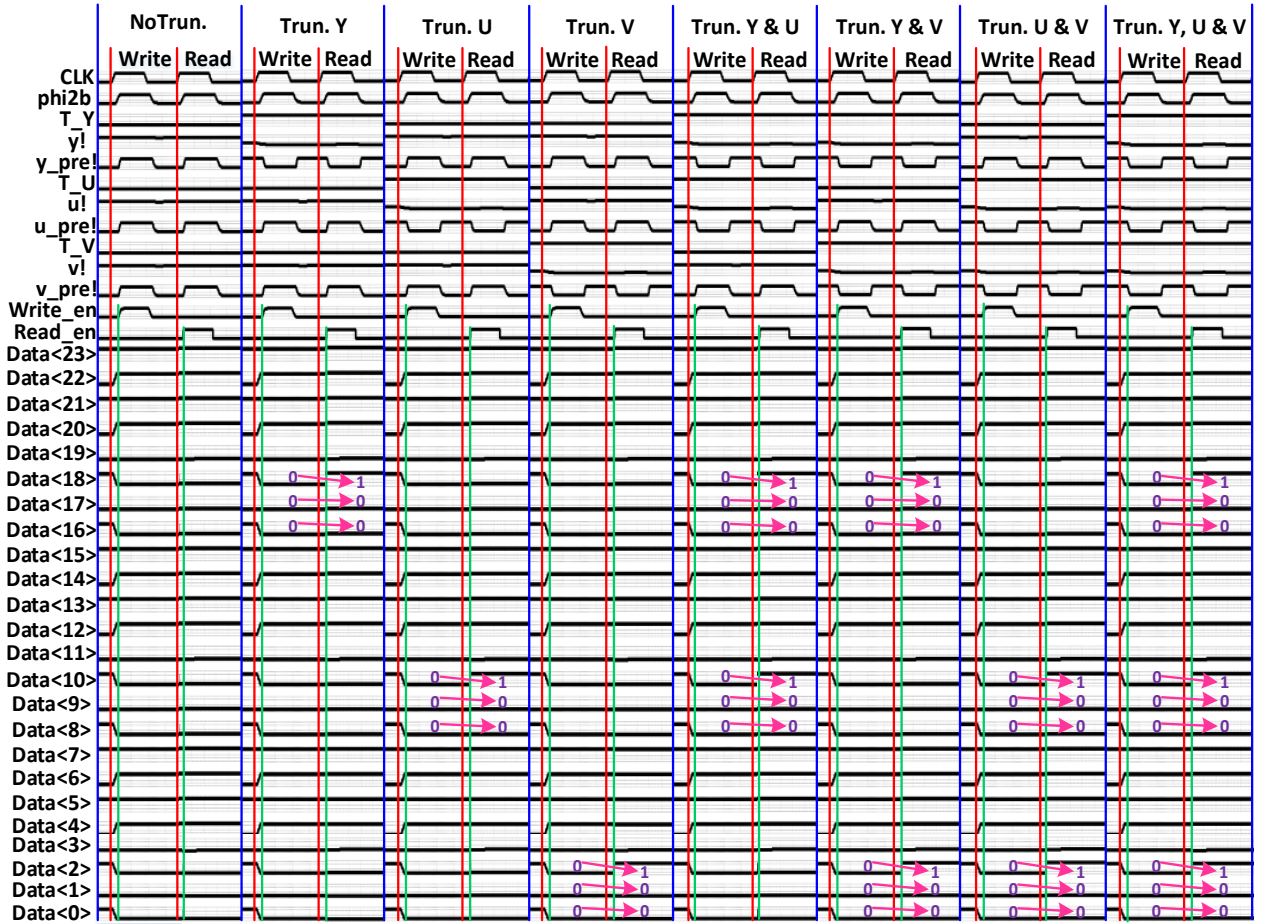


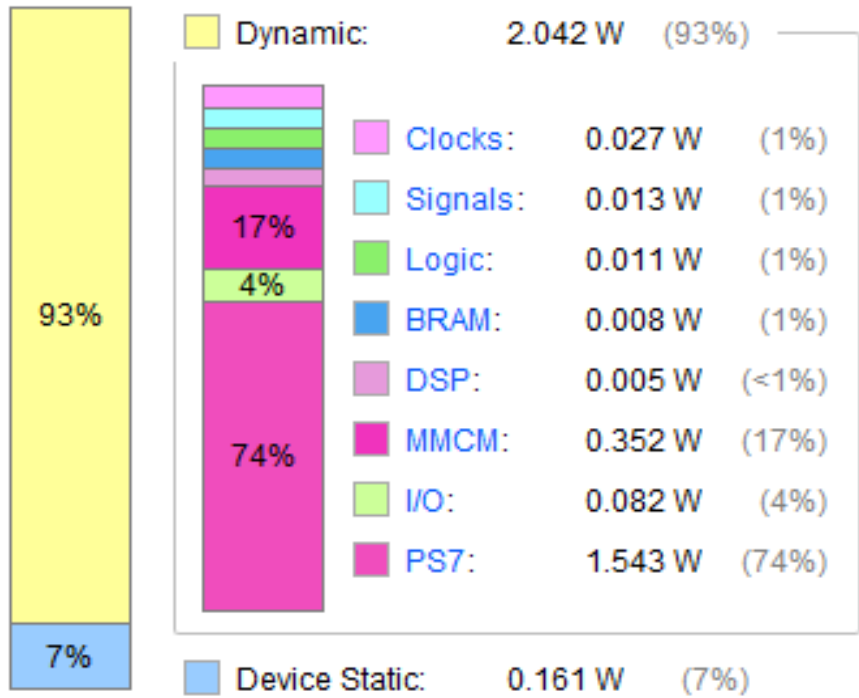
Figure 44. Timing diagram of the frame buffer circuit.

4.6.3. Circuit-Level Frame Buffer Power Saving Analysis

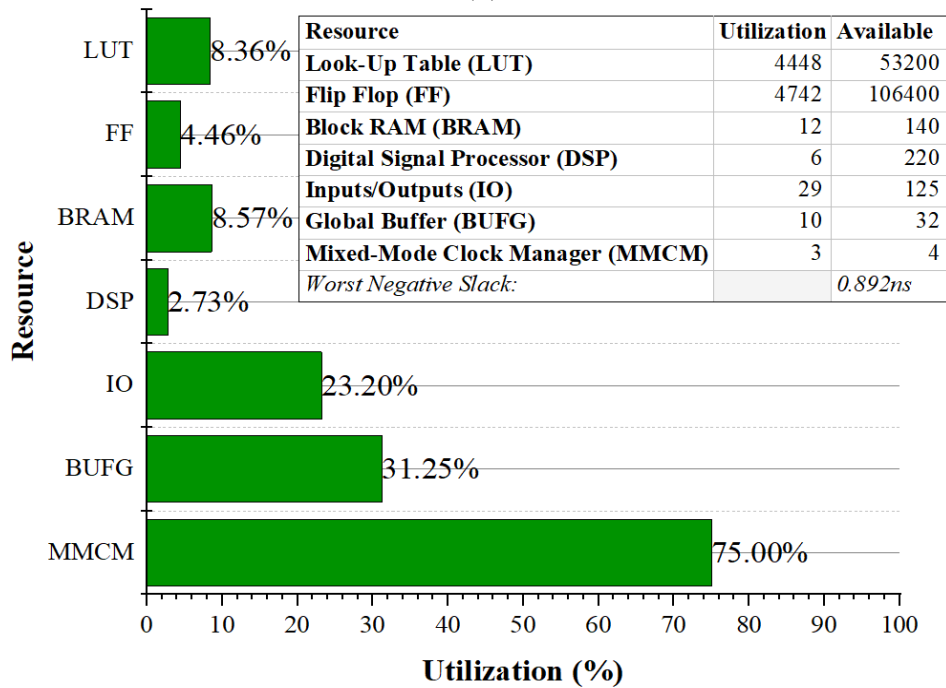
Figure 47 presents the power consumption of the proposed frame buffer in all eight possible conditions, including seven truncation cases and one baseline case without bit truncation. Specifically, the eight cases include: (i) No truncation with control signals T_Y & T_U & $T_V = '0'$, (ii) Y vector truncation with $T_Y = '1'$, (iii) U vector truncation with $T_U = '1'$, (iv) V vector truncation with $T_V = '1'$, (v) Y and U vectors truncation with T_Y & $T_U = '1'$, (vi) U and V vectors truncation with T_U & $T_V = '1'$, (vii) V and Y vectors truncation with T_V & $T_Y = '1'$, and (viii) YUV vectors truncation with T_Y & T_U & $T_V = '1'$. As discussed in Section 4.3.2, for the truncated vectors, the three LSB will be truncated to “100” to maximize

power savings. All 8 truncation cases presented in Figure 47 are tested in 6 ways: when written ('0' to '0', '0' to '1', '1' to '0', '1' to '1') and when read back ('0' & '1'). The power consumed in each case was calculated, and then the average is presented.

At first, a random word was initialized with (A5A5A5)₁₆, then the same memory word was immediately read back with (F0F0F0)₁₆, then all the '1's and '0's written and read received the same priority in the power consumption calculations. The same word consumed 3.90E-4 W power without any bit truncation. When the circuitry selected any T_Y, T_U or T_V control option, where 3-LSBs were truncated from each one selected, 6.67% power was saved when compared against no bit truncation. When T_Y & T_U, T_Y & T_V or T_U & T_V were selected, where 3-LSBs were truncated from each selected byte, then 13.33% power was saved when compared against no bit truncation. Finally, when T_Y, T_U, and T_V were selected for truncation, where 3-LSBs were truncated from each selected byte, then 19.74% power was saved. The supply voltage for this simulation was 1V, where the proposed frame buffer circuit can operate to specification and had no faulty bit(s).

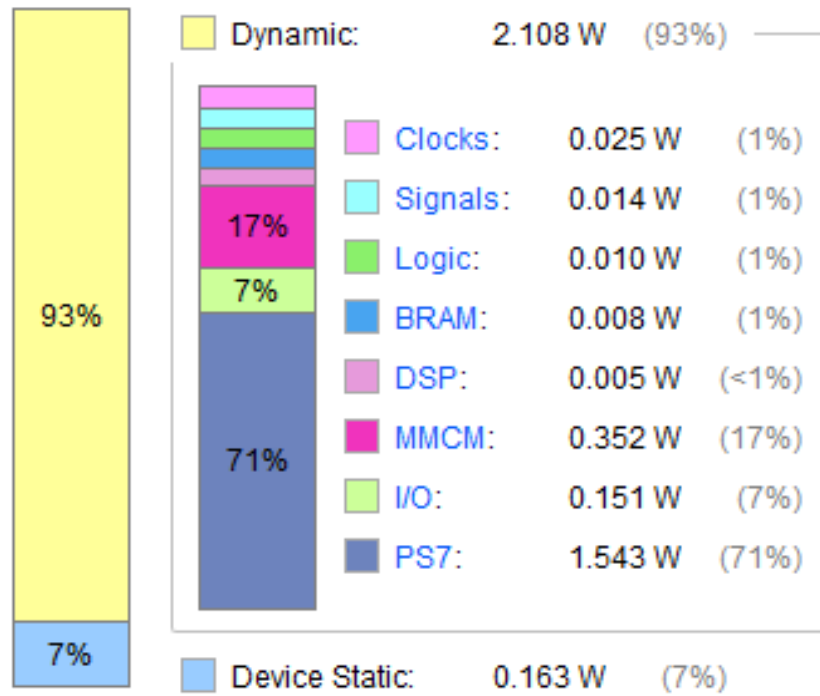


(a)

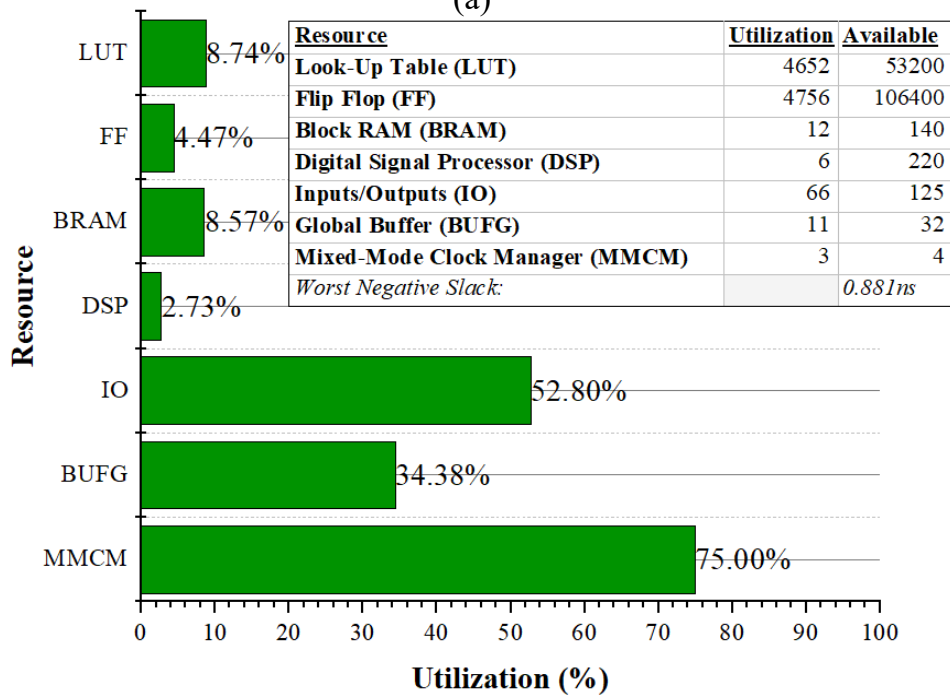


(b)

Figure 45. Hardware FPGA system post-implementation project summary without BTM. (a) On-Chip Power, Total Power: 2.203W. (b) Resource allocation.



(a)



(b)

Figure 46. Hardware FPGA system post-implementation project summary with BTM. (a) On-Chip Power, Total Power: 2.271W. (b) Resource allocation.

Table 11. Visual Comparison of Selected Video Frames

Foreman_cif frame 0		
	Proposed Method: WPSNR = 54.03 dB	Previous Method [7]: WPSNR = 37.00 dB
mother_daughter_cif frame 299		
	Proposed Method: WPSNR = 53.77 dB	Previous Method [7]: WPSNR = 48.03 dB
carphone_qcif frame 248		
	Proposed Method: WPSNR = 54.52 dB	Previous Method [7]: WPSNR = 48.04 dB

4.6.4. Video Visual Quality Comparisons

Table 11 shows visual frame comparisons for three selected videos with ROI between the proposed method and [7]. It can be seen that the proposed technique enables significant visual quality improvement as compared to [7]. Specifically, for the Foreman_cif video, due to the truncated LSBs in [7], the man's cheeks, forehead, and hat shadows experience noticeable banding distortion, negatively affecting video quality. Alternatively, the proposed ROI-aware technique effectively reduces the banding distortion and improves the visual quality. Similarly, with [7], the mother_daughter_cif demonstrates banding distortion around the cheeks and hair, and the carphone_qcif video suffers from discoloration around the cheeks and chin. The introduced ROI awareness of the proposed technique effectively avoids losing the quality of videos. Another observation from Table 11 is that the proposed technique achieves a much higher WPSNR value of all three videos. A more detailed analysis on WPSNR will be provided in next sub-section.

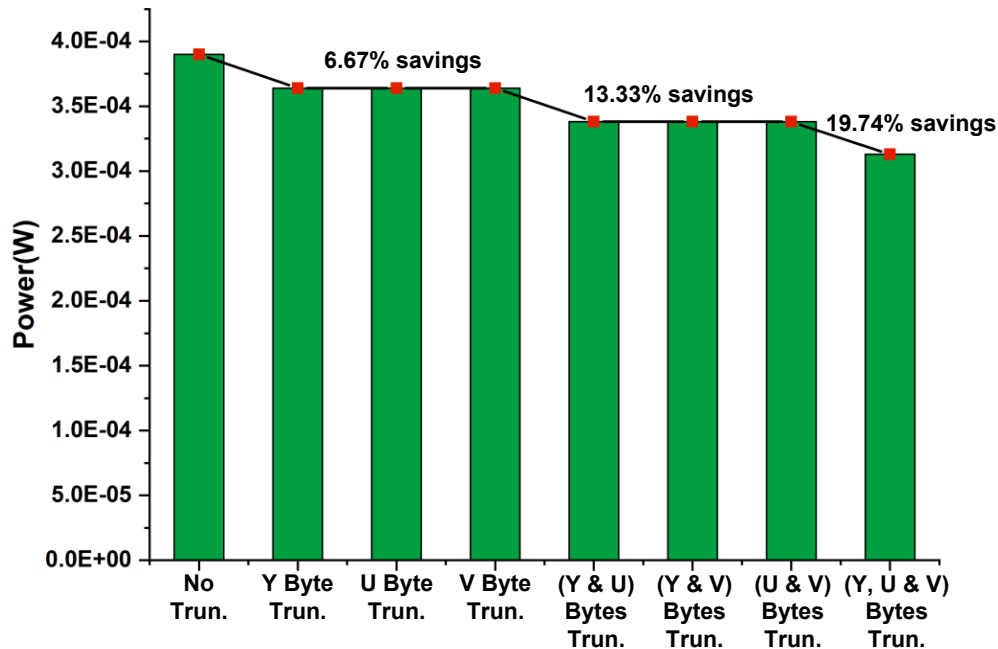


Figure 47. Power savings (one word) of the frame buffer circuit.

4.6.5. Objective Video Quality and Bit Truncation Analysis

Table 12 compares WPSNR values and the number of truncated bits of 60 videos with ROI using the proposed technique to the state-of-the-art [7]. As shown, the proposed technique can enable 26.46% additional truncated bits as compared to [7]. Meanwhile, with the ROI awareness, the proposed technique can effectively enhance the quality of the majority of videos. On average, the proposed technique can increase the WPSNR values by 20.17% videos, as compared to [7].

The impact of the MB variance characteristics (low, medium, and high variance) on the effectiveness of the proposed technique was analyzed. The results are shown in Figure 48. As can be seen, the WPSNR improvement strongly depends on the MB variance of videos. Specifically, videos with high variance achieve the most significant quality improvement using the proposed technique, with 47.31% WPSNR increase on average. With the proposed technique, all videos with medium variance also demonstrate quality improvement, with 13.74% WPSNR increase on average. However, the proposed technique shows little video quality improvement for videos with low variance and even results in minimal video quality degradation (with 1.75% WPSNR loss on average). This suggests that the proposed technique is particularly effective for videos with high and medium MB variance.

Finally, the results of 14 videos without ROI were analyzed. As shown in Table 12, the proposed technique can enable a significant number of truncated bits, with a minimal PSNR drop. On average, 44.61% additional truncated bits can be achieved, with 3dB PSNR loss.

4.6.6. Video-Level Power Saving Analysis

To compare the power effectiveness of the proposed ROI-aware technique to the traditional memory design and the state-of-the art [7], the power consumption of the memory for a video was modeled as:

$$\left\{ \begin{array}{l} P(\text{Video}_i) = \frac{1}{N_i} \sum_{j=1}^{N_i} P_k(j) \\ k \in (0,1,2,3) \end{array} \right. \quad (\text{Equation 18})$$

where N_i is the total number of bytes for the video i , $P_k(j)$ is the normalized power consumption to store byte j with k truncated bits. For the proposed memory, $k = 3$; for the traditional memory, $k = 0$; for the memory in [7], $k = 0, 1, 2, \text{ or } 3$. For a fair comparison, the normalized power consumption $P_k(j)$ is based on the power consumption reported in [9]. The results are listed in Table 12 and Table 13.

