UC Davis
UC Davis Electronic Theses and Dissertations

Title
Design of Display Stream Compression Video Codecs

Permalink
https://escholarship.org/uc/item/9j85h9hn

Author
Wu, Shifu

Publication Date
2021

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Design of Display Stream Compression Video Codecs

By

SHIFU WU
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Bevan M. Baas, Chair

Venkatesh Akella

Rajeevan Amirtharajah

Committee in Charge
2021

© Copyright by Shifu Wu 2021
All Rights Reserved

Abstract

Video displays with ultra-high-definition (UHD) resolutions such as 4K (3840×2160) and 8K (7680×4320) are now available. Video frame rates such as 120 frames per second (fps) and beyond are becoming more prevalent. Moreover, new display technologies have enabled wide color gamut (WCG) and high dynamic range (HDR). As a result, the required bandwidth to transmit uncompressed video data over display links has dramatically increased (e.g., 120 Gbps for 8K videos with 30-bit color at 120 fps); however, the physical layer bandwidth is not keeping pace with this demand. To address the disparity, a widely accepted low-cost solution is to compress the video stream prior to transmission and decompress upon being displayed. The Display Stream Compression (DSC) standard developed by Video Electronics Standards Association (VESA) enables low-cost and low-latency hardware implementations of visually lossless video codecs over display links. This dissertation analyzes the DSC algorithm, presents three hardware encoder architectures and the design of a fabricated and first published encoder chip, discusses four hardware decoder architectures and six decoder implementations, and describes the design of many-core software DSC decoders.

The DSC encoder hardware architectures for a slice encoder, a slice-interleaved encoder, and a time-interleaved encoder are presented. A DSC encoder chip based on the time-interleaved encoder architecture and supporting up to 4K video resolution is designed and fabricated in TSMC 28 nm CMOS technology. The chip is capable of processing two slices in parallel, resulting in a throughput of two pixels per cycle for 4:4:4 pixels, and four pixels per cycle for 4:2:2 and 4:2:0 pixels. The chip shares combinational computational resources across slices, requiring 627.7 K logic gates, yielding a 1.75 times logic area reduction. The time-interleaved encoding scheme lowers energy per pixel by 1.87–1.96 times compared to non-interleaved encoding at nominal voltage. At 1.15 V, the chip achieves up to 1448 megapixels per second (Mpixels/s) at 362 MHz, which is equivalent to 174.6 fps for 4K videos. At 0.8 V, it achieves 441 Mpixels/s for 4:2:2/4:2:0 pixels and 221 Mpixels/s for 4:4:4 pixels, while demonstrating a minimum energy of 84 pJ, 92 pJ, and 163 pJ per 4:2:0, 4:2:2, and 4:4:4 pixel, respectively. The chip achieves 2.7–33 times lower area and 2.0–45 times better throughput

per area compared to prior video encoder chips.

Next, this dissertation presents four hardware architectures for DSC decoders. A slice decoder design realizes the DSC algorithm and achieves a throughput of three pixels per cycle for 4:4:4 pixels, and six pixels per cycle for 4:2:2 and 4:2:0 pixel formats. A slice-interleaved decoder architecture is proposed to support decoding of multiple columns of slices per picture with minimum area overhead. A parallel slice decoder architecture utilizes multiple parallel slice decoders to linearly increase the throughput. In addition, a parallel-interleaved decoder architecture offers area and throughput trade offs. To evaluate the proposed architectures, six decoders are implemented in 28 nm CMOS using a standard-cell-based design flow. The six implemented decoders support decoding of 1080p (1920×1080) videos, and require 169.6–627 K logic gates. At 1.0 V, the decoders operate at maximum frequencies of 495–610 MHz, achieve maximum throughput of 1490–13,040 Mpixels/s while dissipating 22.9–58.6 pJ per pixel. The slice decoder achieves five times higher throughput than prior work.