Table 12. Selected ROI Videos. Analysis Results

Videos with ROI	Truncated bits			Normalized power consumption			WPSNR (Alpha = 0.9)		
	Ref.[7]	Proposed	diff	Ref. [7]	Proposed	diff	Ref. [7]	Proposed	diff
akiyo cif	30,412,800	32,922,081	8.25%	90.97%	93.55%	-2.83%	57.53	55.14	-4.16%
claire qcif	50,079,744	44,009,266	-12.12%	90.97%	94.76%	-4.16%	57.61	57.10	-0.88%
dinner 1080p30	1,969,920,000	1,629,113,461	-17.30%	90.97%	95.07%	-4.50%	57.32	58.21	1.56%
grandma qcif	88,197,120	78,023,685	-11.53%	90.97%	94.73%	-4.12%	57.57	57.21	-0.62%
intros 422 cif	36,495,360	43,295,433	18.63%	90.97%	92.93%	-2.15%	57.45	55.15	-4.01%
Johnny 1280x720 60	553,881,600	456,595,131	-17.56%	90.97%	95.09%	-4.52%	57.07	58.44	2.39%
KristenAndSara 1280x720 60	553,881,600	488,596,289	-11.79%	90.97%	94.74%	-4.14%	56.99	57.37	0.65%
miss am qcif	15,206,400	10,966,364	-27.88%	90.97%	95.70%	-5.20%	57.60	58.54	1.63%
news cif	30,412,800	36,162,420	18.91%	90.97%	92.91%	-2.13%	57.52	54.48	-5.27%
rush hour 1080p25	1,036,800,000	1,134,324,798	9.41%	90.97%	93.48%	-2.75%	57.49	55.78	-2.97%
sign irene cif	54,743,040	64,431,060	17.70%	90.97%	92.98%	-2.21%	57.48	55.01	-4.29%
trevor qcif	15,206,400	16,565,578	8.94%	90.97%	93.50%	-2.78%	57.31	54.99	-4.05%
vidyo1 720p 60fps	553,881,600	597,801,678	7.93%	90.97%	93.57%	-2.85%	57.54	56.05	-2.58%
west wind easy 1080p	1,181,952,000	1,086,498,282	-8.08%	90.97%	94.52%	-3.90%	56.81	55.68	-1.99%
720p50 mobcal ter	928,972,800	1,325,076,458	42.64%	86.60%	82.99%	4.17%	47.88	54.16	13.12%
720p50 shields ter	928,972,800	1,245,591,004	34.08%	86.60%	84.01%	2.99%	47.69	54.48	14.25%
aspen 1080p	2,363,904,000	3,041,639,112	28.67%	86.60%	84.66%	2.24%	47.92	54.97	14.71%
blue sky 1080p25	899,942,400	1,060,438,048	17.83%	86.60%	85.95%	0.75%	47.66	55.10	15.61%
bowing cif	60,825,600	76,915,166	26.45%	86.60%	84.92%	1.94%	48.02	53.69	11.81%
bridge close cif	405,504,000	560,890,106	38.32%	86.60%	83.51%	3.57%	47.95	54.13	12.88%
carphone qcif	77,451,264	88,390,034	14.12%	86.60%	86.39%	0.24%	48.04	54.52	13.48%
controlled burn 1080p	2,363,904,000	2,937,098,762	24.25%	86.60%	85.18%	1.63%	47.96	55.18	15.06%
crew 4cif	121,651,200	172,691,140	41.96%	86.60%	83.07%	4.07%	47.54	53.54	12.61%
crowd run 1080p50	2,073,600,000	3,005,452,078	44.94%	86.60%	82.72%	4.48%	47.42	53.55	12.93%
deadline cif	278,581,248	334,134,316	19.94%	86.60%	85.70%	1.04%	48.02	54.48	13.45%
FourPeople 1280x720 60	1,107,763,200	1,318,759,148	19.05%	86.60%	85.80%	0.92%	47.96	55.12	14.94%
Lecture 1080P-412e	1,034,726,400	998,909,402	-3.46%	86.60%	88.49%	-2.18%	48.07	57.01	16.91%
life 1080p30	3,421,440,000	4,187,455,252	22.39%	86.60%	85.41%	1.38%	48.04	54.99	14.46%
mother daughter cif	60,825,600	79,283,536	30.35%	86.60%	84.46%	2.47%	48.03	53.78	11.95%
pamphlet cif	60,825,600	83,561,818	37.38%	86.60%	83.62%	3.44%	47.93	53.27	11.14%
paris cif	215,930,880	302,557,572	40.12%	86.60%	83.29%	3.82%	47.88	53.81	12.40%
pedestrian area 1080p25	1,555,200,000	1,749,179,384	12.47%	86.60%	86.59%	0.01%	48.05	55.59	15.70%
rush field cuts 1080p	2,363,904,000	3,080,806,912	30.33%	86.60%	84.46%	2.47%	47.86	54.09	13.01%
salesman qcif	91,035,648	127,208,586	39.73%	86.60%	83.34%	3.77%	47.97	53.74	12.03%
station2 1080p25	1,298,073,600	1,803,902,208	38.97%	86.60%	83.43%	3.66%	47.79	53.71	12.39%
students cif	204,171,264	283,317,790	38.76%	86.60%	83.45%	3.63%	47.92	53.64	11.93%
sunflower 1080p25	2,073,600,000	2,874,084,316	38.60%	86.60%	83.47%	3.61%	47.81	53.79	12.52%
suzie qcif	30,412,800	31,039,198	2.06%	86.60%	87.83%	-1.42%	48.07	55.79	16.07%
touchdown pass 1080p	2,363,904,000	2,715,649,830	14.88%	86.60%	86.30%	0.35%	47.94	55.38	15.52%
tractor 1080p25	2,861,568,000	4,031,161,306	40.87%	86.60%	83.20%	3.92%	47.50	53.67	12.99%
vidyo3 720p 60fps	1,107,763,200	1,368,042,822	23.50%	86.60%	85.27%	1.53%	48.11	54.54	13.36%
vidyo4 720p 60fps	1,107,763,200	1,206,225,090	8.89%	86.60%	87.02%	-0.48%	48.13	55.78	15.89%
720p50 parkrun ter	1,393,459,200	2,074,332,336	48.86%	82.11%	73.37%	10.64%	36.71	53.47	45.63%
720p5994 stockholm ter	1,669,939,200	2,434,415,832	45.78%	82.11%	73.93%	9.97%	35.49	53.38	50.41%
ducks take off 1080p50	3,110,400,000	4,661,102,586	49.86%	82.11%	73.20%	10.86%	36.36	53.30	46.61%
football 422 cif	109,486,080	161,366,445	47.39%	82.11%	73.64%	10.32%	36.54	53.96	47.67%
football cif	79,073,280	114,873,738	45.28%	82.11%	74.02%	9.86%	36.58	53.79	47.03%
foreman cif	91,238,400	123,714,882	35.60%	82.11%	75.75%	7.75%	37.01	54.04	46.01%
hall monitor cif	91,238,400	136,539,606	49.65%	82.11%	73.23%	10.82%	36.66	53.56	46.12%
harbour 4cif	182,476,800	268,811,991	47.31%	82.11%	73.65%	10.31%	36.34	53.91	48.34%
ice 4cif	145,981,440	183,650,610	25.80%	82.11%	77.50%	5.62%	35.22	52.91	50.21%
mobile calendar 422 cif	109,486,080	163,476,849	49.31%	82.11%	73.29%	10.74%	36.37	53.06	45.88%
old town cross 420 720p50	1,382,400,000	2,060,977,467	49.09%	82.11%	73.33%	10.69%	36.44	53.60	47.07%
riverbed 1080p25	1,555,200,000	2,285,697,270	46.97%	82.11%	73.71%	10.23%	36.62	53.29	45.54%
silent cif	91,238,400	130,669,137	43.22%	82.11%	74.38%	9.41%	36.70	53.46	45.64%
soccer 4cif	182,476,800	249,118,389	36.52%	82.11%	75.58%	7.96%	36.49	53.36	46.25%
tennis sif	45,619,200	67,173,204	47.25%	82.11%	73.66%	10.29%	36.27	54.29	49.69%
tt sif	34,062,336	50,343,861	47.80%	82.11%	73.56%	10.41%	36.16	54.24	49.98%
vtclnw 422 ntsc	109,486,080	146,642,766	33.94%	82.11%	76.04%	7.39%	36.67	53.74	46.55%
washdc 422 ntsc	109,486,080	163,223,193	49.08%	82.11%	73.33%	10.69%	36.48	53.62	46.99%
AVE			26.46%	86.27%	83.79%	3.06%			20.17%

Table 13. Results of NON-ROI Videos

Videos without ROI	Truncated bits			Normalized power consumption			PSNR loss (dB)		
	Ref. [7]	Proposed	diff	Ref. [7]	Proposed	diff	Ref. [7]	Proposed	diff
bus cif	30,412,800	43,042,560	41.53%	86.60%	82.47%	4.77%	48.15	40.82	7 dB
galleon 422 cif	72,990,720	102,643,456	40.63%	86.60%	83.23%	3.89%	48.20	40.72	7 dB
highway cif	405,504,000	569,610,752	40.47%	86.60%	83.25%	3.87%	48.24	41.14	7 dB
tempete cif	52,715,520	77,495,808	47.01%	86.60%	83.12%	4.01%	48.12	40.81	7 dB
bridge far cif	638,972,928	958,070,016	49.94%	82.11%	73.18%	10.88%	42.56	40.60	2 dB
city 4cif	182,476,800	271,175,040	48.61%	82.11%	73.42%	10.59%	42.48	40.84	2 dB
coastguard cif	91,238,400	120,874,752	32.48%	82.11%	76.30%	7.08%	42.48	41.07	1 dB
container cif	91,238,400	125,445,888	37.49%	82.11%	75.41%	8.17%	42.45	41.01	1 dB
flower cif	76,032,000	107,439,360	41.31%	82.11%	74.72%	9.00%	42.50	40.93	2 dB
flower garden 422 cif	109,486,080	162,798,336	48.69%	82.11%	73.40%	10.61%	42.57	40.62	2 dB
garden sif	34,974,720	52,448,256	49.96%	82.11%	73.18%	10.88%	42.52	40.73	2 dB
husky cif	76,032,000	111,882,240	47.15%	82.11%	73.68%	10.27%	42.48	40.91	2 dB
mobile cif	91,238,400	136,164,864	49.24%	82.11%	73.31%	10.73%	42.57	40.70	2 dB
waterfall cif	79,073,280	118,609,920	50.00%	82.11%	73.17%	10.89%	42.40	40.63	2 dB
AVE			44.61%	83.40%	76.56%	8.26%			3dB

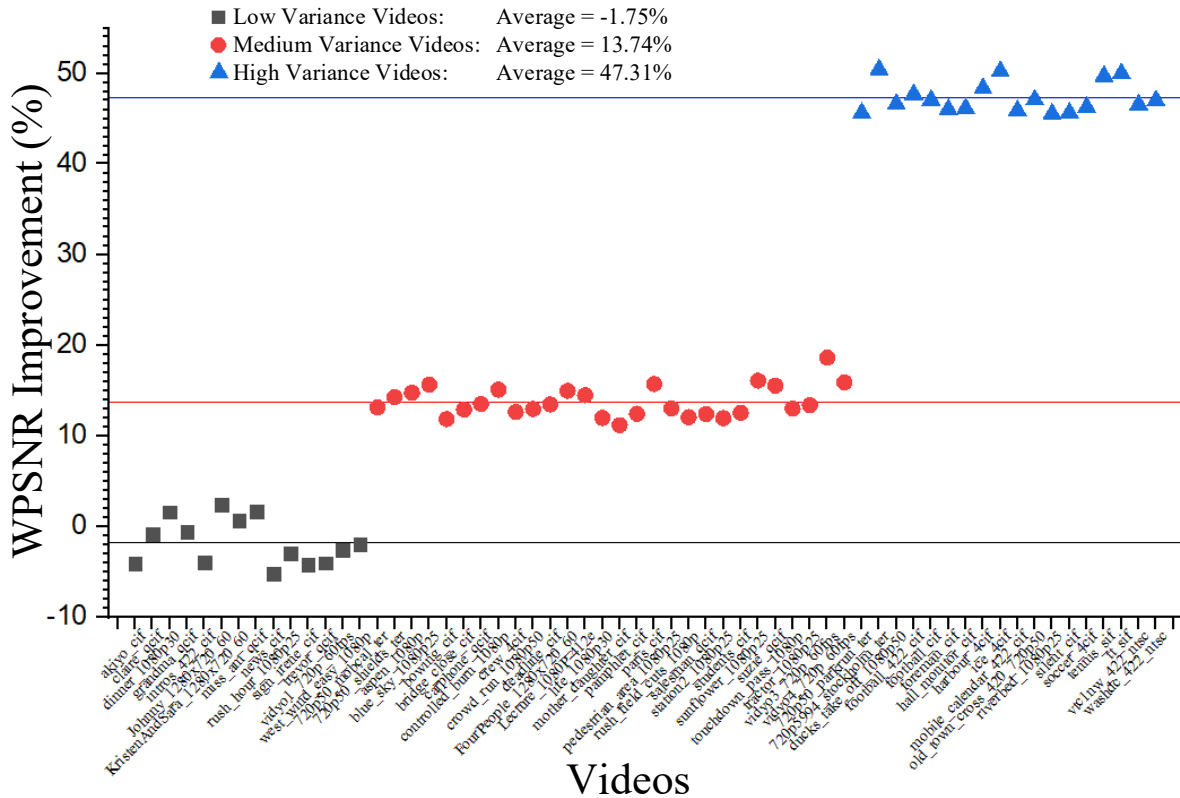


Figure 48. Impact of the video content characteristics on the effectiveness of the proposed technique, compared to old technique.

As observed, the proposed technique only consumes 83.79% and 76.56% total power on average for videos with ROI and videos without ROI, respectively, as compared to the traditional

memory. Also, the proposed technique achieves 3.06% and 8.26% power savings for videos with ROI and videos without ROI, respectively, as compared to [7]. It is worth mentioning that, the analysis only considers the facial features as ROI of videos and integrating advanced ROI identification algorithms will convert videos without ROI to videos with ROI, thereby further increasing the effectiveness of the proposed technique to general videos.

4.6.7. Statistical Analysis

In-order to confirm that the selected video analysis results are a representation of the full population of all videos, a statistical analysis of the results was carried out. The statistical analysis was verified to determine that the results are relevant across all videos not analyzed. Specifically, the Pearson's Chi-square test [33], which is also known as the Chi-Squared goodness-of-fit test, is used in the analysis. The goodness-of-fit test checks whether the sample data is likely to be from a specific theoretical distribution, and therefore represents the data expected in the actual population. The idea is, if the sample data does fit an expected distribution, then it shows that the sample data represents the full population of the video data in existence. The statistical results will either reject or accept the working statement called the null hypothesis, H_0 , which is the opposite of the alternative hypothesis, H_1 . To reject or accept the null hypothesis, several methods exist, one of which is the Probability value method i.e. P-Value method. The P-Value is the evidence against the null hypothesis, i.e., the smaller the P-Value, the stronger the evidence that the null hypothesis should be rejected. The P-Value method is based on a critical value, which is determined based on the distribution. For example, if a normally distributed population was dealt with – which is the case according to the statistical results shown later, this critical value is a z-score. The z-score is a value that is then used to lookup the P-Value in a Standard Normal z-table, which is used to then test the null hypothesis.

If a P-Value is greater than an alpha or α value of 0.10, then the statistical results are “not significant” and thus, the null hypothesis is accepted. However, if the P-Value is less than or equal to α values of 0.05 or 0.01, then the results are “significant” or “highly significant” respectively, and thus, the null hypothesis is rejected in favor of the alternative hypothesis. The rejection regions depend on the confidence level that the results are significant, e.g., if a confidence level is 95%, then an α value of 5% or 0.05 is chosen: 100% - 95%.

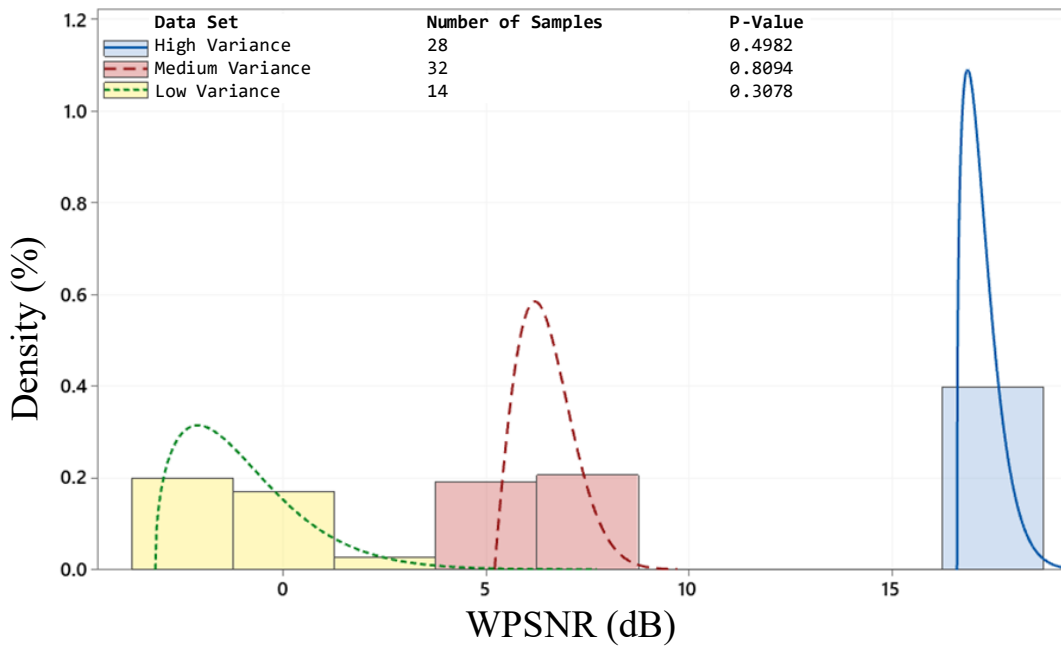


Figure 49. Histogram of quality Improvement distributions. Number of data points and P-value shown, between the truncation method in [7] and the proposed method. All distributions are 3-parameter Weibull distributions that fall within a 95% Confidence Interval.

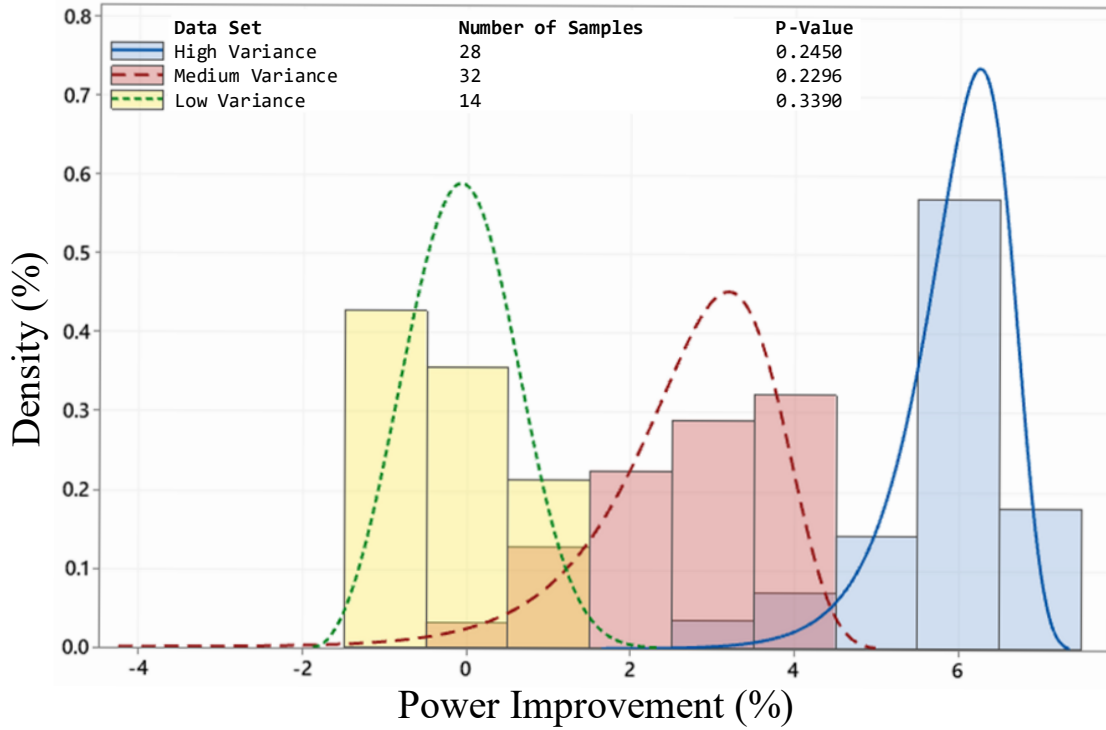


Figure 50. Histogram of power savings, measured in percentage improvement, between the truncation method in [7] and the proposed method. All distributions are 3-parameter Weibull distributions that fall within a 95% Confidence Interval.

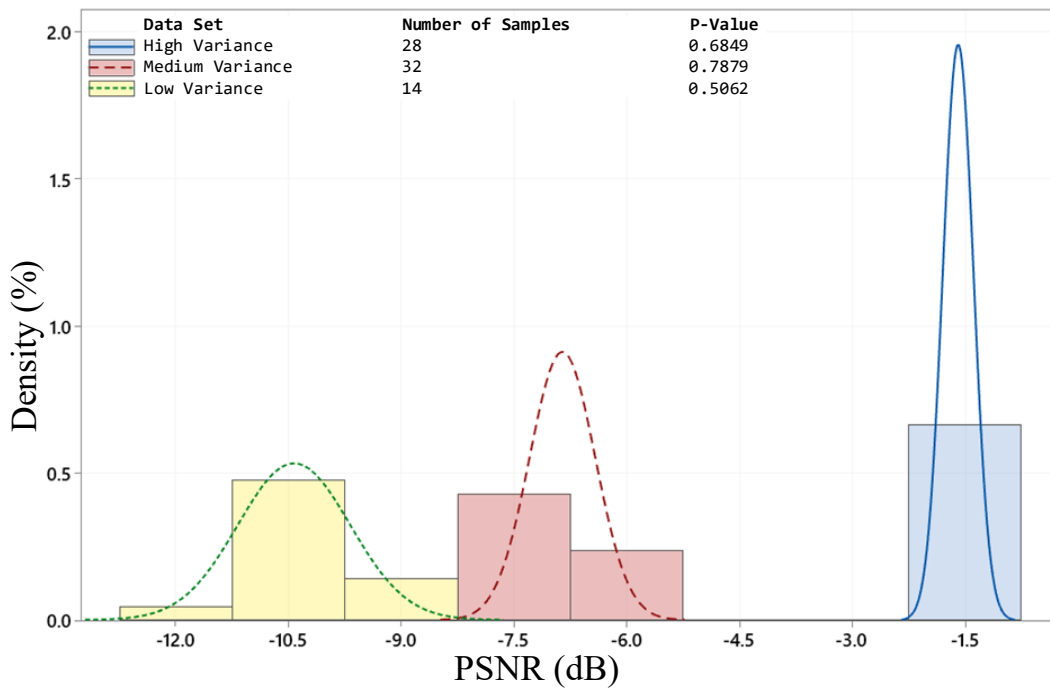


Figure 51. Histogram of PSNR noise increase, between the truncation method in [7] and the proposed method. All distributions are Normal Distributions that fall within a 95% Confidence Interval.

In the analysis, the null hypothesis for the Chi-Squared goodness-of-fit test, H_0 , is, “For the given set of video data points, a specified distribution accurately represents the data”, and therefore, the alternative hypothesis, H_1 , is, “For the given set of video data points, a specified distribution does not accurately represent the data.” Hence, the goal of the statistical analysis is to validate the null hypothesis and thus deduce that the specified distribution would fit the data. To achieve this statistical result, P-values were calculated for each data set – low, medium, and high variance – for WPSNR metrics, Power Savings, and video noise introduced. The Chi-Squared goodness-of-fit test can only be used for data put into classes (or bins); therefore, the data sets were placed into histograms: Figure 49 to Figure 51. The MathWave Technologies EasyFit software was used [34], to find the Chi-Squared goodness-of-fit test, in order to determine the type of distribution. In the video analysis results, the WPSNR metrics, power savings, and video noise introduced for all 74 videos were calculated for both the truncation method in [9] and the proposed method. As well, the data was split into three sets referred to as low, medium, and high variance, which corresponded to 1-bit, 2-bit, and 3-bit truncation videos using the truncation method in [7], respectively. These data are what is referred to as video data points in the statistical analysis.

Figure 49 demonstrates how categorizing the data creates clear groupings when comparing the truncation method in [7] to the proposed method. The figure shows three distinct 3-parameter Weibull distributions that describe the quality improvement between the proposed and [7]. These Weibull distributions are within the 95% confidence interval required. Each distribution reports a P-value greater than 0.1, implying that the null hypothesis is rejected and accept this distribution as a possible representation of the data. Figure 50 shows the power savings distribution for each video type as a 3-parameter Weibull distribution. Power savings is

reported as a percentage increase, using the total number of bits truncated in each video and the power consumption shown in Figure 47. All of these distributions pass the 95% confidence interval. Figure 51 shows the probability of noise increase in a random video stream. All distributions shown fall into the category of normal distributions with a 95% confidence interval.

It was determined that because all videos are compared to themselves for improvement, e.g. video after the proposed method is applied verses the original video, video resolution has no statistical impact in the data set. Power Consumption will be presented by improvement percentage, thus ignoring linear growth in watts saved in larger scale videos. Similarly, it is statistically sound that a larger dataset is not needed to affirm the distributions. As all distributions shown fall within the 95% confidence interval, there is only a 5% chance that the data collected is far from the specified distribution.

In summary, videos categorized as high variance show the biggest improvements in WPSNR quality, the most power saving by percentage, and introduce the least noise as measured by PSNR. With medium variance videos also saving on power consumption, with a more noticeable drop in quality and increase in noise. As such, videos classified as low variance often have little to gain using this method, and sometimes even cause video quality degradation.

CHAPTER 5. CONCLUSIONS AND FUTURE WORK

5.1. Chapter 2: Content-Adaptive Memory for Viewer-Aware Energy-Quality Scalable

Mobile Video Systems

In Chapter 2, a video context-aware memory technique was presented for energy-quality tradeoff using viewer's perspectives. Based on the influence of how video content characteristics impact the viewer's experience, developed were two simple, but effective models to enable hardware adaptation. Implemented was a new viewer-aware bit-truncation technique with minimized impact on the viewer's experience, while introducing energy-quality adaption to the video storage. Future investigations would include incorporating the motions of videos in the viewer's experience study as well as combining the viewing luminance awareness to further enable energy-quality adaption in different viewing surroundings.

During the hardware implementation process, a single percentage for the entire video was used in order to minimize the overhead of the design. In order to better suit the applicability and energy-quality scalability, future research could investigate the capability of calculating the macroblock percentage for each frame. This per frame calculation could allow for real-time adjustment of truncated bits at the cost of additional area overhead. Expanding the number of participants and video samples in order to create a more comprehensive model could also be used to improve the model results. Finally, further studying the relationship between the content information and the psychophysical human visual system models could be used to better understand what other metrics could be used to support the hardware design.

5.2. Chapter 3: Flexible Low Cost Power-Efficient Video Memory with ECC-Adaptation

In Chapter 3, presented is a flexible power-efficient video memory that dynamically adjusted the strength of error-correction-code (ECC), thereby enabling power-quality trade-offs

to achieve considerable power savings (up to 35.37%) without a noticeable degradation in video quality. To minimize the implementation overhead, the following two techniques have been developed: (i) a new parity storage scheme that utilizes the bit significance characteristics of video data for both ECC74 and ECC1511, and (ii) an integrated ECC encoder/decoder hardware design to support both ECC74 and ECC1511 that automatically shuts down part or all of the ECC circuitry when ECC74 or No ECC is selected, respectively. The proposed adaptive ECC method is also validated in hardware using a commercial SRAM chip, demonstrating significant supply voltage reduction without noticeable video quality degradation Table 9 compares this work against state-of-the art video memory designs. As shown, the proposed memory enables run-time quality adaptation without inducing bitcell area overhead and also did not require multiple supply voltages.

Since parity bit errors caused by memory failures resulted in the ECC decoder incorrectly flipping bits, which caused an increase in video quality degradation, future work will consider hardening specific bits, such as parity bits, to provide better video quality, with the trade-off being increased area overhead. Additionally, other multi-bit error correcting codes, besides ECC74 and ECC1511 used in this work, could also be considered. Furthermore, a mathematical/theoretical approach to determine specific cutoff values for switching between ECC methods could be investigated and compared to the experimental approach employed in this chapter.

5.3. Chapter 4: Content-Adaptable ROI-Aware Video Storage for Power-Quality Scalable Mobile Streaming

In Chapter 4, a video content-adaptable Region-of-Interest (ROI)-aware video storage technique is presented to optimize the power efficiency. The ROI of videos is identified and

protected to preserve the video quality, while other regions are truncated with 3-LSB truncation for power savings. To support the proposed method, a low-power frame buffer was developed that implemented 3-LSB truncation which enabled runtime quality and power adaptation. The results show that the proposed technique only uses 83.79% and 76.56% of the power on average for videos with ROI and without ROI respectively, as compared to the traditional memory and the state-of-the art [7], respectively. Meanwhile, the proposed technique can increase the quality (i.e. WPSNR values) by 20.17% on average for the videos with ROI and 26.46% additional truncated bits as compared to [7]. For the videos without ROI, the proposed technique can realize 44.61% additional truncated bits and 8.26% power savings as compared to [7], with a minimal quality loss (3 dB PSNR drop on average). This chapter focuses on the facial features as ROI of videos; future investigations would include extensions of ROI identification to deal with general videos. Additionally, psychological experiments will be conducted to access the visual experience of viewers for hardware optimization.

CHAPTER 6. REFERENCES

6.1. Chapter 2

- [1] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper. Accessed on Dec. 1, 2017. [Online]. Available: <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-chapter-c11-520862.html>
- [2] F. Sampaio, M. Shafique, B. Zatt, S. Bampi, and J. Henkel, “Energy-Efficient Architecture for Advanced Video Memory,” in Proc. 2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Nov. 2014, pp. 132-139.
- [3] T. Liu, T. Lin, S. Wang, W. Lee, J. Yang, K. Hou, and C. Lee, “A 125 uW, fully scalable MPEG-2 and H.264/AVC video decoder for mobile applications,” IEEE J. Solid-State Circuits, vol. 42, no. 1, pp. 161–169, Jan. 2007.
- [4] D. Zhou, S. Wang, H. Sun, J. Zhou, J. Zhu, Y. Zhao, J. Zhou, S. Zhang, S. Kimura, T. Yoshimura, S. Goto, “A 4Gpixel/s 8/10b H.265/HEVC Video Decoder Chip for 8K Ultra HD Applications,” in Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC), Feb. 2016, pp. 266-267.
- [5] M. Zhao, X. Gong, J. Liang, W. Wang, X. Que, and S. Cheng, “QoE-Driven Cross-Layer Optimization for Wireless Dynamic Adaptive Streaming of Scalable Videos Over HTTP,” IEEE Trans. on Circuits and Systems for Video Technology, vol. 25, no. 3, pp. 451-466, Mar. 2015.
- [6] D. Chen, X. Wang, J. Wang, and N. Gong, “VCAS: Viewing context aware power-efficient mobile video embedded memory,” in Proc. 2015 28th IEEE International System-on-Chip Conference (SOCC), Sept. 2015, pp. 333-338.

- [7] J. Edstrom, D. Chen, J. Wang, H. Gu, E. A. Vazquez, M. E. McCourt, and N. Gong, "Luminance-Adaptive Smart Video Storage System," in Proc. IEEE International Symposium on Circuits and Systems (ISCAS), 2016, pp. 734-737.
- [8] D. Chen, J. Edstrom, L. Yang, M. E. McCourt, J. Wang, and N. Gong, "Viewer-Aware Intelligent Efficient Mobile Video Embedded Memory," IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 26, no. 4, Apr. 2018, pp. 684-696.
- [9] O. Hirabayashi, A. Kawasumi, A. Suzuki, Y. Takeyama, K. Kushida, T. Sasaki, A. Katayama, G. Fukano, Y. Fujimura, T. Nakazato, Y. Shizuki, N. Kushiya, and T. Yabe, "A Process-Variation-Tolerant Dual-Power-Supply SRAM With 0.179 Cell in 40 nm CMOS Using Level-Programmable Wordline Driver," in Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC), Feb. 2009, pp. 458-459.
- [10] P. Wang, H. J. Liao, H. Yamauchi, Y. H. Chen, Y. L. Lin, S. H. Lin, D. C. Liu, H. C. Chang, and W. Hwang, "A 45 nm Dual-port SRAM with Write and Read Capability Enhancement at Low Voltage," in Proc. IEEE Int. SOC Conf., Sep. 2007, pp. 211-214.
- [11] F. Tachibana, O. Hirabayashi, Y. Takeyama, M. Shizuno, A. Kawasumi, K. Kushida, A. Suzuki, Y. Niki, S. Sasaki, T. Yabe, and Y. Unekawa, "A 27% Active and 85% Standby Power Reduction in Dual-Power-Supply SRAM Using BL Power Calculator and Digitally Controllable Retention Circuit," IEEE J. Solid-State Circuits, vol. 49, no. 1, pp. 118-126, Jan. 2014.
- [12] T.-H. Kim, J. Liu, and C. H. Kim, "A Voltage Scalable 0.26 V, 64 kb 8T SRAM with V_{min} Lowering Techniques and Deep Sleep Mode," IEEE J. Solid-State Circuits, vol. 44, no. 6, pp. 1785-1795, 2009.

- [13] M.-F. Chang, S.-W. Chang, P.-W. Chou, and W.-C. Wu, "A 130 mV SRAM with Expanded Write and Read Margins for Subthreshold Applications," *IEEE J. Solid-State Circuits*, vol. 46, no. 2, pp. 520-529, Feb. 2011.
- [14] H. Noguchi et al., "A 10T Non-precharge Two-port SRAM for 74% Power Reduction in Video Processing," in *Proc. IEEE Computer Society Annual Symp. VLSI Circuits*, Mar. 2007, pp. 107-112.
- [15] M. K. Qureshi and Z. Chishti, "Operating Secded-based Caches at Ultralow Voltage with Flair," in *Proc. 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp.1–11.
- [16] A. Ansari, S. Feng, S. Gupta, and S. A. Mahlke, "Archipelago: A Polymorphic Cache Design for Enabling Robust Near-threshold Operation," in *Proc. IEEE Symp. on High Performance Computer Architecture (HPCA)*, 2011, pp. 539–550.
- [17] I. Chang, D. Mohapatra, and K. Roy, "A priority-based 6T/8T hybrid SRAM architecture for aggressive voltage scaling in video applications," *IEEE Trans. on Circuits System for Video Technology*, vol. 21, no. 2, pp. 101-112, Feb. 2011.
- [18] J. Kwon, I. Lee, and J. Park, "Heterogeneous SRAM Cell Sizing for Low Power H.264 Applications," *IEEE Trans. on Circuits and Systems I*, vol. 99, no. 2, pp. 1-10, Feb. 2012.
- [19] N. Gong, S. Jiang, A. Challapalli, S. Fernandes, R. Sridhar, "Ultra-Low Voltage Split-data-aware Embedded SRAM for Mobile Video Applications," *IEEE Trans. on Circuits and Systems II* vol. 59, no. 12, pp. 883-887, Dec. 2012,
- [20] L. Kerofsky, R. Vanam, and Y. Reznik, "Adapting Objective Video Quality Metrics to Ambient lighting," in *Proc. Seventh International Workshop on Quality of Multimedia Experience (QoMEX)*, 2015, pp. 1-6.

- [21] F. Frustaci, D. Blaauw, D. Sylvester, and M. Alioto, "Approximate SRAMs With Dynamic Energy-Quality Management," IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 24, no. 6, pp. 2128-2141, Jun. 2016.
- [22] Youtube-8M Dataset. 2017. [Online]. Available:
<https://research.google.com/youtube8m/>
- [23] M. Shafique, S. Rehman, F. Kribel, M. U. K. Khan, B. Zatt, A. Subramaniyan, B. Vizzotoo, and J. Henkel, "Application-Guided Power-Efficient Fault Tolerance for H.264 Context Adaptive Variable Length Coding," IEEE Trans. on Computers, vol. 66, no. 4, pp. 560-574, April 2017.
- [24] M. Shafique, B. Molkenhain, and J. Henkel, "An HVS-based Adaptive Computational Complexity Reduction Scheme for H.264/AVC video encoder using Prognostic Early Mode Exclusion," in Proc. 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), 2010, pp. 1713-1718.
- [25] Xiph.org Video Test Media [derf's collection]. 2017. [Online]. Available:
<https://media.xiph.org/video/derf/>
- [26] Methodology for the subjective assessment of the quality of television pictures. 2012. [Online]. Available: https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.500-13-201201-I!!PDF-E.pdf
- [27] Recommendation ITU-R BT.500-13., [Online]. Available:
https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.500-13-201201-I!!PDF-E.pdf
- [28] FreePDK45. [Online]. Available:
<http://www.eda.ncsu.edu/wiki/FreePDK45:Contents>.

- [29] J. S. Wang, P. Y. Chang, T. S. Tang, J. W. Chen, and J. I. Guo, "Design of subthreshold SRAMs for energy-efficient quality-scalable video applications," *IEEE Trans. Emerging Sel. Topics Circuits Syst.*, vol. 1, no. 2, pp. 183-192, Jun. 2011.
- [30] Y. Feng, G. Cheung, W.-t. Tan, P. L. Callet, Y. Ji, "Low-Cost Eye Gaze Prediction System for Interactive Networked Video Streaming," *IEEE Trans. on Multimedia*, vol. 15, no. 8, pp. 1865-1879.
- [31] T. Zhao, Q. Liu, and C. W. Chen, "QoE in Video Transmission: A User Experience-Driven Strategy," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, 2017.
- [32] M. Ruggiero, A. Bartolini, and L. Benini, "DBS4video: Dynamic Luminance Backlight Scaling based on Multi-Histogram Frame Characterization for Video Streaming Application," in *Proc. ACM EMSOFT*, pp. 109-118, Oct. 2008.
- [33] C. Yim, A. C. Bovik, "Quality Assessment of Deblocked Images," *IEEE Trans. On Image Processing*, vol. 20, no. 1, Jan. 2011.
- [34] M. H. Pinson and S. Wolf, "A New Standardized Method for Objectively Measuring Video Quality," *IEEE Trans. On Broadcasting*, vol. 50, no. 3, pp. 312-322, Sep. 2004.
- [35] NTIA General Model (aka VQM) and Full Reference Calibration Standards. [Online]. Available: <https://www.its.bldrdoc.gov/resources/video-quality-research/standards/hidden-general-model.aspx>
- [36] Q. Bin, "Osen Logic OSD10 h.264 decoder," [Online]. Available: <http://bbs.eetop.cn/viewthread.php?tid=628991>. [Accessed 2018].
- [37] FFmpeg. [Online]. Available: <https://www.ffmpeg.org/>. [Accessed 2018].

6.2. Chapter 3

- [1] "Cisco Visual Networking Index: Forecast and Methodology, 2016-2021," Cisco Systems, Inc., [Online]. Available:
<https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html>.
- [2] J. S. Wang, P. Y. Chang, T. S. Tang, J. W. Chen and J. I. Guo, "Design of subthreshold SRAMs for energy-efficient quality-scalable video applications," *IEEE Trans. Emerging Sel. Topics Circuits and Systems*, vol. 1, no. 2, pp. 183-192, 2011.
- [3] J. Edstrom, Y. Gong, A. Haidous, B. Humphrey, M. McCourt, Y. Xu, J. Wang and N. Gong, "Content-Adaptive Memory for Viewer-Aware Energy-Quality Scalable Mobile Video Systems," *IEEE Access*, vol. 7, pp. 47479-47493, 2019.
- [4] F. Sampaio, M. Shafique, B. Zatt, S. Bampi and J. Henkel, "Energy-Efficient Architecture for Advanced Video Memory," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2014 .
- [5] I. Chang, D. Mohapatra and K. Roy, "A Priority-Based 6T/8T Hybrid SRAM Architecture for Aggressive Voltage Scaling in Video Applications," *IEEE. Trans. Circuits and Systems for Video Technology*, pp. 101-112, 2011.
- [6] J. Kwon, I. Lee and J. Park, "Heterogeneous SRAM Cell Sizing for Low Power H.264 Applications," *IEEE Trans. on Circuits and Systems I*, vol. 99, no. 2, pp. 1-10, 2012.
- [7] N. Gong, S. Jiang, A. Challapalli, S. Fernandes and R. Sridhar, "Ultra-Low Voltage Split-data-aware Embedded SRAM for Mobile Video Applications," *IEEE Trans. on Circuits and Systems II*, vol. 59, no. 12, pp. 883-887, 2012.

- [8] J. Edstrom, D. Chen, Y. Gong, J. Wang and N. Gong, "Data-Pattern Enabled Self-Recovery Low-Power Storage System for Big Video Data," *IEEE Trans. on Big Data*, vol. 51, no. 1, pp. 95-105, 2019.
- [9] D. Chen, J. Edstrom, Y. Gong, P. Gao, L. Yang, M. McCourt, J. Wang and N. Gong, "Viewer-Aware Intelligent Efficient Mobile Video Embedded Memory," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 4, pp. 684-696, 2018.
- [10] Y. Xu, H. Das, Y. Gong and N. Gong, "On Mathematical Models of Optimal Video Memory Design," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 1, pp. 256-266, 2020.
- [11] F. Frustaci, M. Khayatzadeh, D. Blaauw, D. Sylvester and M. Alioto, "SRAM for Error-Tolerant Applications With Dynamic Energy-Quality Management in 28nm CMOS," *IEEE J. Of Solid-State Circuits*, vol. 50, no. 5, pp. 1310-1323, 2015.
- [12] C. Duan, A. J. Gotterba, M. E. Sinangil and A. P. Chandrakasan, "Energy-Efficient Reconfigurable SRAM: Reducing Read Power Through Data Statistics," *IEEE Journal Of Solid-State Circuits*, vol. 52, no. 10, pp. 2703-2711, 2017.
- [13] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147-160, Apr. 1950.
- [14] "YouTube-8M Dataset.," 2017. [Online]. Available: <https://research.google.com/youtube8m/>.
- [15] F. Frustaci, D. Blaauw, D. Sylvester and M. Alioto, "Better-Than Voltage Scaling Energy Reduction in Approximate SRAMs Via Bit Dropping and Bit Reuse," in *25th Int. Workshop Power Timing Model., Optim.*, 2015.

- [16] Cypress, "Cypress CY62146GN Datasheet," 19 12 2017. [Online]. Available: <https://www.cypress.com/file/223376/download>. [Accessed 05 03 2020].
- [17] S. M. Jahinuzzaman, J. S. Shah, D. J. Rennie and M. Sachdev, "Design and analysis of A 5.3-pJ 64-kb gated ground SRAM with multiword ECC," *IEEE J. Solid-State Circuits*, p. 2543–2553, 2009.
- [18] Z. Chishti, A. R. Alameldeen, C. Wilkerson, W. Wu and S.-L. Lu, "Improving cache lifetime reliability at ultra-low voltages," in *42nd IEEE/ACM Int. Symp. Microarchit.*, 2009.