Furthermore, this dissertation presents the design of software DSC decoders on a fine-grained many-core processor array. The slice decoder exploits fine-grained task-level and component-level parallelism and can decode pictures configured into one column of slices; it is implemented with 88 processors and 2 memory modules. The parallel slice decoders facilitate higher performance by leveraging scalable slice-level parallelism; two designs that process two and four slices in parallel are implemented utilizing from 178 processors and 4 memory modules to 359 processors and 6 memory modules. At 1.75 GHz and 1.1 V, the proposed decoders decode 1080p videos in 4:2:0, 4:2:2, and 4:4:4 pixel formats—achieving up to 94.7 fps, 95.6 fps, and 47.9 fps. The minimum energy of 11.8 nJ, 13.3 nJ, and 23.4 nJ per 4:2:0, 4:2:2, and 4:4:4 pixel is achieved in the slice decoder at 0.76 V. The proposed designs achieve up to 159 times higher throughput and 769 times lower energy per pixel than a DSC decoder implemented on one core of an Intel i7-7700HQ processor.

To my wife Hui, and my parents.

Acknowledgments

The journey of my PhD study has been challenging but yet a learning and growing experience. I would not be able to get it done without the support from many people.

I would like to express my sincerest gratitude to my advisor and mentor, Professor Bevan Baas, for his valuable guidance and support on my research throughout my PhD study. He introduced me into digital VLSI design research and I have learned from him on conducting independent research and presenting work with clear writing. I have benefited a lot from his vision, enthusiasm, and valuable feedback.

I would like to thank Professor Venkatesh Akella for serving in my qualifying examination and dissertation committee, and Professor Rajeevan Amirtharajah for serving in my dissertation committee. I deeply appreciate their valuable feedback on my dissertation. I would like to thank Professor Hussain Al-Asaad, Professor Khaled Abdel-Ghaffar, and Professor S. Felix Wu for serving on my qualifying examination committee.

I would like to thank Professor Massimo Alioto for his guidance on the DSC chip design and funding support for the chip fabrication, and for his valuable time and guidance in presenting the work in a clear and accurate manner of paper writing.

I would like to acknowledge Kalana De Silva and Snehlata Gutgutia for their valuable work on the DSC encoder chip project. I would like to extend my appreciation to Viveka Konandur for his help in the chip package related matters.

I would like to thank Brent Bohnenstiehl for his guidance and support over the years. He is very patient and always available. I have learned a lot from him. I would like to thank Brent for his work on the KiloCore software tool chain and Mark Hildebrand for his work on the task mapping on many-core processor arrays, which enabled efficient design of DSC decoders on the KiloCore processor array.

My special thanks goes to Satyabrata Sarangi and Timothy Andreas for their endless support as colleagues and great friends. We met when I started my PhD and have been working together and stay in close touch since then. I received generous help from Timothy during the testing of the DSC chip, which is highly appreciated. I would also like to thank both of them for their valuable and detailed feedback on my papers and dissertation.

I am grateful to have the privilege of working with past and current members of the

VLSI Computation Laboratory not already mentioned: Aaron Stillmaker, Bin Liu, Jon Pimentel, Emmanuel Adeagbo, Peiyao Shi, Renjie Chen, Jin Cui, Filipe Borges, Christi Tain, Sharmila Kulkarni, Yushan Wu, Arthur Hlaing, Zhangfan Zhao, Ziyuan Dong, Haotian Wu, Yikai Mao, Tony Tsoi, Yuanyuan Xiang, Delvin Huynh, and Sarvagya Singh.

I would like to express my special appreciation to my wife, Hui Wang. This dissertation would not be possible without her endless support and encouragement. I also owe many thanks to my parents for their support and help.

Contents

Abstract ii

Acknowledgments v

List of Figures x

List of Tables xiii

1 Introduction 1
- 1.1 Video Coding . . . 1
  - 1.1.1 Overview . . . 1
  - 1.1.2 Video Coding Standards . . . 2
  - 1.1.3 Compare DSC With Other Video Coding Standards . . . 5
- 1.2 Research Motivation . . . 5
- 1.3 Dissertation Organization . . . 8