6.3. Chapter 4

- [1] [Online]. Available: <https://www.adcolony.com/blog/2019/03/05/video-on-track-to-be-nearly-80-of-mobile-data-traffic-by-2022/>.
- [2] T. Liu, S. Wang, W. Lee, J. Yang, K. Hou, Lee and C, "A 125 uW, fully scalable MPEG-2 and H.264/AVC video decoder for mobile applications," *IEEE J. Solid-State Circuits*, vol. 42, no. 1, p. 161–169, 2007.
- [3] F. Sampaio, M. Shafique, B. Zatt, S. Bampi and J. Henkel, "Energy-Efficient Architecture for Advanced Video Memory," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2014.
- [4] D. Chen, X. Wang, J. Wang and N. Gong, "VCAS: Viewing context aware power-efficient mobile video embedded memory," in *28th IEEE International System-on-Chip Conference (SOCC)*, Beijing, 2015.
- [5] D. Chen, J. Edstrom, L. Yang, M. E. McCourt, J. Wang and N. Gong, "Viewer-Aware Intelligent Efficient Mobile Video Embedded Memory," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, pp. 684-696, 2018.
- [6] J. Edstrom, D. Chen, J. Wang, H. Gu, E. A. Vazquez, M. E. McCourt and N. Gong, "Luminance-Adaptive Smart Video Storage System," in *International Symposium on Circuits and Systems (ISCAS)*, 2016.
- [7] J. Edstrom, Y. Gong, A. Haidous, B. Humphrey, M. E. McCourt, Y. Xu, J. Wang and N. Gong, "Content-Adaptive Memory for Viewer-Aware Energy-Quality Scalable Mobile Video Systems," *IEEE Access.*, vol. 7, pp. 47479-47493, 2019.

- [8] I. Chang, D. Mohapatra and K. Roy, "A Priority-Based 6T/8T Hybrid SRAM Architecture for Aggressive Voltage Scaling in Video Applications," *IEEE. Trans. Circuits and Systems for Video Technology*, pp. 101-112, 2011.
- [9] N. Gong, S. Jiang, A. Challapalli, S. Fernandes and R. Sridhar, "Ultra-Low Voltage Split-data-aware Embedded SRAM for Mobile Video Applications," *IEEE Trans. on Circuits and Systems II*, vol. 59, no. 12, pp. 883-887, 2012.
- [10] J. Kwon, I. Lee and J. Park, "Heterogeneous SRAM Cell Sizing for Low Power H.264 Applications," *IEEE Trans. on Circuits and Systems I*, vol. 99, no. 2, pp. 1-10, 2012.
- [11] F. Frustaci, M. Khayatzadeh, D. Blaauw, D. Sylvester and M. Alioto, "SRAM for Error-Tolerant Applications With Dynamic Energy-Quality Management in 28nm CMOS," *IEEE J. Of Solid-State Circuits*, vol. 50, no. 5, pp. 1310-1323, 2015.
- [12] A. Kazimirsky, A. Teman, N. Edri and A. Fish, "A 0.65-V, 500-MHz Integrated Dynamic and Static RAM for Error Tolerant Applications," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 9, pp. 2411-2418, 2017.
- [13] M.-C. Chi, C.-H. Yeh and M.-J. Chen, "Robust Region-of-Interest Determination Based on User Attention Model Through Visual Rhythm Analysis," *IEEE Trans. on Circuits and Systems on Vodeo Technology*, vol. 19, no. 7, pp. 1025-1038, 2009.
- [14] " YouTube-8M Dataset.," 2017. [Online]. Available: <https://research.google.com/youtube8m/> .
- [15] I. Himawan, W. Song and D. Tjondronegoro, "Automatic Region-of-Interest Detection and Prioritisation for Visually Optimised," in *IEEE Workshop on Applications of Computer Vision (WACV)*, 2013.

- [16] Y. Huo, X. Wang, P. Zhang, J. Jiang and L. Hanzo, "Unequal Error Protection Aided Region of Interest Aware Wireless Panoramic Video," *IEEE Access*, vol. 7, p. 2019, 80262-80276.
- [17] Y.-F. Ma, X.-S. Hua, L. Lu and H.-J. Zhang, "A generic framework of user attention model and its application in video summarization," *IEEE Trans. Multimedia*, vol. 7, no. 5, p. 907–919, 2005.
- [18] I. Culjak, D. Abram, T. Pribanic, H. Dzapo and M. Cifrek, "A brief introduction to OpenCV," in *35th International Convention MIPRO*, Opatija, 2012.
- [19] X.-W. Tang, X.-L. Huang, F. Hu and Q. Shi, "Human-Perception-Oriented Pseudo Analog Video Transmissions With Deep Learning," *IEEE Transactions on Vehicular Technology*, 2020.
- [20] M. Shafique, "Application-guided power-efficient fault tolerance for H.264 context adaptive variable length coding," *IEEE Trans. Comput.*, vol. 66, no. 4, pp. 560-574, 2017.
- [21] M. Shafique, B. Molkenthin and J. Henkel, "An HVS-based adaptive computational complexity reduction scheme for H.264/AVC video encoder using prognostic early mode exclusion," in *Design, Automation & Test in Europe Conference & Exhibition*, 2010.
- [22] Y. Liang, H. Wang and K. El-Maleh, "Design and implementation of content-adaptive background skipping for wireless video," *IEEE International Symposium on Circuits and Systems*, pp. 4-7, 2006.
- [23] R. P. Foundation, "Raspberry Pi Documentation," [Online]. Available: <https://www.raspberrypi.org/documentation/>.

- [24] Xilinx, "Z-turn Board (with Zynq-7020)," [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/1-571ww1.html>. [Accessed 01 11 2020].
- [25] MAGEWELL, "USB Capture Utility V3," [Online]. Available: <http://www.magewell.com/usb-capture-utility-v3>. [Accessed 01 11 2020].
- [26] Ylonen, T. Rinne and Tatu, "scp(1) - Linux man page," 14 04 2013. [Online]. Available: <https://linux.die.net/man/1/scp>. [Accessed 01 11 2020].
- [27] Q. Bin, "Osen Logic OSD10 h.264 decoder," [Online]. Available: <http://bbs.eetop.cn/viewthread.php?tid=628991>. [Accessed 2018].
- [28] I. E. Richardson, The H.264 Advanced Video Compression Standard (Second Edition), West Sussex, UK: John Wiley & Sons, Ltd, 2010.
- [29] Y. Wang, S. Inguva and B. Adsumilli, "YouTube UGC Dataset for Video Compression Research," in 2019 IEEE 21st International Workshop on Multimedia Signal Processing (MMSP), Kuala Lumpur, Malaysia, 2019.
- [30] "Xiph.org Video Test Media [derf's collection]," Xiph, [Online]. Available: <https://media.xiph.org/video/derf/>. [Accessed 18 October 2020].
- [31] J. Erfurt, C. R. Helmrich, S. Bosse, H. Schwarz, D. Marpe and T. Wiegand, "A Study of the Perceptually Weighted Peak Signal-To-Noise Ratio (WPSNR) for Image Compression," in 2019 IEEE International Conference on Image Processing (ICIP), Taipei, Taiwan, 2019 .
- [32] Xilinx, "Vivado Design Suite," 2019. [Online]. Available: <https://www.xilinx.com/products/design-tools/vivado.html>.

- [33] K. Pearson, "Chapter 56 - Karl Pearson, paper on the chi square goodness of fit test (1900)," in *Landmark Writings in Western Mathematics 1640-1940*, ELSEVIER, 2005, pp. 724-731.
- [34] MathWave Technologies, EasyFit Software, 2015.
- [35] F. Frustaci, D. Blaauw, D. Sylvester and M. Alioto, "Better-Than Voltage Scaling Energy Reduction in Approximate SRAMs Via Bit Dropping and Bit Reuse," in *25th Int. Workshop Power Timing Model., Optim.*, 2015.
- [36] C. Duan, A. J. Gotterba, M. E. Sinangil and A. P. Chandrakasan, "Energy-Efficient Reconfigurable SRAM: Reducing Read Power Through Data Statistics," *IEEE Journal Of Solid-State Circuitis*, vol. 52, no. 10, pp. 2703-2711, 2017.
- [37] J. Edstrom, D. Chen, Y. Gong, J. Wang and N. Gong, "Data-Pattern Enabled Self-Recovery Low-Power Storage System for Big Video Data," *IEEE Trans. on Big Data*, vol. 51, no. 1, pp. 95-105, 2019.
- [38] "Cisco Visual Networking Index: Forecast and Methodology, 2016-2021," Cisco Systems, Inc., [Online]. Available:
<https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html>.
- [39] J. S. Wang, P. Y. Chang, T. S. Tang, J. W. Chen and J. I. Guo, "Design of subthreshold SRAMs for energy-efficient quality-scalable video applications," *IEEE Trans. Emerging Sel. Topics Circuits and Systems*, vol. 1, no. 2, pp. 183-192, 2011.
- [40] Cypress, "Cypress CY62146GN Datasheet," 19 12 2017. [Online]. Available:
<https://www.cypress.com/file/223376/download>. [Accessed 05 03 2020].

- [41] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, Apr. 1950..
- [42] Y. Xu, H. Das, Y. Gong and N. Gong, "On Mathematical Models of Optimal Video Memory Design," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 1, pp. 256-266, 2020.
- [43] M. E. Sinangil and A. P. Chandrakasan, "Application-Specific SRAM Design Using Output Prediction to Reduce Bit-Line Switching Activity and Statistically Gated Sense Amplifiers for Up to 1.9 Lower Energy/Access," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 1, pp. 107-117, 2014.
- [44] Sampaio, F., M. Shafique, B. Zatt, S. Bampi and J. Henkel, "Energy-Efficient Architecture for Advanced Video Memory," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2014.
- [45] P. Pokorny, "Lossy Compression in the Chroma Subsampling Process," Tomas Bata University in Zlín Nad Stráněmi, 4511, 760 05 Zlín Czech Republic, 2016.
- [46] A. Inc, "Youtube," [Online]. Available: <https://www.youtube.com/>. [Accessed 01 11 2020].
- [47] L. Liu, W. Ouyang and X. Wang, "Deep Learning for Generic Object Detection: A Survey," *Int J Comput Vis*, vol. 128, pp. 261-318, 2019.
- [48] M. Chen, S. Mao and Y. Liu, "Big Data: A Survey.," *Mobile Netw Appl*, vol. 19, p. 171–209, 2014.
- [49] Z. Wang, A. C. Bovik, H. R. Sheikh, Simoncelli and E. P., "Image quality assessment: from error visibility to structural similarity," *IEEE Trans. on Image Processing*, vol. 13, no. 4, pp. 600-612, 2004.

APPENDIX A. MACROBLOCK VARIANCE TRUNCATION

Description: This program takes as an input a RAW YUV 4:2:0 H264 decoded format video and analyzes each frame for Macroblock variance percentage. The outputs are a .csv file of the analysis results as well as selected frames bit-truncated using proprietary techniques used in Ali Haidous's PhD dissertation. Written by: Ali Ahmad Haidous.

```
#!/usr/bin/python

import struct
import sys
import math
import pickle
import cv2
import numpy as np
from tqdm import tqdm
import os
from skimage.measure import compare_ssim

FIRST_PLAIN_MB = 21.5571
SECOND_PLAIN_MB = 1.96405

#helpful links
#https://docs.opencv.org/master/d8/d01/group_imgproc_color_conversions.html #all cv2 color codes

class VideoCaptureYUV(object):
    def __init__(self, filename, size):
        self.height, self.width = size
        self.filename = filename
        self.filesize = os.stat(filename).st_size
        self.framecount = ( 2 * self.filesize ) / ( self.height * self.width * 3 )
        self.frame_len = self.width * self.height * 3 / 2
        self.f = open(filename, 'rb')
        self.shape = (int(self.height*1.5), self.width)

    def file_statistics(self):
        return (self.filesize, self.framecount)

    def read_raw(self):
        try:
            raw = self.f.read(self.frame_len)
            yuv = np.frombuffer(raw, dtype=np.uint8)
            yuv = yuv.reshape(self.shape)
        except Exception as e:
            print str(e)
            return False, None
        return True, yuv

    def read(self):
        ret, yuv = self.read_raw()
        if not ret:
            return ret, yuv
        bgr = cv2.cvtColor(yuv, cv2.COLOR_YUV2BGR_I420)
        return ret, bgr

    def fetch_raw_frame(self, frame_num):
        f = open(self.filename, 'rb')
        raw = None
        for _ in range(frame_num):
            raw = f.read(self.frame_len)
        try:
```

```

        yuv = np.frombuffer(raw, dtype=np.uint8)
        yuv = yuv.reshape(self.shape)
    except Exception as e:
        print str(e)
        return None
    return yuv

def display_raw_frame(self, raw_frame, name="frame"):
    frame = cv2.cvtColor(raw_frame, cv2.COLOR_YUV2BGR_I420)
    #cv2.imshow(name, frame)
    cv2.imwrite(name, frame)

def play_video(self):
    while True:
        ret, frame = self.read(cv2.COLOR_YUV2BGR_I420)
        if ret:
            cv2.imshow("frame", frame)
            cv2.waitKey(30)
        else:
            break

def calc_macroblock_per(yuv_frame, low_variance_threshold=1.25):
    offset = int(len(yuv_frame)/3)
    rows = len(yuv_frame) - offset
    columns = len(yuv_frame[0])

    y = yuv_frame[0:rows, 0:columns]
    u = yuv_frame[rows:rows+(offset/2), 0:columns]
    v = yuv_frame[rows+(offset/2):rows+offset, 0:columns]

    def macroblock_per(sub_frame):
        total_macroblocks = 0
        plain_macroblocks = 0
        macroblock_per = 0
        for row_position in range(0, len(sub_frame), 16):
            for column_position in range(0, len(sub_frame[0]), 16):
                macroblock = []
                total_macroblocks += 1

                for j in range(row_position, row_position+16):
                    for i in range(column_position, column_position+16):
                        try:
                            macroblock.append(0.0001560911143834408 * pow(int(sub_frame[j][i]),
2.628389343175764))
                        except IndexError:
                            break

                try:
                    avg_lum = sum(macroblock) / len(macroblock)
                    variance = sum([pow(byte - avg_lum, 2) / len(macroblock) for byte in
macroblock])
                except ZeroDivisionError:
                    pass
                else:
                    if variance <= low_variance_threshold:
                        plain_macroblocks += 1

            macroblock_per = ((plain_macroblocks * 100.0) / total_macroblocks)
            return macroblock_per

    return (macroblock_per(y), macroblock_per(u), macroblock_per(v))

def bits_to_truncate(macroblock_per):
    if macroblock_per >= FIRST_PLAIN_MB:
        return 1
    elif macroblock_per >= SECOND_PLAIN_MB:
        return 2
    else:
        return 3

def truncate_frame(yuv_frame, y_bits, u_bits, v_bits):

```

```

offset = int(len(yuv_frame)/3)
rows = len(yuv_frame) - offset
columns = len(yuv_frame[0])

y = yuv_frame[0:rows, 0:columns]
u = yuv_frame[rows:rows+(offset/2), 0:columns]
v = yuv_frame[rows+(offset/2):rows+offset, 0:columns]

yuv_frame_copy = np.copy(yuv_frame)
for row in range(0, rows):
    for column in range(0, columns):
        yuv_frame_copy[row, column] = yuv_frame[row, column] & ((0xFF >> y_bits) << y_bits)
for row in range(rows, rows+(offset/2)):
    for column in range(0, columns):
        yuv_frame_copy[row, column] = yuv_frame[row, column] & ((0xFF >> u_bits) << u_bits)
for row in range(rows+(offset/2), rows+offset):
    for column in range(0, columns):
        yuv_frame_copy[row, column] = yuv_frame[row, column] & ((0xFF >> v_bits) << v_bits)

return yuv_frame_copy

def psnr(img1, img2):
    mse = np.mean( (img1 - img2) ** 2 )
    if mse == 0:
        return 100
    PIXEL_MAX = 255.0
    return 20 * math.log10(PIXEL_MAX / math.sqrt(mse))

def main():
    # Get arguments
    filename = sys.argv[1]
    xres = int(sys.argv[2])
    yres = int(sys.argv[3])

    # Do OS operations
    path = os.path.join(os.getcwd(), str(filename[:-4].rsplit('/', 1)[-1]))
    frames_path = os.path.join(os.getcwd(), str(filename[:-4].rsplit('/', 1)[-1])+"\\frames\\")
    try:
        os.makedirs(path)
        os.makedirs(frames_path)
    except OSError:
        print ("Creation of the directory %s failed" % frames_path)
    else:
        print ("Successfully created the directory %s " % frames_path)

    # Read in the file
    cap = VideoCaptureYUV(filename, (yres, xres))
    filesize, framecount = cap.file_statistics()
    print "Size of file in bytes: %d\nNumber of frames: %d\n" % (filesize, framecount)
    #cap.play_video()

    # Calculate macroblock percentage per frame
    macroblock_per_y = []
    macroblock_per_u = []
    macroblock_per_v = []
    for _ in tqdm(range(framecount), unit="Frame"):
        try:
            ret, frame = cap.read_raw()
        except Exception:
            break
        if ret:
            (y_per, u_per, v_per) = calc_macroblock_per(frame)
            print (y_per, u_per, v_per)
            macroblock_per_y.append(y_per)
            macroblock_per_u.append(u_per)
            macroblock_per_v.append(v_per)
        else:
            break

    # Calculate average macroblock percentages

```

```

macroblock_per_y_avg = sum(macroblock_per_y) / len(macroblock_per_y)
macroblock_per_u_avg = sum(macroblock_per_u) / len(macroblock_per_u)
macroblock_per_v_avg = sum(macroblock_per_v) / len(macroblock_per_v)

# Filter >= 99% macroblocks
for x in range(len(macroblock_per_y)):
    if macroblock_per_y[x] >= 99:
        macroblock_per_y[x] = macroblock_per_y_avg
for x in range(len(macroblock_per_u)):
    if macroblock_per_u[x] >= 99:
        macroblock_per_u[x] = macroblock_per_u_avg
for x in range(len(macroblock_per_v)):
    if macroblock_per_v[x] >= 99:
        macroblock_per_v[x] = macroblock_per_v_avg

# Calculate amount of bits truncated with old vs new method
total_bits = framecount * xres * yres * 1.5 * 8
old_method = framecount * bits_to_truncate(macroblock_per_y_avg) * xres * yres
new_method = (sum([bits_to_truncate(per) for per in macroblock_per_y]) * xres * yres +
              sum([bits_to_truncate(per) for per in macroblock_per_u]) * xres * yres * 0.25 +
              sum([bits_to_truncate(per) for per in macroblock_per_v]) * xres * yres * 0.25)

# Calculate average psnr and ssim for the whole video
cap = VideoCaptureYUV(filename, (yres, xres))
psnr_old = []
ssim_old = []
psnr_new = []
ssim_new = []
for index in tqdm(range(framecount), unit="Frame"):
    try:
        ret, frame = cap.read_raw()
    except Exception:
        break
    if ret:
        frame_truncate_old = truncate_frame(frame,
                                             bits_to_truncate(macroblock_per_y_avg),
                                             0,
                                             0)
        frame_truncate_new = truncate_frame(frame,
                                             bits_to_truncate(macroblock_per_y[index]),
                                             bits_to_truncate(macroblock_per_u[index]),
                                             bits_to_truncate(macroblock_per_v[index]))
        psnr_old.append(psnr(frame, frame_truncate_old))
        (ssim, _) = compare_ssim(frame, frame_truncate_old, full=True)
        ssim_old.append(ssim)
        psnr_new.append(psnr(frame, frame_truncate_new))
        (ssim, _) = compare_ssim(frame, frame_truncate_new, full=True)
        ssim_new.append(ssim)
        cap.display_raw_frame(frame, frames_path+str(filename[:-4].rsplit('/', 1)[-1])+
                               "_frame"+str(index)+".png")
        cap.display_raw_frame(frame_truncate_old, frames_path+str(filename[:-4].rsplit('/',
1)[-1])+ "_truncate_old"+str(index)+".png")
        cap.display_raw_frame(frame_truncate_new, frames_path+str(filename[:-4].rsplit('/',
1)[-1])+ "_truncate_new"+str(index)+".png")
    else:
        break

# Calculate max and min macroblock frame index for y u and v
max_frame_y_index = macroblock_per_y.index(max(macroblock_per_y))
min_frame_y_index = macroblock_per_y.index(min(macroblock_per_y))
max_frame_u_index = macroblock_per_u.index(max(macroblock_per_u))
min_frame_u_index = macroblock_per_u.index(min(macroblock_per_u))
max_frame_v_index = macroblock_per_v.index(max(macroblock_per_v))
min_frame_v_index = macroblock_per_v.index(min(macroblock_per_v))

#Calculate average PSNR and SSIM
psnr_old_avg = sum(psnr_old) / len(psnr_old)
ssim_old_avg = sum(ssim_old) / len(ssim_old)
psnr_new_avg = sum(psnr_new) / len(psnr_new)
ssim_new_avg = sum(ssim_new) / len(ssim_new)

```

```

# Write data to CSV file
with open(os.path.join(path, str(filename[:-4].rsplit('/', 1)[-1])+".csv"), "wb") as file:
    file.write("Total Bits:,"+
               "Old Total:,"+
               "New Total:,"+
               "MB % Y Avg:,"+
               "MB % U Avg:,"+
               "MB % V Avg:,"+
               "MB Y Max Idx:,"+
               "MB Y Min Idx:,"+
               "MB U Max Idx:,"+
               "MB U Min Idx:,"+
               "MB V Max Idx:,"+
               "MB V Min Idx:,"+
               "PSNR Old Avg:,"+
               "SSIM Old Avg:,"+
               "PSNR New Avg:,"+
               "SSIM New Avg:,"+"\n")
    file.write(str(total_bits)+","+"
               str(old_method)+","+"
               str(new_method)+","+"
               str(macroblock_per_y_avg)+","+"
               str(macroblock_per_u_avg)+","+"
               str(macroblock_per_v_avg)+","+"
               str(max_frame_y_index)+","+"
               str(min_frame_y_index)+","+"
               str(max_frame_u_index)+","+"
               str(min_frame_u_index)+","+"
               str(max_frame_v_index)+","+"
               str(min_frame_v_index)+","+"
               str(psnr_old_avg)+","+"
               str(ssim_old_avg)+","+"
               str(psnr_new_avg)+","+"
               str(ssim_new_avg)+"\n\n\n")
    file.write("Frame,MB % Y,MB % U,MB % V,PSNR Old,SSIM Old,PSNR New,SSIM New\n")
    for index, mb_y, mb_u, mb_v, psnr_o, ssim_o, psnr_n, ssim_n in zip(range(framecount),
                                                                    macroblock_per_y,
                                                                    macroblock_per_u,
                                                                    macroblock_per_v,
                                                                    psnr_old,
                                                                    ssim_old,
                                                                    psnr_new,
                                                                    ssim_new):
file.write(str(index)+","+"str(mb_y)+","+"str(mb_u)+","+"str(mb_v)+","+"str(psnr_o)+","+"str(ssim_o)+",
"+"str(psnr_n)+","+"str(ssim_n)+"\n")

if __name__ == "__main__":
    main()

```


APPENDIX B. ECC 74 AND ECC 1511 ANALYZER

Description: This program takes as an input a RAW YUV 4:2:0 H264 decoded format video and simulates ECC errors along a Normal Uniform Distribution for both 16-bit and 32-bit ECC. The outputs are a .csv file of the analysis results as well as the original frames, error frames, and the ECC corrected frames for both ECC74 and ECC1511. Written by: Ali Ahmad Haidous.

```
#!/C:/Python27/python.exe

import numpy as np
import os
import math
import csv
import random
import multiprocessing
import time
import shutil
import re

THREAD_COUNT = 8 # Specify the amount of threads on your system

YUV_FRAMES = ".\\yuv_frames" # Specify where all the YUV frames to process are located
X_RESOLUTION = 320 # Specify the X resolution of the YUV frames
Y_RESOLUTION = 240 # Specify the Y resolution of the YUV frames

# Error injection percentages, ERROR LOW and ERROR HIGH numbers are divided by 10000
ERROR_LOW = 1
ERROR_HIGH = 100
ERROR_INCREMENT = 1

#####
##
## Quality measurements
#####
##

def psnr(img1, img2):
    img1 = img1.astype(np.float64) / 255.
    img2 = img2.astype(np.float64) / 255.
    mse = np.mean((img1 - img2) ** 2)
    if mse == 0:
        return "Same Image"
    return 10 * math.log10(1. / mse)

#####
##

#####
##
## Bit Operations
#####
##

def getBit(byte, position):
    return (byte >> position) & 0x01

def toggleKthBit(n, k):
    return (n ^ (1 << (k)))
```

```

def injectError(bytes, err_per=0):
    bytes_out = bytes

    # pick a random position to inject an error
    position = int(round(np.random.uniform(low=0.0, high=float((len(bytes)*8)-1), size=None)))
    for _ in range(len(bytes)*8): # interate over all positions
        if position < (len(bytes)*8)-1:
            position += 1
        else:
            position -= (len(bytes)*8)

    # if the error threshold is met, inject an error at the appropriate bit position
    if np.random.uniform(low=0.0, high=1.0, size=None) <= err_per:
        bytes_out[int(position/8)] = toggleKthBit(bytes[int(position/8)],
            position-(int(position/8)*8))

    return bytes_out

def injectSameError(original_byte, error_byte, encoded_byte):
    output_byte = encoded_byte
    for x in range(8):
        if ((original_byte >> x) & 0x01) != ((error_byte >> x) & 0x01):
            output_byte = toggleKthBit(output_byte, x)
    return output_byte

#####
##

#####
##
## ECC Encoder
#####
##

def getP1(bytes, algorithm):
    P1 = 0
    if algorithm == 74:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            P1 = ((getBit(byte1, 7) ^
                getBit(byte2, 15-8) ^
                getBit(byte2, 14-8)) & 0x01)
        elif len(bytes) == 4:
            byte1, byte2, byte3, byte4 = bytes
            P1 = ((getBit(byte1, 7) ^
                getBit(byte1, 15-8) ^
                getBit(byte4, 31-24)) & 0x01)
    elif algorithm == 1511:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            P1 = ((getBit(byte1, 7) ^
                getBit(byte2, 15-8) ^
                getBit(byte2, 14-8) ^
                getBit(byte1, 5) ^
                getBit(byte1, 4) ^
                getBit(byte1, 3) ^
                getBit(byte1, 2)) & 0x01)
        elif len(bytes) == 4:
            byte1, byte2, byte3, byte4 = bytes
            P1 = ((getBit(byte1, 7) ^
                getBit(byte2, 15-8) ^
                getBit(byte4, 31-24) ^
                getBit(byte1, 6) ^
                getBit(byte3, 22-16) ^
                getBit(byte1, 5) ^
                getBit(byte3, 21-16)) & 0x01)

    return P1

def getP2(bytes, algorithm):
    P2 = 0
    if algorithm == 74:

```

```

    if len(bytes) == 2:
        byte1, byte2 = bytes
        P2 = ((getBit(byte1, 7) ^
                getBit(byte1, 6) ^
                getBit(byte2, 14-8)) & 0x01)
    elif len(bytes) == 4:
        byte1, byte2, byte3, byte4 = bytes
        P2 = ((getBit(byte1, 7) ^
                getBit(byte3, 23-16) ^
                getBit(byte4, 31-24)) & 0x01)
elif algorithm == 1511:
    if len(bytes) == 2:
        byte1, byte2 = bytes
        P2 = ((getBit(byte1, 7) ^
                getBit(byte1, 6) ^
                getBit(byte2, 14-8) ^
                getBit(byte2, 13-8) ^
                getBit(byte1, 4) ^
                getBit(byte2, 11-8) ^
                getBit(byte1, 2)) & 0x01)
    elif len(bytes) == 4:
        byte1, byte2, byte3, byte4 = bytes
        P2 = ((getBit(byte1, 7) ^
                getBit(byte3, 23-16) ^
                getBit(byte4, 31-24) ^
                getBit(byte2, 14-8) ^
                getBit(byte3, 22-16) ^
                getBit(byte2, 13-8) ^
                getBit(byte3, 21-16)) & 0x01)

return P2

def getP3(bytes, algorithm):
    P3 = 0
    if algorithm == 74:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            P3 = ((getBit(byte2, 15-8) ^
                    getBit(byte1, 6) ^
                    getBit(byte2, 14-8)) & 0x01)
        elif len(bytes) == 4:
            byte1, byte2, byte3, byte4 = bytes
            P3 = ((getBit(byte2, 15-8) ^
                    getBit(byte3, 23-16) ^
                    getBit(byte4, 31-24)) & 0x01)
    elif algorithm == 1511:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            P3 = ((getBit(byte2, 15-8) ^
                    getBit(byte1, 6) ^
                    getBit(byte2, 14-8) ^
                    getBit(byte2, 12-8) ^
                    getBit(byte1, 3) ^
                    getBit(byte2, 11-8) ^
                    getBit(byte1, 2)) & 0x01)
        elif len(bytes) == 4:
            byte1, byte2, byte3, byte4 = bytes
            P3 = ((getBit(byte2, 15-8) ^
                    getBit(byte3, 23-16) ^
                    getBit(byte4, 31-24) ^
                    getBit(byte4, 30-24) ^
                    getBit(byte1, 5) ^
                    getBit(byte2, 13-8) ^
                    getBit(byte3, 21-16)) & 0x01)

return P3

def getP4(bytes, algorithm):
    P4 = 0
    if algorithm == 1511:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            P4 = ((getBit(byte1, 5) ^

```

```

        getBit(byte2, 13-8) ^
        getBit(byte1, 4) ^
        getBit(byte2, 12-8) ^
        getBit(byte1, 3) ^
        getBit(byte2, 11-8) ^
        getBit(byte1, 2) & 0x01)
    elif len(bytes) == 4:
        byte1, byte2, byte3, byte4 = bytes
        P4 = ((getBit(byte1, 6) ^
              getBit(byte2, 14-8) ^
              getBit(byte3, 22-16) ^
              getBit(byte4, 30-24) ^
              getBit(byte1, 5) ^
              getBit(byte2, 13-8) ^
              getBit(byte3, 21-16)) & 0x01)
    return P4

def getByte1(bytes, algorithm):
    if algorithm == 74:
        if len(bytes) == 2:
            return (bytes[0] & 0xFC) | (getP2(bytes, algorithm) << 1) | getP1(bytes, algorithm)
        elif len(bytes) == 4:
            return (bytes[0] & 0xFE) | getP1(bytes, algorithm)
    elif algorithm == 1511:
        if len(bytes) == 2:
            return (bytes[0] & 0xFC) | (getP2(bytes, algorithm) << 1) | getP1(bytes, algorithm)
        elif len(bytes) == 4:
            return (bytes[0] & 0xFE) | getP1(bytes, algorithm)
    else:
        return bytes[0]

def getByte2(bytes, algorithm):
    if algorithm == 74:
        if len(bytes) == 2:
            return (bytes[1] & 0xFE) | getP3(bytes, algorithm)
        elif len(bytes) == 4:
            return (bytes[1] & 0xFE) | getP2(bytes, algorithm)
    elif algorithm == 1511:
        if len(bytes) == 2:
            return (bytes[1] & 0xFC) | (getP4(bytes, algorithm) << 1) | getP3(bytes, algorithm)
        elif len(bytes) == 4:
            return (bytes[1] & 0xFE) | getP2(bytes, algorithm)
    else:
        return bytes[1]

def getByte3(bytes, algorithm):
    if algorithm == 74:
        return (bytes[2] & 0xFE) | getP3(bytes, algorithm)
    elif algorithm == 1511:
        return (bytes[2] & 0xFE) | getP3(bytes, algorithm)
    else:
        return bytes[2]

def getByte4(bytes, algorithm):
    if algorithm == 74:
        return bytes[3]
    elif algorithm == 1511:
        return (bytes[3] & 0xFE) | getP4(bytes, algorithm)
    else:
        return bytes[3]

def ecc_encode(bytes, algorithm):
    encoded_array = bytes
    if len(bytes) == 2:
        encoded_array[0] = (getByte1(bytes, algorithm))
        encoded_array[1] = (getByte2(bytes, algorithm))
    elif len(bytes) == 4:
        encoded_array[0] = (getByte1(bytes, algorithm))
        encoded_array[1] = (getByte2(bytes, algorithm))
        encoded_array[2] = (getByte3(bytes, algorithm))
        encoded_array[3] = (getByte4(bytes, algorithm))

```

```

    return encoded_array

#####
##

#####
##
## ECC Decoder
#####
##

def getE1(bytes, algorithm):
    E1 = 0
    if algorithm == 74:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            E1 = ((getBit(byte1, 0) ^
                    getBit(byte1, 7) ^
                    getBit(byte2, 7) ^
                    getBit(byte2, 6)) & 0x01)
        elif len(bytes) == 4:
            byte1, byte2, byte3, byte4 = bytes
            E1 = ((getBit(byte1, 0) ^
                    getBit(byte1, 7) ^
                    getBit(byte2, 15-8) ^
                    getBit(byte4, 31-24)) & 0x01)
    elif algorithm == 1511:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            E1 = ((getBit(byte1, 0) ^
                    getBit(byte1, 7) ^
                    getBit(byte2, 7) ^
                    getBit(byte2, 6) ^
                    getBit(byte1, 5) ^
                    getBit(byte1, 4) ^
                    getBit(byte1, 3) ^
                    getBit(byte1, 2)) & 0x01)
        elif len(bytes) == 4:
            byte1, byte2, byte3, byte4 = bytes
            E1 = ((getBit(byte1, 0) ^
                    getBit(byte1, 7) ^
                    getBit(byte2, 15-8) ^
                    getBit(byte4, 31-24) ^
                    getBit(byte1, 6) ^
                    getBit(byte3, 22-16) ^
                    getBit(byte1, 5) ^
                    getBit(byte3, 21-16)) & 0x01)

    return E1

def getE2(bytes, algorithm):
    E2 = 0
    if algorithm == 74:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            E2 = ((getBit(byte1, 1) ^
                    getBit(byte1, 7) ^
                    getBit(byte1, 6) ^
                    getBit(byte2, 6)) & 0x01)
        elif len(bytes) == 4:
            byte1, byte2, byte3, byte4 = bytes
            E2 = ((getBit(byte2, 8-8) ^
                    getBit(byte1, 7) ^
                    getBit(byte3, 23-16) ^
                    getBit(byte4, 31-24)) & 0x01)
    elif algorithm == 1511:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            E2 = ((getBit(byte1, 1) ^
                    getBit(byte1, 7) ^
                    getBit(byte1, 6) ^

```

```

        getBit(byte2, 6) ^
        getBit(byte2, 5) ^
        getBit(byte1, 4) ^
        getBit(byte2, 3) ^
        getBit(byte1, 2) & 0x01)
    elif len(bytes) == 4:
        byte1, byte2, byte3, byte4 = bytes
        E2 = ((getBit(byte2, 8-8) ^
            getBit(byte1, 7) ^
            getBit(byte3, 23-16) ^
            getBit(byte4, 31-24) ^
            getBit(byte2, 14-8) ^
            getBit(byte3, 22-16) ^
            getBit(byte2, 13-8) ^
            getBit(byte3, 21-16)) & 0x01)

    return E2

def getE3(bytes, algorithm):
    E3 = 0
    if algorithm == 74:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            E3 = ((getBit(byte2, 0) ^
                getBit(byte2, 7) ^
                getBit(byte1, 6) ^
                getBit(byte2, 6)) & 0x01)
        elif len(bytes) == 4:
            byte1, byte2, byte3, byte4 = bytes
            E3 = ((getBit(byte3, 16-16) ^
                getBit(byte2, 15-8) ^
                getBit(byte3, 23-16) ^
                getBit(byte4, 31-24)) & 0x01)
    elif algorithm == 1511:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            E3 = ((getBit(byte2, 0) ^
                getBit(byte2, 7) ^
                getBit(byte1, 6) ^
                getBit(byte2, 6) ^
                getBit(byte2, 4) ^
                getBit(byte1, 3) ^
                getBit(byte2, 3) ^
                getBit(byte1, 2)) & 0x01)
        elif len(bytes) == 4:
            byte1, byte2, byte3, byte4 = bytes
            E3 = ((getBit(byte3, 16-16) ^
                getBit(byte2, 15-8) ^
                getBit(byte3, 23-16) ^
                getBit(byte4, 31-24) ^
                getBit(byte4, 30-24) ^
                getBit(byte1, 5) ^
                getBit(byte2, 13-8) ^
                getBit(byte3, 21-16)) & 0x01)

    return E3

def getE4(bytes, algorithm):
    E4 = 0
    if algorithm == 1511:
        if len(bytes) == 2:
            byte1, byte2 = bytes
            E4 = ((getBit(byte2, 1) ^
                getBit(byte1, 5) ^
                getBit(byte2, 5) ^
                getBit(byte1, 4) ^
                getBit(byte2, 4) ^
                getBit(byte1, 3) ^
                getBit(byte2, 3) ^
                getBit(byte1, 2)) & 0x01)
        elif len(bytes) == 4:
            byte1, byte2, byte3, byte4 = bytes
            E4 = ((getBit(byte4, 24-24) ^

```

```

        getBit(byte1, 6) ^
        getBit(byte2, 14-8) ^
        getBit(byte3, 22-16) ^
        getBit(byte4, 30-24) ^
        getBit(byte1, 5) ^
        getBit(byte2, 13-8) ^
        getBit(byte3, 21-16)) & 0x01

    return E4

def getBitPosition(bytes, algorithm):
    BitPosition = 0
    if algorithm == 74:
        if len(bytes) == 2:
            dic = {1:0, 2:1, 3:7, 4:8, 5:15, 6:6, 7:14}
        elif len(bytes) == 4:
            dic = {1:0, 2:8, 3:7, 4:16, 5:15, 6:23, 7:31}
        E = ((getE3(bytes, algorithm) << 2) +
            (getE2(bytes, algorithm) << 1) +
            (getE1(bytes, algorithm) << 0))
    elif algorithm == 1511:
        if len(bytes) == 2:
            dic = {1:0, 2:1, 3:7, 4:8, 5:15, 6:6, 7:14, 8:9, 9:5, 10:13, 11:4, 12:12, 13:3,
14:11, 15:2}
        elif len(bytes) == 4:
            dic = {1:0, 2:8, 3:7, 4:16, 5:15, 6:23, 7:31, 8:24, 9:6, 10:14, 11:22, 12:30, 13:5,
14:13, 15:21}
        E = ((getE4(bytes, algorithm) << 3) +
            (getE3(bytes, algorithm) << 2) +
            (getE2(bytes, algorithm) << 1) +
            (getE1(bytes, algorithm) << 0))

    try:
        BitPosition = dic[E]
    except KeyError:
        BitPosition = None
    return BitPosition

def ecc_decode(bytes, algorithm):
    decoded_array = bytes
    if len(bytes) == 2:
        bitPosition = getBitPosition(bytes, algorithm)
        byte1, byte2 = bytes
        if bitPosition is not None:
            if bitPosition < 8:
                decoded_array[0] = (toggleKthBit(byte1, bitPosition))
                decoded_array[1] = (byte2)
            elif bitPosition < 16:
                decoded_array[0] = (byte1)
                decoded_array[1] = (toggleKthBit(byte2, bitPosition-8))
        else:
            decoded_array[0] = (byte1)
            decoded_array[1] = (byte2)
    elif len(bytes) == 4:
        bitPosition = getBitPosition(bytes, algorithm)
        byte1, byte2, byte3, byte4 = bytes
        if bitPosition is not None:
            if bitPosition < 8:
                decoded_array[0] = (toggleKthBit(byte1, bitPosition))
                decoded_array[1] = (byte2)
                decoded_array[2] = (byte3)
                decoded_array[3] = (byte4)
            elif bitPosition < 16:
                decoded_array[0] = (byte1)
                decoded_array[1] = (toggleKthBit(byte2, bitPosition-8))
                decoded_array[2] = (byte3)
                decoded_array[3] = (byte4)
            elif bitPosition < 24:
                decoded_array[0] = (byte1)
                decoded_array[1] = (byte2)
                decoded_array[2] = (toggleKthBit(byte3, bitPosition-16))
                decoded_array[3] = (byte4)
            elif bitPosition < 32:

```

```

        decoded_array[0] = (byte1)
        decoded_array[1] = (byte2)
        decoded_array[2] = (byte3)
        decoded_array[3] = (toggleKthBit(byte4, bitPosition-24))
    else:
        decoded_array[0] = (byte1)
        decoded_array[1] = (byte2)
        decoded_array[2] = (byte3)
        decoded_array[3] = (byte4)
    return decoded_array

#####
##

#####
##
## YUV Operations
#####
##

def read_yuv_frame(filename, width=X_RESOLUTION, height=Y_RESOLUTION):
    shape = (int(height*1.5), width)
    yuv = np.fromfile(filename, dtype='uint8').reshape(shape)
    return yuv

def save_yuv_frame(yuv_frame, name=".\\frame", width=X_RESOLUTION, height=Y_RESOLUTION):
    shape = (int(height*1.5), width)
    yuv_frame.reshape(shape).tofile(name+".yuv")

def save_yuv_frame_to_csv(yuv_frame, bits, name=".\\frame", width=X_RESOLUTION,
height=Y_RESOLUTION):
    size = int(height*1.5) * width * 8
    rows = (size/bits)
    shape = (rows, bits)
    binary_arr = np.unpackbits(yuv_frame.flatten()).astype(np.uint8)
    np.savetxt(name+".csv", binary_arr.reshape(shape), fmt="%d", delimiter=",")

def count_errors(yuv_original, yuv_error):
    xor = np.bitwise_xor(yuv_original.flatten(), yuv_error.flatten())
    num_errors = 0
    for x in xor:
        num_errors += bin(x).count("1")
    total_size = len(yuv_original.flatten())*8
    return (num_errors, total_size)

#####
##

def execute(filename, output_fp, err_per, thread_num, return_dict):
    # Log the current thread number
    print("Spawned thread %d" % thread_num)

    # Read in the frame
    yuv = read_yuv_frame(filename)
    original_arr = yuv.flatten()

    # Save the original frame
    save_yuv_frame(original_arr, output_fp+"original")
    save_yuv_frame_to_csv(original_arr, 16, output_fp+"original_16bit")
    save_yuv_frame_to_csv(original_arr, 32, output_fp+"original_32bit")

    # Inject errors in the original array
    original_error_arr = np.array_split(original_arr, int(len(original_arr.flatten())/2)) # Split
array into 2 byte/component chunks
    original_error_arr = np.apply_along_axis(injectError, 1, original_error_arr, err_per=err_per)

    # Save the original error frame
    save_yuv_frame(original_error_arr, output_fp+"original_error")
    save_yuv_frame_to_csv(original_error_arr, 16, output_fp+"original_error_16bit")
    save_yuv_frame_to_csv(original_error_arr, 32, output_fp+"original_error_32bit")

```



```

# calculate the amount of errors injected
single_bit, total_size = count_errors(original_arr.flatten(), original_error_arr.flatten())
print("%s Error %s %% - Number of errors: %s / %s %.03f%%" %
      (filename, err_per*100, single_bit, total_size, (float(single_bit)/(total_size))*100))

output = []
for algorithm in [74, 1511]:
    # Encode frame for both 16bit and 32bit
    encoded_16bit_arr = np.array_split(original_arr, int(len(original_arr.flatten())/2))
#16bit
    encoded_32bit_arr = np.array_split(original_arr, int(len(original_arr.flatten())/4))
#32bit
    encoded_16bit_arr = np.apply_along_axis(ecc_encode, 1, encoded_16bit_arr, algorithm)
    encoded_32bit_arr = np.apply_along_axis(ecc_encode, 1, encoded_32bit_arr, algorithm)

    # Save the 16bit and 32bit encoded frame
    save_yuv_frame(encoded_16bit_arr, output_fp+"encoded_16bit"+str(algorithm))
    save_yuv_frame(encoded_32bit_arr, output_fp+"encoded_32bit"+str(algorithm))
    save_yuv_frame_to_csv(encoded_16bit_arr, 16, output_fp+"encoded_16bit"+str(algorithm))
    save_yuv_frame_to_csv(encoded_32bit_arr, 32, output_fp+"encoded_32bit"+str(algorithm))

    # Inject the same errors across all frames that were previously generated
    encoded_error_16bit_arr = original_arr.flatten()
    encoded_error_32bit_arr = original_arr.flatten()
    for index, original_byte, error_byte, encoded_byte16, encoded_byte32 in
zip(range(len(original_arr.flatten()))),
original_arr.flatten(),
original_error_arr.flatten(),
encoded_16bit_arr.flatten(),
encoded_32bit_arr.flatten()):
        encoded_error_16bit_arr[index] = (injectSameError(original_byte, error_byte,
encoded_byte16))
        encoded_error_32bit_arr[index] = (injectSameError(original_byte, error_byte,
encoded_byte32))

    # Save the 16bit and 32bit encoded error frame
    save_yuv_frame(encoded_error_16bit_arr, output_fp+"encoded_error_16bit"+str(algorithm))
    save_yuv_frame(encoded_error_32bit_arr, output_fp+"encoded_error_32bit"+str(algorithm))

    # Decode the encoded error frames
    decoded_16bit_arr = np.array_split(encoded_error_16bit_arr,
int(len(encoded_error_16bit_arr.flatten())/2)) #16bit
    decoded_32bit_arr = np.array_split(encoded_error_32bit_arr,
int(len(encoded_error_32bit_arr.flatten())/4)) #32bit
    decoded_16bit_arr = np.apply_along_axis(ecc_decode, 1, decoded_16bit_arr, algorithm)
    decoded_32bit_arr = np.apply_along_axis(ecc_decode, 1, decoded_32bit_arr, algorithm)

    # Save the 16bit and 32bit decoded frame
    save_yuv_frame(decoded_16bit_arr, output_fp+"decoded_16bit"+str(algorithm))
    save_yuv_frame(decoded_32bit_arr, output_fp+"decoded_32bit"+str(algorithm))
    save_yuv_frame_to_csv(decoded_16bit_arr, 16, output_fp+"decoded_16bit"+str(algorithm))
    save_yuv_frame_to_csv(decoded_32bit_arr, 32, output_fp+"decoded_32bit"+str(algorithm))

    # Calculate quality metrics
    psnr_orig_error = str(psnr(original_arr.flatten(), original_error_arr.flatten()))
    psnr_encoded_16bit = str(psnr(original_arr.flatten(), encoded_16bit_arr.flatten()))
    psnr_encoded_32bit = str(psnr(original_arr.flatten(), encoded_32bit_arr.flatten()))
    psnr_encoded_16_bit_32bit = str(psnr(encoded_16bit_arr.flatten(),
encoded_32bit_arr.flatten()))
    psnr_error_16bit = str(psnr(original_arr.flatten(), encoded_error_16bit_arr.flatten()))
    psnr_error_32bit = str(psnr(original_arr.flatten(), encoded_error_32bit_arr.flatten()))
    psnr_error_16bit_32bit = str(psnr(encoded_error_16bit_arr.flatten(),
encoded_error_32bit_arr.flatten()))
    psnr_final_16bit = str(psnr(original_arr.flatten(), decoded_16bit_arr.flatten()))
    psnr_final_32bit = str(psnr(original_arr.flatten(), decoded_32bit_arr.flatten()))
    psnr_final_16bit_32bit = str(psnr(decoded_16bit_arr.flatten(),
decoded_32bit_arr.flatten()))

```

```

# print("PSNR original error "+str(algorithm)+": " + psnr_orig_error)
# print("PSNR encoded 16bit "+str(algorithm)+": " + psnr_encoded_16bit)
# print("PSNR encoded 32bit "+str(algorithm)+": " + psnr_encoded_32bit)
# print("PSNR encoded 16bit vs. 32bit "+str(algorithm)+": " + psnr_encoded_16_bit_32bit)
# print("PSNR error 16bit "+str(algorithm)+": " + psnr_error_16bit)
# print("PSNR error 32bit "+str(algorithm)+": " + psnr_error_32bit)
# print("PSNR error 16bit vs. 32bit "+str(algorithm)+": " + psnr_error_16bit_32bit)
# print("PSNR final 16bit "+str(algorithm)+": " + psnr_final_16bit)
# print("PSNR final 32bit "+str(algorithm)+": " + psnr_final_32bit)
# print("PSNR final 16bit vs. 32bit "+str(algorithm)+": " + psnr_final_16bit_32bit)

output.append([psnr_orig_error,
               psnr_encoded_16bit,
               psnr_encoded_32bit,
               psnr_encoded_16_bit_32bit,
               psnr_error_16bit,
               psnr_error_32bit,
               psnr_error_16bit_32bit,
               psnr_final_16bit,
               psnr_final_32bit,
               psnr_final_16bit_32bit])

return_dict.update({output_fp:output})

def main():
    # start fresh everytime
    try:
        shutil.rmtree("./analysis//")
    except Exception:
        pass
    finally:
        os.mkdir("./analysis//")

    thread_num = 0
    manager = multiprocessing.Manager()
    return_dict = manager.dict()
    jobs = []

    # start a thread for each frame
    for dirpath, _, filenames in os.walk(YUV_FRAMES):
        for frame in filenames:
            for err_per in range(ERROR_LOW, ERROR_HIGH, ERROR_INCREMENT):
                try:
                    os.mkdir("analysis//"+frame+str(err_per)+"//")
                except OSError:
                    print ("Creation of the directory %s failed, deleting..." %
                          "analysis//"+frame+str(err_per)+"//")
                finally:
                    thread = multiprocessing.Process(
                        target=execute,
                        args=(os.path.abspath(os.path.join(dirpath, frame)),
                            "analysis//"+frame+str(err_per)+"//",
                            float(err_per/10000.0),
                            thread_num,
                            return_dict))
                    jobs.append(thread)
                    thread.start()
                    thread_num = thread_num + 1
                    time.sleep(0.01)

                if thread_num >= THREAD_COUNT:
                    for thread in jobs:
                        thread.join()
                    jobs = []
                    thread_num = 0

    # Wait for all threads to finish
    for thread in jobs:

```

```

thread.join()

# Write output.csv file to log results
with open("analysis//output.csv", "wb") as file:
    wr = csv.writer(file, dialect='excel')
    wr.writerow(['file',
                'error_per',
                'psnr_orig_error_74',
                'psnr_encoded_16bit_74',
                'psnr_encoded_32bit_74',
                'psnr_encoded_16_bit_32bit_74',
                'psnr_error_16bit_74',
                'psnr_error_32bit_74',
                'psnr_error_16bit_32bit_74',
                'psnr_final_16bit_74',
                'psnr_final_32bit_74',
                'psnr_final_16bit_32bit_74',
                'psnr_orig_error_1511',
                'psnr_encoded_16bit_1511',
                'psnr_encoded_32bit_1511',
                'psnr_encoded_16_bit_32bit_1511',
                'psnr_error_16bit_1511',
                'psnr_error_32bit_1511',
                'psnr_error_16bit_32bit_1511',
                'psnr_final_16bit_1511',
                'psnr_final_32bit_1511',
                'psnr_final_16bit_32bit_1511'])

    for f in return_dict.keys():
        output = return_dict.get(f)
        for err_per in range(ERROR_LOW, ERROR_HIGH, ERROR_INCREMENT):
            desired_file = re.search(r'analysis//\w+(\.yuv\d+)', f).group(1)
            if '.yuv'+str(err_per) == desired_file:
                wr.writerow([f, str(float(err_per/10000.0)*100)]+[j for i in output for j in
i])

if __name__ == '__main__':
    main()

```

APPENDIX C. SRAM TEST PLATFORM SUITE

Description: This consists of the Arduino, Raspberry Pi, and SRAM voltage slave code used as an SRAM Test Platform. Written by Ali Ahmad Haidous.

C.1. Raspberry Pi Master Controller

C.1.1. arduino-slave.py

```
from smbus2 import SMBusWrapper, i2c_msg
import random
from enum import Enum
import struct

ARDUINO_ADDRESS = 0x04
TRANSACTION_LENGTH = 9

class I2CSlaveCommand(Enum):
    UNDEFINED_CMD = 0
    INVALID_CMD = 1
    WRITE_CMD = 2
    READ_CMD = 3

class I2CSlaveError(Enum):
    INVALID_ERROR = 0
    CRC_ERROR = 1
    LENGTH_ERROR = 2
    NO_RESPONSES_ERROR = 3

def writeBlock(block):
    block = []
    try:
        with SMBusWrapper(1) as bus:
            msg = i2c_msg.write(ARDUINO_ADDRESS, block)
            bus.i2c_rdwr(msg)
    except IOError:
        pass
    for m in msg:
        block.append(m)
    return block

def readBlock(numBytes):
    block = []
    try:
        with SMBusWrapper(1) as bus:
            msg = i2c_msg.read(ARDUINO_ADDRESS, numBytes)
            bus.i2c_rdwr(msg)
    except IOError:
        pass
    for m in msg:
        block.append(m)
    return block

def genCRC8(data, length):
    crc = 0xff;
    for i in range(length):
        crc ^= data[i];
        for j in range(8):
            if ((crc & 0x80) != 0):
                crc = ((crc << 1) ^ 0x31) & 0xff;
            else:
                crc <<= 1;
    return crc;

def createTransaction(command, address, data):
```

```

transaction = []
transaction.append(command)
transaction.append((address) & 0xFF)
transaction.append((address >> 8) & 0xFF)
transaction.append((address >> 16) & 0xFF)
transaction.append((address >> 24) & 0xFF)
transaction.append((data) & 0xFF)
transaction.append((data >> 8) & 0xFF)
transaction.append(0x00)
crc = genCRC8(transaction, TRANSACTION_LENGTH-1)
transaction.append(crc)
return transaction

def genRandomData():
    randomData = struct.unpack('>H', bytearray(random.getrandbits(8) for _ in xrange(2)))[0]
    return randomData

def writeSramBlock(startingAddress, length):
    sramBlock = []

    for i in range(length):
        randomData = genRandomData()
        transaction = createTransaction(I2CSlaveCommand.WRITE_CMD, startingAddress+i, randomData)
        writeBlock(transaction)
        rblock = readBlock(TRANSACTION_LENGTH)
        crc = genCRC8(rblock, TRANSACTION_LENGTH-1)
        if crc != rblock[TRANSACTION_LENGTH-1]:
            sramBlock.extend(writeSramBlock(startingAddress+i, 1))
        else:
            expectedRx = [I2CSlaveCommand.INVALID_CMD, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00,
I2CSlaveError.NO_RESPONSES_ERROR, crc]
            for i in range(TRANSACTION_LENGTH-1):
                if rblock[i] != expectedRx[i]:
                    sramBlock.extend(writeSramBlock(startingAddress+i, 1))
                    break
            sramBlock.append((startingAddress+i, randomData, 0))
    return sramBlock

def readSramBlock(startingAddress, length):
    sramBlock = []

    for i in range(length):
        transaction = createTransaction(I2CSlaveCommand.READ_CMD, startingAddress+i, 0x0000)
        writeBlock(transaction)
        rblock = readBlock(TRANSACTION_LENGTH)
        crc = genCRC8(rblock, TRANSACTION_LENGTH-1)
        if crc != rblock[TRANSACTION_LENGTH-1]:
            sramBlock.extend(readSramBlock(startingAddress+i, 1))
        rxaddress = rblock[4]
        rxaddress = (rxaddress << 8) | rblock[3]
        rxaddress = (rxaddress << 8) | rblock[2]
        rxaddress = (rxaddress << 8) | rblock[1]
        rxdata = rblock[6]
        rxdata = (rxdata << 8) | rblock[5]
        rxerror = rblock[7]
        sramBlock.append((rxaddress, rxdata, rxerror))
    return sramBlock

def main():
    wsramBlocks = writeSramBlock(0, 10)
    rsramBlocks = readSramBlock(0, 10)
    print wsramBlocks
    print rsramBlocks
    writeFile = open("sramdata.txt", "w+")
    writeFile.write("Address | Data | Error\r\n")
    for wsramBlock, rsramBlock in zip(wsramBlocks, rsramBlocks):
        waddress, wdata, werror = wsramBlock
        raddress, rdata, rerror = rsramBlock
        writeFile.write("w: " + hex(waddress) + " | " + hex(wdata) + " | " + hex(werror) + "
(\r\n")

```

```

        writeFile.write("r: " + hex(raddress) + " | " + hex(rdata) + " | " + hex(rerror) + "
|\r\n")
        writeFile.close()

if __name__ == "__main__":
    main()

```

C.2. Arduino SRAM Slave Controller

C.2.1. arduino_sram_slave.ino

```

#include "Global_Defines.h"
#include "CY62147GE.h"
#include "SRAM_Slave.h"
#include "I2C.h"

volatile union I2CBuffer_U buf;

uint16_t GetRandomOffset();
void VerifyWrite(uint32_t range);
void VerifyRead(uint32_t range);
void CommandVoltage(uint16_t voltage);

void setup() {
#ifdef SERIAL_DEBUG_ENABLED
    Serial.begin(115200);
#endif /*SERIAL_DEBUG_ENABLED*/
    I2c.Init(I2C_OUR_SLAVE_ADDRESS);
    SramSlave.Init();
#ifdef VALIDATE_SRAM
    CommandVoltage(250);
#else
#endif
    randomSeed(analogRead(0));
}

uint16_t GetRandomOffset()
{
    uint16_t random_offset = random(0x1000, 0xFFFF);
#ifdef SERIAL_DEBUG_ENABLED
    Serial.print("Random offset: ");
    Serial.println(random_offset, HEX);
#endif /*SERIAL_DEBUG_ENABLED*/

    return random_offset;
}

void loop() {

#ifdef VALIDATE_SRAM
    uint16_t voltage = 0;
    for (voltage = 150; voltage > 95; voltage--)
    {
        CY62147GE_Powerdown();
        delay(50);
        CommandVoltage(50);
        delay(500);
        CommandVoltage(voltage);
        delay(500);
    }
#endif

#ifdef VERIFY_ZEROS
    VerifyWrite(MAX_SRAM_ADDRESS, 0);
#else
    uint16_t offset = GetRandomOffset();
    VerifyWrite(MAX_SRAM_ADDRESS, offset);
#endif
}

```

```

    delay(500);
#ifdef VERIFY_ZEROS
    VerifyRead(MAX_SRAM_ADDRESS, 0);
#else
    VerifyRead(MAX_SRAM_ADDRESS, offset);
#endif

    // Reset SRAM
    CommandVoltage(220);
    delay(500);
    uint32_t address = 0;
    for(address = 0; address < MAX_SRAM_ADDRESS; address++)
    {
        CY62147GE_Write(address, (uint16_t)0xFFFF);
    }
}

#endif
while(1);
}

void VerifyWrite(uint32_t range, uint16_t offset)
{
    uint32_t address = 0;
    for(address = 0; address < range; address++)
    {
#ifdef VERIFY_ZEROS
        CY62147GE_Write(address, (uint16_t)0);
#else
        CY62147GE_Write(address, (uint16_t)(address + offset));
#endif
        if (!(address % 1024))
        {
#ifdef SERIAL_DEBUG_ENABLED_VALIDATION
            Serial.print("Address: ");
            Serial.print(address, HEX);
            Serial.println(" written");
#endif /*SERIAL_DEBUG_ENABLED_VALIDATION*/
        }
    }
}

void VerifyRead(uint32_t range, uint16_t offset)
{
    uint32_t address = 0;
    uint16_t data = 0;
    uint8_t error = 0;

    for(address = 0; address < range; address++)
    {
        CY62147GE_Read(address, &data, &error);
        if (!(address % 1024))
        {
#ifdef SERIAL_DEBUG_ENABLED_VALIDATION
            Serial.print("Address: ");
            Serial.print(address, HEX);
            Serial.println(" read");
#endif /*SERIAL_DEBUG_ENABLED_VALIDATION*/
        }
#ifdef VERIFY_ZEROS
        if ((uint16_t)0 != data)
#else
        if ((uint16_t)(address + offset) != data)
#endif
        {
#ifdef SERIAL_DEBUG_ENABLED
            Serial.print("Address: ");
            Serial.print(address, HEX);
            Serial.print(" expected: ");

```

```

#ifdef VERIFY_ZEROS
    Serial.print((uint16_t)0, HEX);
#else
    Serial.print((uint16_t)(address + offset), HEX);
#endif
    Serial.print(" data: ");
    Serial.println(data, HEX);
#endif /*SERIAL_DEBUG_ENABLED*/
}
}

void CommandVoltage(uint16_t voltage)
{
    buf.Buffer.Length = I2C_HEADER_LENGTH+2;
    buf.Buffer.Command = DAC_OUTPUT_CMD;
    buf.Buffer.Data[0] = voltage & 0xFF;
    buf.Buffer.Data[1] = (voltage >> 8) & 0xFF;
    I2c.Transmit(I2C_VOLTAGE_SLAVE, &buf);
    delay(100);
    I2c.Transmit(I2C_VOLTAGE_SLAVE, &buf);
    float analog = ((float)voltage)/100.0;

#ifdef SERIAL_DEBUG_ENABLED
    Serial.print("Command Voltage: ");
    Serial.print(analog);
    Serial.println("V");
#endif /*SERIAL_DEBUG_ENABLED*/
}

```

C.2.2. SRAM_Slave.h

```

#ifndef SRAM_SLAVE_H
#define SRAM_SLAVE_H

#include "Global_Defines.h"
#include "I2C.h"

#define SRAM_MAX_ADDRESS (0x3FFFF)
#define SRAM_MIN_ADDRESS (0x00000)

class SRAM_Slave
{
private:
    static union I2CBuffer_U respBuffer;
    static uint16_t sramData;
    static uint8_t sramError;
#ifdef TEST_I2C
    static void testI2c(uint8_t *buffer);
#endif /*TEST_I2C*/
    static void writeSram(uint8_t *buffer);
    static void readSram(uint8_t *buffer);
    static I2C i2c;
public:
    SRAM_Slave();
    void Init();
#ifdef TEST_SRAM
    void Test_SRAM_Write(uint32_t lowRange, uint32_t highRange, uint16_t offset);
    void Test_SRAM_Read(uint32_t lowRange, uint32_t highRange, uint16_t offset);
#endif /*TEST_SRAM*/
};

extern SRAM_Slave SramSlave;

#endif /*SRAM_SLAVE_H*/