2 Display Stream Compression (DSC) Standard 9
- 2.1 Overview . . . 9
- 2.2 DSC Encoding Algorithm . . . 12
  - 2.2.1 DSC Encoding Process . . . 12
  - 2.2.2 Color Space Conversion . . . 14
  - 2.2.3 Prediction, Quantization, and Reconstruction . . . 14
  - 2.2.4 Indexed Color History (ICH) . . . 19
  - 2.2.5 Line Buffer . . . 21
  - 2.2.6 VLC Entropy Encoding . . . 22
  - 2.2.7 Rate Control . . . 23
  - 2.2.8 Substream and Slice Multiplexing . . . 25
- 2.3 DSC Decoding Algorithm . . . 26
  - 2.3.1 DSC Decoding Process . . . 26
  - 2.3.2 Slice and Substream Demultiplexing . . . 27
  - 2.3.3 VLC Entropy Decoding . . . 28
  - 2.3.4 Rate Control . . . 29
  - 2.3.5 Line Buffer . . . 29
  - 2.3.6 Prediction, Inverse Quantization, and Reconstruction . . . 29
  - 2.3.7 Indexed Color History . . . 30
  - 2.3.8 Color Space Conversion . . . 30
- 2.4 Summary . . . 30

3 Hardware Architectures for DSC Encoders 31
3.1 Slice Encoder . . . 31
3.1.1 System Controllers . . . 32
3.1.2 Prediction, Quantization, and Reconstruction . . . 33
3.1.3 Indexed Color History . . . 36
3.1.4 Line Buffer . . . 36
3.1.5 Rate Control . . . 37
3.1.6 VLC Entropy Encoder . . . 38
3.1.7 Substream Multiplexing . . . 39
3.2 Slice-Interleaved Encoder . . . 44
3.2.1 Architecture . . . 44
3.2.2 Dynamic Memory Allocation Across Slices . . . 45
3.2.3 Slice Encode Timing and Extra Buffering . . . 46
3.3 Time-Interleaved Encoder . . . 48
3.3.1 Time-Interleaved Encoding Architecture . . . 48
3.3.2 Memory Architecture . . . 50
3.3.3 Slice Encode Timing and Extra Buffering . . . 52
3.4 Summary . . . 55

4 A DSC Encoder Chip in 28 nm for 4K Video Resolution 56
4.1 Design Goals and Key Features . . . 56
4.2 Chip-Level Architecture . . . 57
4.2.1 Double Data Rate I/O . . . 57
4.2.2 Dual-Clock FIFO . . . 60
4.2.3 On-Chip Test Generator . . . 60
4.2.4 Memories . . . 62
4.3 Physical Design . . . 64
4.3.1 Physical Design Flow . . . 64
4.3.2 Physical Design Results . . . 66
4.4 Chip and Testing . . . 70
4.4.1 Chip . . . 70
4.4.2 Testing . . . 71
4.5 Measured Results . . . 72
4.5.1 Throughput . . . 73
4.5.2 Energy Efficiency . . . 75
4.5.3 Multi-Instance Scalability . . . 76
4.6 Comparison With Video Encoder Chips . . . 77
4.7 Summary . . . 77

5 Hardware Architectures and Physical Design of DSC Decoders 79
5.1 Slice Decoder . . . 79
5.1.1 Substream Demultiplexing . . . 81
5.1.2 Entropy Decoder . . . 81
5.1.3 Rate Control . . . 81
5.1.4 Prediction, Inverse Quantization, and Reconstruction . . . 82
5.1.5 Indexed Color History . . . 82
5.1.6 Line Buffer . . . 82
5.2 Slice-Interleaved Decoder . . . 83
5.2.1 Architecture . . . 83

5.2.2 Slice Decode Timing and Extra Buffering . . . 83
5.3 Parallel Slice Decoders . . . 85
5.3.1 Architecture . . . 86
5.3.2 Slice Decode Timing and Extra Buffering . . . 86
5.4 Parallel-Interleaved Decoder . . . 90
5.4.1 Architecture . . . 91
5.4.2 Slice Decode Timing and Extra Buffering . . . 92
5.5 DSC Decoder Implementations in 28 nm CMOS . . . 94
5.5.1 Design Flow and Implementations . . . 94
5.5.2 Results . . . 97
5.5.3 Scalability . . . 100
5.5.4 Comparison With Related Work . . . 101
5.6 Summary . . . 101