```


C.2.3. SRAM_Slave.cpp

```
#include "CY62147GE.h"
#include "I2C.h"
#include "SRAM_Slave.h"

#define VERIFY_RANGE (0xFFFFu)
#define ADD_OFFSET (0U)

union I2CBuffer_U SRAM_Slave :: respBuffer;
uint16_t SRAM_Slave :: sramData;
uint8_t SRAM_Slave :: sramError;

SRAM_Slave :: SRAM_Slave()
{
}

void SRAM_Slave :: Init()
{
#ifdef TEST_SRAM
    CY62147GE_Init();
    I2c.AddCommandRouter(WRITE_SRAM_CMD, writeSram);
    I2c.AddCommandRouter(READ_SRAM_CMD, readSram);
#endif /*TEST_SRAM*/
#ifdef TEST_I2C
    I2c.AddCommandRouter(TEST_I2C_CMD, testI2c);
#endif /*TEST_I2C*/
#ifdef TEST_SRAM_DEBUG_ENABLED
    Serial.println("Init SRAM Slave");
#endif /*TEST_SRAM_DEBUG_ENABLED*/
}

#ifdef TEST_I2C
void SRAM_Slave :: testI2c(uint8_t *buffer)
{
    uint8_t i;

    for (i = 0; i < BUFFER_LENGTH-I2C_HEADER_LENGTH; i++)
    {
        respBuffer.Buffer.Data[i] = buffer[i];
    }

    respBuffer.Buffer.Command = TEST_I2C_RESP;
    respBuffer.Buffer.Length = BUFFER_LENGTH;
    I2c.PostResponse(&respBuffer);

#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED
    Serial.print("testI2c: ");
    for (i = 0; i < BUFFER_LENGTH; i++)
    {
        Serial.print(respBuffer.Bytes[i], HEX);
        Serial.print(" ");
    }
    Serial.println();
#endif /*I2C_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
}
#endif /*TEST_I2C*/

#ifdef TEST_SRAM
void SRAM_Slave :: writeSram(uint8_t *buffer)
{
    uint32_t address;
    uint16_t data;
    uint8_t i;

    address = buffer[3];
    address = (address << 8) | buffer[2];

```

```

address = (address << 8) | buffer[1];
address = (address << 8) | buffer[0];
data = buffer[5];
data = (data << 8) | buffer[4];

CY62147GE_Write(address, data);

respBuffer.Buffer.Data[0] = buffer[0];
respBuffer.Buffer.Data[1] = buffer[1];
respBuffer.Buffer.Data[2] = buffer[2];
respBuffer.Buffer.Data[3] = buffer[3];
respBuffer.Buffer.Data[4] = buffer[4];
respBuffer.Buffer.Data[5] = buffer[5];
respBuffer.Buffer.Command = WRITE_SRAM_RESP;
respBuffer.Buffer.Length = I2C_HEADER_LENGTH + 6;
I2c.PostResponse(&respBuffer);

#ifdef SERIAL_DEBUG_ENABLED
#ifdef TEST_SRAM_DEBUG_ENABLED
Serial.print("writeSram: ");
Serial.print(" address: ");
Serial.print(address, HEX);
Serial.print(" data: ");
Serial.print(data, HEX);
Serial.println();
#endif /*TEST_SRAM_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
}

void SRAM_Slave :: readSram(uint8_t *buffer)
{
    uint32_t address;
    uint8_t i;

    address = buffer[3];
    address = (address << 8) | buffer[2];
    address = (address << 8) | buffer[1];
    address = (address << 8) | buffer[0];

    CY62147GE_Read(address, &sramData, &sramError);

    respBuffer.Buffer.Data[0] = buffer[0];
    respBuffer.Buffer.Data[1] = buffer[1];
    respBuffer.Buffer.Data[2] = buffer[2];
    respBuffer.Buffer.Data[3] = buffer[3];
    respBuffer.Buffer.Data[4] = (uint8_t)(sramData & 0xFF);
    respBuffer.Buffer.Data[5] = (uint8_t)((sramData >> 8) & 0xFF);
    respBuffer.Buffer.Data[6] = sramError;
    respBuffer.Buffer.Command = READ_SRAM_RESP;
    respBuffer.Buffer.Length = I2C_HEADER_LENGTH + 7;
    I2c.PostResponse(&respBuffer);

#ifdef SERIAL_DEBUG_ENABLED
#ifdef TEST_SRAM_DEBUG_ENABLED
Serial.print("readSram: ");
Serial.print(" address: ");
Serial.print(address, HEX);
Serial.print(" data: ");
Serial.print(sramData, HEX);
Serial.print(" error: ");
Serial.print(sramError, HEX);
Serial.println();
#endif /*TEST_SRAM_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
}
#endif /*TEST_SRAM*/

SRAM_Slave SramSlave = SRAM_Slave();

```

C.2.4. I2C.h

```
#ifndef I2C_H
#define I2C_H

#include "Global_Defines.h"
#include <Wire.h>

#define I2C_HEADER_LENGTH (3u)
#define I2C_RESPONSE_QUEUE_LENGTH (10u)

enum I2CCommand_E
{
    INVALID_CMD,
    TEST_I2C_CMD,
    WRITE_SRAM_CMD,
    READ_SRAM_CMD,
    DAC_OUTPUT_CMD,
    MAX_COMMANDS
};

enum I2CResponse_E
{
    INVALID_RESP = MAX_COMMANDS+1,
    TEST_I2C_RESP,
    WRITE_SRAM_RESP,
    READ_SRAM_RESP,
    MAX_RESPONSES
};

enum I2CErrorCodes_E
{
    NO_ERROR,
    NO_RESPONSES_ERROR,
    LENGTH_ERROR,
    CRC_ERROR,
    COMMAND_ERROR,
    INVALID_ERROR
};

struct I2CBuffer_S {
    uint8_t Crc;
    uint8_t Length;
    uint8_t Command;
    uint8_t Data[BUFFER_LENGTH - I2C_HEADER_LENGTH];
};

union I2CBuffer_U {
    struct I2CBuffer_S Buffer;
    uint8_t Bytes[BUFFER_LENGTH];
};

struct I2CResponseQueue_S {
    union I2CBuffer_U Buffers[I2C_RESPONSE_QUEUE_LENGTH];
    uint8_t Head;
    uint8_t Tail;
    uint8_t Size;
    uint16_t Overrun;
};

using I2CCmdHandler = void(*) (uint8_t *);

struct I2CCommandRouter_S {
    uint8_t Command;
    I2CCmdHandler Handler;
};

class I2C
{
private:
```

```

static union I2CBuffer_U rxCommand;
static struct I2CResponseQueue_S txResponseQueue;
static struct I2CCommandRouter_S commandRouter[];
static uint8_t commands;

static void receiveData(int byteCount);
static void sendData();
static void removeResponse(union I2CBuffer_U *resp);
static void postError(uint8_t error, uint8_t errorReason);
static void routeCommand(struct I2CBuffer_S *buf);
static uint8_t genCRC8(uint8_t *data, uint8_t len);
public:
    I2C();
    void Init(uint8_t address);
    void AddCommandRouter(uint8_t cmd, I2CCmdHandler cmdHandler);
    void Transmit(uint8_t slaveAddress, union I2CBuffer_U *buf);
    bool Request(uint8_t slaveAddress, union I2CBuffer_U *buf);
    void PostResponse(union I2CBuffer_U *resp);
    void Scanner(uint8_t* devices, uint8_t* numDevices);
};

extern I2C I2c;

#endif /*I2C_H*/

```

C.2.5. I2C.cpp

```

#include "I2C.h"

#define I2C_STANDARD_MODE    (100000u)
#define I2C_FAST_MODE       (400000u)
#define I2C_FAST_PLUS_MODE  (1000000u)
#define I2C_HIGH_SPEED_MODE (3400000u)

union I2CBuffer_U I2C :: rxCommand;
struct I2CResponseQueue_S I2C :: txResponseQueue;
struct I2CCommandRouter_S I2C :: commandRouter[MAX_COMMANDS];
uint8_t I2C :: commands;

////////////////////////////////////
/// Public
////////////////////////////////////

I2C :: I2C()
{
}

void I2C :: Init(uint8_t address)
{
    Wire.setClock(I2C_HIGH_SPEED_MODE);
    Wire.begin(address);
    Wire.onReceive(receiveData);
    Wire.onRequest(sendData);

    commands = 0;
    txResponseQueue.Head = 0;
    txResponseQueue.Tail = 0;
    txResponseQueue.Size = 0;
    txResponseQueue.Overrun = 0;
#ifdef I2C_DEBUG_ENABLED
    Serial.println("Init I2C");
#endif /*I2C_DEBUG_ENABLED*/
}

void I2C :: AddCommandRouter(uint8_t cmd, I2CCmdHandler cmdHandler)
{

```

```

    if (commands < MAX_COMMANDS)
    {
        commandRouter[commands].Command = cmd;
        commandRouter[commands].Handler = cmdHandler;
        commands++;
    }
}

void I2C :: Transmit(uint8_t slaveAddress, union I2CBuffer_U *buf)
{
    uint8_t i;

    buf->Buffer.Crc = genCRC8(&(buf->Bytes[1]), (buf->Buffer.Length) - 1);

    Wire.beginTransmission(slaveAddress);
    Wire.write((const uint8_t *) (buf->Bytes), buf->Buffer.Length);
    Wire.endTransmission();

#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED
    Serial.print("i2ctx: ");
    for (i = 0; i < buf->Buffer.Length; i++)
    {
        Serial.print(buf->Bytes[i], HEX);
        Serial.print(" ");
    }
    Serial.println();
#endif
#endif /*I2C_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
}

bool I2C :: Request(uint8_t address, union I2CBuffer_U *buf)
{
    uint8_t i;
    bool retVal = true;
    uint8_t rxBytes = 0;
    uint8_t crc;

    Wire.requestFrom(address, (uint8_t)BUFFER_LENGTH);
    while (Wire.available())
    {
        buf->Bytes[rxBytes++] = Wire.read();
    }
    crc = genCRC8(&(buf->Bytes[1]), (buf->Buffer.Length) - 1);
    if ((crc != buf->Buffer.Crc) || (rxBytes != buf->Buffer.Length))
    {
        retVal = false;
#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED
        Serial.print("Bad CRC ");
#endif
#endif /*I2C_DEBUG_ENABLED*/
#ifdef SERIAL_DEBUG_ENABLED
    }
    else
    {
        retVal = true;
    }
#endif

#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED
    Serial.print("i2crx: ");
    for (i = 0; i < rxBytes; i++)
    {
        Serial.print(buf->Bytes[i], HEX);
        Serial.print(" ");
    }
    Serial.println();
#endif
#endif /*I2C_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/

    return retVal;
}

```

```

}

void I2C :: PostResponse(union I2CBuffer_U *resp)
{
    uint8_t i;

    if (txResponseQueue.Size >= I2C_RESPONSE_QUEUE_LENGTH)
    {
        txResponseQueue.Overrun++;
    }
    else
    {
        resp->Buffer.Crc = genCRC8 (&(resp->Bytes[1]), (resp->Buffer.Length) - 1);
        for (i = 0; i < resp->Buffer.Length; i++)
        {
            txResponseQueue Buffers[txResponseQueue.Head].Bytes[i] = resp->Bytes[i];
        }
        txResponseQueue.Head++;
        txResponseQueue.Size++;
        if (txResponseQueue.Head >= I2C_RESPONSE_QUEUE_LENGTH)
        {
            txResponseQueue.Head = 0;
        }
    }
}

void I2C :: Scanner(uint8_t* devices, uint8_t* numDevices)
{
    uint8_t error;
    uint8_t address;
    uint8_t i;

    *numDevices = 0;
    for (address = 1; address < 127; address++)
    {
        // The i2c_scanner uses the return value of
        // the Write.endTransmission to see if
        // a device did acknowledge to the address.
        Wire.beginTransmission(address);
        error = Wire.endTransmission();

        if (error == 0)
        {
            devices[(*numDevices)++] = address;
        }
    }

#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED
    Serial.print("I2C Scanner: ");
    Serial.print(*numDevices);
    Serial.print(" device(s) found at address(es) -> ");
    for (i = 0; i < *numDevices; i++)
    {
        Serial.print(devices[i], HEX);
        Serial.print(" ");
    }
    Serial.println();
#endif /*I2C_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
}

//////////
/// Private
//////////

void I2C :: removeResponse(union I2CBuffer_U *resp)
{
    uint8_t i;

    if (txResponseQueue.Size == 0)

```

```

    {
        resp->Buffer.Length = 4;
        resp->Buffer.Command = INVALID_CMD;
        resp->Buffer.Data[0] = NO_RESPONSES_ERROR;
        resp->Buffer.Crc = genCRC8(&(resp->Bytes[1]), resp->Buffer.Length - 1);
    }
    else
    {
        for (i = 0; i < txResponseQueue Buffers[txResponseQueue.Tail].Buffer.Length; i++)
        {
            resp->Bytes[i] = txResponseQueue Buffers[txResponseQueue.Tail].Bytes[i];
        }
        txResponseQueue.Size--;
        txResponseQueue.Tail++;
        if (txResponseQueue.Tail >= I2C_RESPONSE_QUEUE_LENGTH)
        {
            txResponseQueue.Tail = 0;
        }
    }
}

void I2C :: receiveData(int byteCount)
{
    uint8_t i;
    uint8_t crc;

    for (i = 0; (i < BUFFER_LENGTH) && Wire.available(); i++)
    {
        rxCommand.Bytes[i] = Wire.read();
    }

    if ((byteCount > BUFFER_LENGTH) || (rxCommand.Buffer.Length > BUFFER_LENGTH))
    {
        postError(LENGTH_ERROR, byteCount);
        while (Wire.available())
        {
            (void)Wire.read();
        }
#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED
        Serial.print("Bad Length ");
#endif /*I2C_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
    }

    crc = genCRC8(&rxCommand.Bytes[1], rxCommand.Buffer.Length - 1);
    if (crc != rxCommand.Buffer.Crc)
    {
        postError(CRC_ERROR, crc);
#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED
        Serial.print("Bad CRC ");
#endif /*I2C_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
    }
    else if (rxCommand.Buffer.Command >= MAX_COMMANDS)
    {
        postError(COMMAND_ERROR, rxCommand.Buffer.Command);
#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED
        Serial.print("Bad Command ");
#endif /*I2C_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
    }
    else
    {
        routeCommand(&(rxCommand.Buffer));
    }

#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED

```

```

Serial.print("i2c>>> ");
for (i = 0; i < BUFFER_LENGTH; i++)
{
    Serial.print(rxCommand.Bytes[i], HEX);
    Serial.print(" ");
}
Serial.println();
#endif /*I2C_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
}

void I2C :: sendData()
{
    uint8_t i;
    union I2CBuffer_U resp;

    removeResponse(&resp);
    Wire.write((const uint8_t *)&resp.Bytes[0], resp.Buffer.Length);

#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED
    Serial.print("i2c<<< ");
    for (i = 0; i < resp.Buffer.Length; i++)
    {
        Serial.print(resp.Bytes[i], HEX);
        Serial.print(" ");
    }
    Serial.println();
#endif /*I2C_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
}

void I2C :: postError(uint8_t error, uint8_t errorReason)
{
    union I2CBuffer_U errorBuf;

    errorBuf.Buffer.Data[0] = error;
    errorBuf.Buffer.Data[1] = errorReason;
    errorBuf.Buffer.Command = INVALID_RESP;
    errorBuf.Buffer.Length = I2C_HEADER_LENGTH + 2;
    I2c.PostResponse(&errorBuf);
}

void I2C :: routeCommand(struct I2CBuffer_S *buf)
{
    uint8_t i;
    for (i = 0; i < MAX_COMMANDS; i++)
    {
        if (commandRouter[i].Command == buf->Command)
        {
            commandRouter[i].Handler(&(buf->Data[0]));
        }
    }
}

uint8_t I2C :: genCRC8(uint8_t *data, uint8_t len)
{
    uint8_t crc = 0xff;
    uint8_t i, j;
    for (i = 0; i < len; i++)
    {
        crc ^= data[i];
        for (j = 0; j < 8; j++)
        {
            if ((crc & 0x80) != 0) crc = (uint8_t)((crc << 1) ^ 0x31);
            else crc <<= 1;
        }
    }
    return crc;
}

```



```
I2C I2c = I2C();
```

C.2.6. CY62147GE.h

```
#ifndef CY62147GE_H
#define CY62147GE_H

#include "Global_Defines.h"

#ifdef TEST_SRAM

#ifdef __cplusplus
extern "C"
{
#endif

#define MAX_SRAM_ADDRESS (262144u)

extern void CY62147GE_Init(void);
extern void CY62147GE_Powerdown(void);
extern void CY62147GE_Write(uint32_t address, uint16_t data);
extern void CY62147GE_Read(uint32_t address, uint16_t *data, uint8_t *error);

#ifdef __cplusplus
} // extern "C"
#endif

#endif /*TEST_SRAM*/

#endif /*CY62147GE_H*/
```

C.2.7. GY62147GE.c

```
#include "CY62147GE_Defines.h"
#include "CY62147GE.h"

#ifdef __cplusplus
extern "C"
{
#endif

#ifdef TEST_SRAM

static void writePins(PinInfo_T *pins, uint8_t size, uint32_t state);
static void readPins(PinInfo_T *pins, uint8_t size, uint32_t *state);
static void changeChipMode(ChipMode_T mode);
static void changePinMode(PinInfo_T *pins, uint8_t size, uint8_t mode);

void CY62147GE_Init(void)
{
    changePinMode(&ChipControlPins[0], NUMBER_OF_CONTROL_PINS, OUTPUT);
    changeChipMode(DESELECT_POWERDOWN);

    changePinMode(&ChipAddressPins[0], NUMBER_OF_ADDRESS_PINS, OUTPUT);
    changePinMode(&ChipIOPins[0], NUMBER_OF_IO_PINS, INPUT);
    changePinMode(&ChipGndPins[0], NUMBER_OF_GND_PINS, OUTPUT);
    changePinMode(&ChipVoltagePins[0], NUMBER_OF_GND_PINS, OUTPUT);
    pinMode(TSOP_ERR, INPUT);

    writePins(&ChipGndPins[0], NUMBER_OF_GND_PINS, 0b00);
    writePins(&ChipVoltagePins[0], NUMBER_OF_VOLTAGE_PINS, 0b11);
    changeChipMode(OUTPUT_DISABLED);
}

void CY62147GE_Powerdown(void)
```

```

{
    changeChipMode(DESELECT_POWERDOWN);
}

void CY62147GE_Write(uint32_t address, uint16_t data)
{
    writePins(&ChipAddressPins[0], NUMBER_OF_ADDRESS_PINS, address);
    changePinMode(&ChipIOPins[0], NUMBER_OF_IO_PINS, OUTPUT);
    writePins(&ChipIOPins[0], NUMBER_OF_IO_PINS, data);
    changeChipMode(DATA_IN_IO0_IO15);
    delayMicroseconds(1); // write the SRAM
    changeChipMode(OUTPUT_DISABLED);
}

void CY62147GE_Read(uint32_t address, uint16_t *data, uint8_t *error)
{
    writePins(&ChipAddressPins[0], NUMBER_OF_ADDRESS_PINS, address);
    changePinMode(&ChipIOPins[0], NUMBER_OF_IO_PINS, INPUT);
    changeChipMode(DATA_OUT_IO0_IO15);
    delayMicroseconds(1); // read the SRAM
    readPins(&ChipIOPins[0], NUMBER_OF_IO_PINS, data);
    *error = digitalRead(TSOP_ERR);
    changeChipMode(OUTPUT_DISABLED);
}

static void changeChipMode(ChipMode_T mode)
{
    switch (mode)
    {
        {
            case DESELECT_POWERDOWN:
                digitalWrite(TSOP_BLE, 1);
                digitalWrite(TSOP_BHE, 1);
                digitalWrite(TSOP_OE, 1);
                digitalWrite(TSOP_WE, 1);
                digitalWrite(TSOP_CE, 1);
                break;
            case DATA_OUT_IO0_IO15:
                digitalWrite(TSOP_WE, 1);
                digitalWrite(TSOP_CE, 0);
                digitalWrite(TSOP_BLE, 0);
                digitalWrite(TSOP_BHE, 0);
                digitalWrite(TSOP_OE, 0);
                break;
            case OUTPUT_DISABLED:
                digitalWrite(TSOP_BLE, 0);
                digitalWrite(TSOP_BHE, 0);
                digitalWrite(TSOP_OE, 1);
                digitalWrite(TSOP_WE, 1);
                digitalWrite(TSOP_CE, 0);
                break;
            case DATA_IN_IO0_IO15:
                digitalWrite(TSOP_OE, 1);
                digitalWrite(TSOP_BLE, 0);
                digitalWrite(TSOP_BHE, 0);
                digitalWrite(TSOP_CE, 0);
                digitalWrite(TSOP_WE, 0);
                break;
            default:
                break;
        }
    }
}

static void writePins(PinInfo_T *pins, uint8_t size, uint32_t state)
{
    uint8_t index;

    for (index = 0; index < size; index++)
    {
        digitalWrite(pins[index].Pin, ((state >> index) & 0x01));
    }
}