6 DSC Decoders for Fine-Grained Many-Core Processor Arrays 104
6.1 Introduction . . . 104
6.2 The Targeted Many-Core Processor Arrays . . . 105
6.2.1 The KiloCore Chip . . . 105
6.2.2 Programming Methodology . . . 105
6.3 Slice Decoder . . . 107
6.3.1 Rate Buffer and Substream Demultiplexer . . . 108
6.3.2 Entropy Decoder . . . 110
6.3.3 Rate Control . . . 110
6.3.4 Line Buffer . . . 111
6.3.5 Prediction, Inverse Quantization, and Reconstruction . . . 112
6.3.6 Indexed Color History . . . 114
6.3.7 Mapping Slice Decoder to a Many-Core Processor Array . . . 124
6.4 Parallel Slice Decoders . . . 125
6.4.1 High-Level Dataflow . . . 125
6.4.2 Parallel-2-Slice Decoder . . . 128
6.4.3 Parallel-4-Slice Decoder . . . 128
6.5 Simulation Results and Analysis . . . 128
6.5.1 Experimental Setup . . . 131
6.5.2 Throughput . . . 132
6.5.3 Area Efficiency . . . 133
6.5.4 Energy Efficiency . . . 134
6.5.5 Scalability for Higher Resolutions and Performance . . . 135
6.5.6 Comparison With Other Software Video Decoders . . . 136
6.6 Summary . . . 136

7 Conclusion and Future Work 139
7.1 Conclusion . . . 139
7.2 Future Work . . . 140
7.2.1 DSC Algorithm Optimization . . . 141
7.2.2 DSC Codec on a Single Chip . . . 141
7.2.3 DSC Encoders for Fine-Grained Many-Core Processor Arrays . . . 141

Glossary 142

Bibliography 147

List of Figures

1.1 Uncompressed data rate for various video resolutions 6

2.1 DSC usage in end-to-end systems 10
2.2 YCbCr 4:2:2 pixels packed into virtual 4:4:4:4 container pixels 10
2.3 YCbCr 4:2:0 pixels packed into virtual 4:4:4 container pixels 11
2.4 Slice configurations 12
2.5 DSC encoding process 13
2.6 Pixels used in modified median-adaptive prediction 15
2.7 Block prediction example 16
2.8 Pixels used in sum of absolute differences calculation for BP search 17
2.9 ICH entries pointing to previous line in native 4:2:0 and 4:2:2 modes 19
2.10 Indexed color history in the DSC encoder 20
2.11 ICH update examples 21
2.12 DSU-VLC coding example 22
2.13 ICH coding example 23
2.14 Substream multiplexer and rate buffer in the DSC encoder 25
2.15 DSC decoding process 27
2.16 Substream demultiplexing in the DSC decoder 28

3.1 System controllers used in the slice encoder 32
3.2 Pipelined block diagram of prediction, quantization, reconstruction, and ICH 34
3.3 Four-stage pipeline of the BP search 35
3.4 Four steps to find the vector with the smallest sum of absolute differences 35
3.5 Component-wise memory architecture for the line buffer 36
3.6 Pipelined block diagram of rate control in the slice encoder 38
3.7 Synchronous FIFO circuit 40
3.8 Mux word construction from syntax element for balance FIFO 41
3.9 Datapath of a slice-interleaved encoder architecture 44
3.10 Dynamic memory allocation in the slice-interleaved encoder 45
3.11 Slice-interleaved encode timing for two and four slices per line 47
3.12 Time-interleaved encoding architecture 49
3.13 Dynamic memory allocation in the time-interleaved encoder 51
3.14 Time-interleaved slice encode timing for two slices per line 53
3.15 Time-interleaved slice encode timing for four slices per line 54

4.1 Chip-level block diagram of the DSC encoder chip 58
4.2 Double data rate I/O 59
4.3 Dual-clock FIFO circuit 59

4.4 Circuit of the 17-bit linear-feedback shift register 61
4.5 Physical design flow of the DSC encoder chip 65
4.6 DSC encoder core floorplan 67
4.7 DSC encoder core layout showing only the clock nets 67
4.8 DSC encoder core layout 68
4.9 DSC encoder core layout with power and ground nets hidden 68
4.10 DSC encoder chip layout 69
4.11 DSC encoder core logic area breakdown 70
4.12 DSC encoder chip die micrograph 71
4.13 Testing setup for the DSC encoder chip 72
4.14 Maximum core clock frequency versus supply voltage of the DSC encoder chip . . . 73
4.15 Maximum throughput versus supply voltage of the DSC encoder chip 74
4.16 Energy per pixel versus supply voltage of the DSC encoder chip 75
4.17 Multiple encoder instances scalability 76

5.1 Pipelined block diagram of the slice decoder 80
5.2 Slice-interleaved decode timing for two and four slices per line 84
5.3 Parallel slice decoder architecture 86
5.4 Parallel slice decode timing for two slices per line 87
5.5 Parallel slice decode timing for four slices per line 89
5.6 Parallel-interleaved decoder architecture 91
5.7 Parallel-interleaved slice decode timing for four slices per line 93
5.8 Core layouts of the six implemented DSC decoders 96
5.9 Comparison of the six implemented DSC decoders 97
5.10 Annotated layout of the slice decoder 98
5.11 Logic area breakdown of the slice decoder 99
5.12 Memory size requirements for 1080p and 4K 101

6.1 Die photo and annotated layouts of the KiloCore chip 106
6.2 Dataflow of the rate buffer, substream demultiplexer, and entropy decoder 109
6.3 Dataflow of the rate control module 111
6.4 Line buffer storage for different pixel components 112
6.5 Dataflow of the prediction, inverse quantization, reconstruction, ICH, and line buffer 113
6.6 Dataflow and processor mapping of the shift entry approach 115
6.7 Data dependency of the shift entry approach 116
6.8 Dataflow and processor mapping of the separate index and entry update approach . 117
6.9 ICH index table update examples 117
6.10 Dataflow of the parallel index update approach 118
6.11 Processor mapping of the parallel index update ICH designs 119
6.12 Processor mapping of the Parallel-Index-16 ICH design 120
6.13 Processor mapping of the Parallel-Index-32 ICH design 121
6.14 Throughput and frame rates of the many-core and i7 ICH designs 122
6.15 Throughput per unit chip area of the many-core and i7 ICH designs 122
6.16 Energy per 1080p frame of the many-core ICH designs 123
6.17 Processor mapping of the slice decoder 125
6.18 Area and energy breakdown of the slice decoder 126
6.19 Dataflow of the parallel slice decoder 126
6.20 Rate buffer and pixel buffer memory storage organization for parallel slice decoders . 127
6.21 Dataflow and processor mapping of the parallel-2-slice decoder 129

6.22 Dataflow and processor mapping of the parallel-4-slice decoder 130
6.23 Throughput versus voltage of the many-core software DSC decoders 132
6.24 Throughput per area versus voltage of the many-core software DSC decoders 133
6.25 Energy per pixel versus voltage of the many-core software DSC decoders 134
6.26 Energy per pixel versus throughput of the many-core software DSC decoders 135

List of Tables

3.1 Size FIFO word width and depth 42

4.1 Binary code and gray code for 3-bit numbers 61
4.2 Memory blocks in the DSC encoder chip 63
4.3 Comparison of video encoders with 4K resolution or higher 78

5.1 Comparison of the proposed DSC decoders 103

6.1 Simulation voltages and frequencies 131
6.2 Comparison of software video decoder implementations 138

Chapter 1

Introduction

This chapter presents a brief overview of video coding and some state-of-the-art video coding standards. The research motivation and organization of this dissertation are also discussed.