```

```

}

static void readPins(PinInfo_T *pins, uint8_t size, uint32_t *state)
{
    uint8_t index;
    uint32_t tempState = 0;

    for (index = 0; index < size; index++)
    {
        tempState += ((digitalRead(pins[index].Pin) & 0x01) << index);
    }
    *state = tempState;
}

static void changePinMode(PinInfo_T *pins, uint8_t size, uint8_t mode)
{
    uint8_t index;
    for (index = 0; index < size; index++)
    {
        pinMode(pins[index].Pin, mode);
    }
}

#endif /*TEST_SRAM*/

#ifdef __cplusplus
} // extern "C"
#endif

```

C.2.8. CY62147GE_Defines.h

```

#ifndef CY62147GE_DEFINES_H
#define CY62147GE_DEFINES_H

#include "Global_Defines.h"

#ifdef TEST_SRAM

#include <stdint.h>

#ifdef __cplusplus
extern "C"
{
#endif

#define NUMBER_OF_PINS (44u)
#define NUMBER_OF_ADDRESS_PINS (18u)
#define NUMBER_OF_IO_PINS (16u)
#define NUMBER_OF_CONTROL_PINS (5u)
#define NUMBER_OF_GND_PINS (2u)
#define NUMBER_OF_VOLTAGE_PINS (2u)

typedef enum ChipPinout_E
{
    /* Left side */
    TSOP_A4 = 53u,
    TSOP_A3 = 51u,
    TSOP_A2 = 49u,
    TSOP_A1 = 47u,
    TSOP_A0 = 45u,
    TSOP_CE = 43u, /* active low */
    TSOP_IO0 = 41u,
    TSOP_IO1 = 39u,
    TSOP_IO2 = 37u,
    TSOP_IO3 = 35u,
    TSOP_VCO = 33u, /* VOLTAGE */
    TSOP_VSS1 = 31u, /* GND */
    TSOP_IO4 = 29u,

```

```

TSOP_IO5 = 27u,
TSOP_IO6 = 25u,
TSOP_IO7 = 23u,
TSOP_WE = 2u, /* active low */
TSOP_A17 = 3u,
TSOP_A16 = 4u,
TSOP_A15 = 5u,
TSOP_A14 = 6u,
TSOP_A13 = 7u,

/* Right side */
TSOP_A5 = 52u,
TSOP_A6 = 50u,
TSOP_A7 = 48u,
TSOP_OE = 46u, /* active low */
TSOP_BHE = 44u,
TSOP_BLE = 42u, /* active low */
TSOP_IO15 = 40u,
TSOP_IO14 = 38u,
TSOP_IO13 = 36u,
TSOP_IO12 = 34u,
TSOP_VSS2 = 32u, /* GND */
TSOP_VCC = 30u, /* VOLTAGE */
TSOP_IO11 = 28u,
TSOP_IO10 = 26u,
TSOP_IO9 = 24u,
TSOP_IO8 = 22u,
TSOP_ERR = 8u,
TSOP_A8 = 9u,
TSOP_A9 = 10u,
TSOP_A10 = 11u,
TSOP_A11 = 12u,
TSOP_A12 = 13u,
}ChipPinout_T;

typedef enum ChipMode_E
{
    DESELECT_POWERDOWN,
    DATA_OUT_IO0_IO15,
    DATA_OUT_IO0_IO17,
    DATA_OUT_IO8_IO15,
    OUTPUT_DISABLED,
    DATA_IN_IO0_IO15,
    DATA_IN_IO0_IO17,
    DATA_IN_IO8_IO15,
} ChipMode_T;

typedef struct PinInfo_S
{
    const uint8_t Pin;
} PinInfo_T;

PinInfo_T ChipAddressPins[NUMBER_OF_ADDRESS_PINS] =
{
    { TSOP_A0 },
    { TSOP_A1 },
    { TSOP_A2 },
    { TSOP_A3 },
    { TSOP_A4 },
    { TSOP_A5 },
    { TSOP_A6 },
    { TSOP_A7 },
    { TSOP_A8 },
    { TSOP_A9 },
    { TSOP_A10 },
    { TSOP_A11 },
    { TSOP_A12 },
    { TSOP_A13 },
    { TSOP_A14 },
    { TSOP_A15 },
    { TSOP_A16 },

```

```

    { TSOP_A17 },
};

PinInfo_T ChipIOPins[NUMBER_OF_IO_PINS] =
{
    { TSOP_IO0 },
    { TSOP_IO1 },
    { TSOP_IO2 },
    { TSOP_IO3 },
    { TSOP_IO4 },
    { TSOP_IO5 },
    { TSOP_IO6 },
    { TSOP_IO7 },
    { TSOP_IO8 },
    { TSOP_IO9 },
    { TSOP_IO10 },
    { TSOP_IO11 },
    { TSOP_IO12 },
    { TSOP_IO13 },
    { TSOP_IO14 },
    { TSOP_IO15 },
};

PinInfo_T ChipControlPins[NUMBER_OF_CONTROL_PINS] =
{
    { TSOP_BLE },
    { TSOP_BHE },
    { TSOP_OE },
    { TSOP_WE },
    { TSOP_CE },
};

PinInfo_T ChipGndPins[NUMBER_OF_GND_PINS] =
{
    { TSOP_VSS1 },
    { TSOP_VSS2 },
};

PinInfo_T ChipVoltagePins[NUMBER_OF_VOLTAGE_PINS] =
{
    { TSOP_VCO },
    { TSOP_VCC },
};

#endif /*TEST_SRAM*/
#ifdef __cplusplus
} // extern "C"
#endif

#endif /*CY62147GE_DEFINES_H*/

```

C.2.9. Global_Defines.h

```

#ifndef GLOBAL_DEFINES_H
#define GLOBAL_DEFINES_H

#include <Arduino.h>

enum I2C_SLAVES
{
    I2C_TEST_SLAVE = (0x03u),
    I2C_SRAM_SLAVE = (0x04u),
    I2C_VOLTAGE_SLAVE = (0x05u),
};

#define I2C_OUR_SLAVE_ADDRESS I2C_SRAM_SLAVE
#define I2C_THEIR_SLAVE_ADDRESS I2C_VOLTAGE_SLAVE

```

```

#define SERIAL_DEBUG_ENABLED
//#define I2C_DEBUG_ENABLED
//#define TEST_SRAM_DEBUG_ENABLED
//#define TEST_SRAM_READ_DEBUG_ENABLED
#define TEST_SRAM
#define TEST_I2C
#define VALIDATE_SRAM
#define VERIFY_ZEROS

#endif /*GLOBAL_DEFINES_H*/

```

C.3. Arduino Voltage Slave Controller

C.3.1. arduino_voltage_slave.ino

```

#include "Voltage_Slave.h"
#include "I2C.h"

void setup() {
  Serial.begin(115200);
  VoltageSlave.Init();
  I2c.Init(I2C_OUR_SLAVE_ADDRESS);
  Serial.println("Ready");

#ifdef TEST_I2C
  delay(1000);
  union I2CBuffer_U buf;
  uint8_t devices[10];
  uint8_t numDevices;

  I2c.Scanner(&devices[0], &numDevices);
#endif
}

void loop() {
#ifdef HEARTBEAT
  delay(1000);
  Serial.print("My i2c address is ");
  Serial.print(I2C_OUR_SLAVE_ADDRESS);
  Serial.print(" - Heartbeat -> ");
  Serial.println(heartbeatcounter, DEC);
  heartbeatcounter++;
#endif
}

```

C.3.2. Voltage_Slave.h

```

#ifndef VOLTAGE_SLAVE
#define VOLTAGE_SLAVE

#include "Global_Defines.h"
#include "I2C.h"

class Voltage_Slave
{
private:
  static union I2CBuffer_U respBuffer;
  static void writeVoltage(uint8_t * buf);
#ifdef TEST_I2C
  static void testI2c(uint8_t *buf);
#endif /*TEST_I2C*/
public:
  Voltage_Slave();

```

```

    void Init();
    void WriteVoltage(float analog);
};

extern Voltage_Slave VoltageSlave;

#endif /*VOLTAGE_SLAVE*/

```

C.3.3. Voltage_Slave.cpp

```

#include "Voltage_Slave.h"

union I2CBuffer_U Voltage_Slave :: respBuffer;

Voltage_Slave :: Voltage_Slave()
{
}

void Voltage_Slave :: Init()
{
    analogWriteResolution(12);
    I2c.AddCommandRouter(DAC_OUTPUT_CMD, writeVoltage);
#ifdef TEST_I2C
    I2c.AddCommandRouter(TEST_I2C_CMD, testI2c);
#endif /*TEST_I2C*/
    Serial.println("Init Voltage_Slave");
}

void Voltage_Slave :: WriteVoltage(float analog)
{
    // Arduino Due does not have an analog output voltage from 0 V to Vref,
    // but from 1/6 to 5/6 of the reference voltage, that is, 0.55 V
    // and 2.75V with Vref = 3.3 V.

    int digital = 0;
    if ((analog >= 0.55) && (analog <= 2.75)) {
        digital = ((analog-0.55)/2.20)*4095;
    } else if (analog < 0.55) {
        digital = 0;
    }
    else {
        digital = 4095;
    }
    analogWrite(DAC0, digital);
}

void Voltage_Slave :: writeVoltage(uint8_t * buf)
{
    uint16_t digital = (buf[0] + (buf[1] << 8)) & 0xFFFF;
    float analog = ((float)digital)/100.0;
    VoltageSlave.WriteVoltage(analog);
#ifdef SERIAL_DEBUG_ENABLED
    Serial.print("writeVoltage: ");
    Serial.print(analog);
    Serial.print("V");
    Serial.println();
#endif /*SERIAL_DEBUG_ENABLED*/
}

#ifdef TEST_I2C
void Voltage_Slave :: testI2c(uint8_t *buf)
{
    uint8_t i;

    for (i = 0; i < BUFFER_LENGTH-I2C_HEADER_LENGTH; i++)
    {
        respBuffer.Buffer.Data[i] = buf[i];
    }
}

```

```
respBuffer.Buffer.Command = TEST_I2C_RESP;
respBuffer.Buffer.Length = BUFFER_LENGTH;
I2c.PostResponse(&respBuffer);

#ifdef SERIAL_DEBUG_ENABLED
#ifdef I2C_DEBUG_ENABLED
Serial.print("testI2c: ");
for (i = 0; i < BUFFER_LENGTH; i++)
{
    Serial.print(respBuffer.Bytes[i], HEX);
    Serial.print(" ");
}
Serial.println();
#endif /*I2C_DEBUG_ENABLED*/
#endif /*SERIAL_DEBUG_ENABLED*/
}
#endif /*TEST_I2C*/

Voltage_Slave VoltageSlave = Voltage_Slave();
```


APPENDIX D. BIT TRUNCATION MANAGER

Description: This IP core is provided to take in a server protocol with frame number, YUV truncation, x1, y1, x2, y2 and a local H264 CAVLC decoder frame_num to bit truncate the exact pixel address of a 24-bit data width 22-bit address width frame buffer to support up-to 1920x1080 progressive (1080p) resolution. The pixels are truncated one at a time using the system clock to realize the least hardware cost possible. Written by: Ali Ahmad Haidous.

D.1. BitTruncationManager IP Core

```
//-----  
---  
// Design      : BitTruncationManager  
// Author(s)   : Ali Haidous  
// Email       : ali.haidous@gmail.com  
//  
// Description: This IP core is provided to take in a server protocol with frame number,  
//              YUV truncation, x1, y1, x2, y2 and a local H264 CAVLC decoder frame_num  
//              to bit truncate the exact pixel address of a 24-bit data width 22-bit  
//              address width frame buffer to support up-to 1920x1080 progressive (1080p)  
//              resolution. The pixels are truncated one at a time using the system clock  
//              to realize the least hardware cost possible.  
//  
//  
//  
//  
// Copyright (C) 2021 Ali Ahmad Haidous  
// All rights reserved  
//-----  
---  
  
//          Truncate          YUV  
`define no_truncation      3'b000  
`define v_truncation       3'b001  
`define u_truncation       3'b010  
`define uv_truncation      3'b011  
`define y_truncation       3'b100  
`define yv_truncation      3'b101  
`define yu_truncation      3'b110  
`define yuv_truncation     3'b111  
  
// this is the mobile device's BitTruncationManager implemented into the H264 decoder  
module BitTruncationManager  
(  
    clock,                // system clock to support pixel truncation  
    decoder_frame_number, // from h264 decoder - frame number currently decoded from the  
decoder  
    frame_number,         // from server - frame number from server protocol  
    yuv_truncation,       // from server - yuv truncation, reference enum above  
    x1,                   // from server - x coordinate region top left  
    y1,                   // from server - y coordinate region top left  
    x2,                   // from server - x coordinate region bottom right  
    y2,                   // from server - y coordinate region bottom right  
    frame_number_request, // to server - frame number requested  
    send_frame_flag,      // to server - flag to send frame  
    truncate_y,           // to frame buffer - 1920 x 1080 max resolution, 2,073,600 y pixel  
addresses
```

```

truncate_u,          // to frame buffer - 1920 x 1080 max resolution, 2,073,600 u pixel
addresses
truncate_v          // to frame buffer - 1920 x 1080 max resolution, 2,073,600 v pixel
addresses
);

// System clock to truncate the frame buffer one pixel at a time
input clock;

// H264 CAVLC Decoder to Bit Truncation Manager
input[21:0] decoder_frame_number;

// Server to Mobile Device to Bit Truncation Manager
input[21:0] frame_number;
input[2:0] yuv_truncation;
input[21:0] x1;
input[21:0] y1;
input[21:0] x2;
input[21:0] y2;

// Bit Truncation Manager to Mobile Device to Server
output reg[21:0] frame_number_request;
output reg send_frame_flag;

// Bit Truncation Manager to Frame Buffer to Display
output reg[21:0] truncate_y;
output reg[21:0] truncate_u;
output reg[21:0] truncate_v;

integer x;
integer y;
integer current_x2;
integer current_y2;

// initialize locals
initial begin
    x <= 2047;
    y <= 2047;
    current_x2 <= 0;
    current_y2 <= 0;
end

// request the frame number currently being decoded by the decoder from the server
always @(decoder_frame_number or frame_number) begin
    if (decoder_frame_number != frame_number) begin
        frame_number_request <= decoder_frame_number;
        send_frame_flag <= 1'b1;
    end else if (decoder_frame_number == frame_number) begin
        send_frame_flag <= 1'b0;
    end
end

// iterate over all the pixels to be truncated in the given region
always @(posedge clock) begin
    if (x2 != current_x2 &&
        y2 != current_y2) begin
        x <= x1;
        y <= y1;
        current_x2 <= x2;
        current_y2 <= y2;
    end else begin
        if (x <= current_x2) begin
            x <= x + 1;
        end else if (y <= current_y2) begin
            x <= x1;
            y <= y + 1;
        end
    end
end
end

```

```

// truncate the pixels in the given region
if (x <= current_x2 ||
    y <= current_y2) begin
    case (yuv_truncation)
        `no_truncation : begin
            truncate_y <= 22'b0;
            truncate_u <= 22'b0;
            truncate_v <= 22'b0;
        end
        `v_truncation  : begin
            truncate_y <= 22'b0;
            truncate_u <= 22'b0;
            truncate_v <= x * y;
        end
        `u_truncation  : begin
            truncate_y <= 22'b0;
            truncate_u <= x * y;
            truncate_v <= 22'b0;
        end
        `uv_truncation : begin
            truncate_y <= 22'b0;
            truncate_u <= x * y;
            truncate_v <= x * y;
        end
        `y_truncation  : begin
            truncate_y <= x * y;
            truncate_u <= 22'b0;
            truncate_v <= 22'b0;
        end
        `yu_truncation : begin
            truncate_y <= x * y;
            truncate_u <= x * y;
            truncate_v <= 22'b0;
        end
        `yv_truncation : begin
            truncate_y <= x * y;
            truncate_u <= 22'b0;
            truncate_v <= x * y;
        end
        `yuv_truncation : begin
            truncate_y <= x * y;
            truncate_u <= x * y;
            truncate_v <= x * y;
        end
        default        : begin
            truncate_y <= 22'b0;
            truncate_u <= 22'b0;
            truncate_v <= 22'b0;
        end
    endcase
end
end

endmodule // BitTruncationManager

```

D.2. BitTruncationManager Test Bench

```

//-----
---
// Design      : BitTruncationManager_tb
// Author(s)   : Ali Haidous
// Email       : ali.haidous@gmail.com
//
// Description: Test bench for the BitTruncationManager IP core.
//
//
//

```

```

//
//
//
//
// Copyright (C) 2021 Ali Ahmad Haidous
// All rights reserved
//-----
--

`timescale 1 ns/10 ps // time-unit = 1 ns, precision = 10 ps

module BitTruncationManager_tb;

    // System clock to truncate the frame buffer one pixel at a time
    reg clock;

    // H264 CAVLC Decoder to Bit Truncation Manager
    reg[21:0] decoder_frame_number;

    // Server to Mobile Device to Bit Truncation Manager
    reg[21:0] frame_number;
    reg[2:0] yuv_truncation;
    reg[21:0] x1;
    reg[21:0] y1;
    reg[21:0] x2;
    reg[21:0] y2;

    // Bit Truncation Manager to Mobile Device to Server
    wire[21:0] frame_number_request;
    wire send_frame_flag;

    // Bit Truncation Manager to Frame Buffer to Display
    wire[21:0] truncate_y;
    wire[21:0] truncate_u;
    wire[21:0] truncate_v;

    // duration for each bit = 20 * timescale = 20 * 1 ns = 20ns
    localparam period = 20;

    BitTruncationManager DUT(.clock(clock),
        .decoder_frame_number(decoder_frame_number),
        .frame_number(frame_number),
        .yuv_truncation(yuv_truncation),
        .x1(x1),
        .y1(y1),
        .x2(x2),
        .y2(y2),
        .frame_number_request(frame_number_request),
        .send_frame_flag(send_frame_flag),
        .truncate_y(truncate_y),
        .truncate_u(truncate_u),
        .truncate_v(truncate_v));

    initial begin
        forever begin
            clock = 0;
            #period; // wait for period
            clock = 1;
            #period; // wait for period
        end
    end

    initial begin

        // test frame request
        #period; // wait for period
        decoder_frame_number = 55;
        frame_number = 35;
    end
endmodule

```

```

#period; // wait for period
frame_number = 55;
#period; // wait for period

#period; // wait for period
decoder_frame_number = 56;
frame_number = 55;
#period; // wait for period
frame_number = 56;
#period; // wait for period

#period; // wait for period
decoder_frame_number = 57;
frame_number = 56;
#period; // wait for period
frame_number = 57;
#period; // wait for period

#period; // wait for period
decoder_frame_number = 58;
frame_number = 57;
#period; // wait for period
frame_number = 58;
#period; // wait for period

// test yuv truncation
yuv_truncation = 7;
x1 = 1;
y1 = 1;
x2 = 8;
y2 = 7;
#840;

x1 = 19;
y1 = 25;
x2 = 90;
y2 = 80;
#78100;

yuv_truncation = 6;
x1 = 14;
y1 = 19;
x2 = 83;
y2 = 74;
#75900;

yuv_truncation = 5;
x1 = 111;
y1 = 113;
x2 = 587;
y2 = 473;
#3427200;

yuv_truncation = 4;
x1 = 111;
y1 = 211;
x2 = 381;
y2 = 471;
#1404000;

yuv_truncation = 3;
x1 = 31;
y1 = 41;
x2 = 58;
y2 = 67;
#14040;

yuv_truncation = 2;
x1 = 991;
y1 = 991;

```

```

x2 = 998;
y2 = 997;
#840;

yuv_truncation = 1;
x1 = 881;
y1 = 881;
x2 = 888;
y2 = 887;
#840;

yuv_truncation = 0;
x1 = 211;
y1 = 211;
x2 = 268;
y2 = 277;
#75240;

yuv_truncation = 3;
x1 = 471;
y1 = 471;
x2 = 498;
y2 = 497;
#14040;

yuv_truncation = 6;
x1 = 751;
y1 = 751;
x2 = 788;
y2 = 787;
#26640;

yuv_truncation = 5;
x1 = 981;
y1 = 981;
x2 = 998;
y2 = 997;
#5440;

yuv_truncation = 2;
x1 = 145;
y1 = 146;
x2 = 848;
y2 = 745;
#8421940;

yuv_truncation = 4;
x1 = 212;
y1 = 213;
x2 = 338;
y2 = 337;
#312480;

yuv_truncation = 0;
x1 = 241;
y1 = 241;
x2 = 268;
y2 = 287;
#24840;

yuv_truncation = 7;
x1 = 271;
y1 = 271;
x2 = 298;
y2 = 377;
#57240;

#100; // Let the simulation finish
end

endmodule // BitTruncationManager_tb

```