1.1 Video Coding

1.1.1 Overview

Video has been widely used in many applications, such as video recording and playback, video streaming, video conferencing, video surveillance, and so on. A video consists of a series of frames. Depending on the frame size, some common video resolutions include full high definition (Full HD) (1920×1080), 4K ultra-high-definition (UHD) (3840×2160), and 8K UHD (7680×4320). Uncompressed videos require very high data rates for real-time transmission and large space for storage. For example, 96 gigabits per second (Gbps) is required to transmit 8K videos with 8 bits per color component (bpc) at 120 frames per second (fps); 12 GB storage space is required to store one second of such uncompressed videos. Many applications reduce the size of video data through compression, which is achieved by encoding the uncompressed video data using video coding algorithms. The video data contain spatial redundancy within each frame and temporal redundancy between different frames. Video coding algorithms exploit ways to reduce the amount of redundancies which yields smaller data size to represent the video. Since lossless compression can only achieve very moderate compression ratio, practical video coding is mostly lossy which means the video data cannot be completely restored after compression and decompression.

1.1.2 Video Coding Standards

Videos are compressed and decompressed based on standardized video coding algorithms. Some state-of-the-art video coding standards are discussed in this section.

The ITU-T and ISO/IEC standards have been widely used in daily lives. ITU-T developed the H.261 [1] standard in 1988 and the H.263 [2] standard in 1996, whereas ISO/IEC developed the MPEG-1 [3] standard in 1991 and the MPEG-4 Visual [4] standard in 1999. The two organizations jointly standardized the H.262/MPEG-2 Video [5] standard in 1995, the H.264/MPEG-4 Advanced Video Coding (AVC) [6] standard in 2003, the High Efficiency Video Coding (HEVC) [7] standard in 2013, and the newest Versatile Video Coding (VVC) [8] standard in 2020.

The VP8 [9] video coding standard was developed by On2 Technologies as a successor for its VPx codec family, which includes the VP3, VP4, VP5, VP6, and VP7 standards. VP8 was transferred to Google and open sourced in 2010. The standard was enhanced resulting in the next generation called VP9 [10]. AV1 [11] is the latest video coding standard of this series and was finalized by the Alliance for Open Media (AOMedia) in 2018.

The Joint Photographic Experts Group (JPEG) developed the Motion JPEG 2000 standard in 2001 and the JPEG XS standard [12] in 2019.

The Video Electronics Standards Association (VESA) has developed two display compression codecs: the Display stream compression (DSC) standard in 2014 and the VESA Display Compression-M (VDC-M) in 2018.

H.264/MPEG-4 Advanced Video Coding (AVC)

In March 2003, H.264/AVC [6] was finalized by the Joint Video Team (JVT), which was formed by the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG). Several extensions of H.264/AVC were developed between 2003 and 2009, such as the Scalable Video Coding (SVC) extension [13]. H.264 increases coding efficiency and is flexible for use over a broad variety of network types and application domains [14]. Its video coding layer uses the conventional block-based motion-compensated hybrid coding concepts with four important differences from its predecessors: 1) enhanced motion-prediction capability; 2) use of a small block-size exact-match transform; 3) adaptive in-loop deblocking filter; 4) enhanced entropy coding methods, which can achieve up to approximately 50% bit rate savings for equivalent

perceptual quality relative to the performance of prior standards [14]. H.264/AVC supports video resolution up to 8K UHD.

High Efficiency Video Coding (H.265)

High Efficiency Video Coding (HEVC) [7] is the successor of H.264/AVC and it is jointly standardized by the Joint Collaborative Team on Video Coding (JCT-VC) formed by the ITU-T VCEG and ISO/IEC MPEG standardization organizations. It was finalized in 2013. HEVC is also known as H.265 and MPEG-H Part 2. The main goal of HEVC is to enable significantly improved compression performance relative to existing standards—in the range of 50% bit-rate reduction for equal perceptual video quality [15]. In addition, it is designed to ease transport system integration and data loss resilience, as well as enable the implementation of parallel processing architectures [15]. HEVC addresses all applications of H.264/AVC with a focus on increased video resolution and increased use of parallel processing elements [15]. HEVC exploits temporal statistical dependencies via inter-picture prediction, spatial statistical dependencies via intra-picture prediction and transform coding of the residuals from prediction. The increased coding efficiency [16] yielded from the coding techniques comes at the cost of increased implementation complexity [17].

Versatile Video Coding (VVC)

Versatile Video Coding (VVC) [8] is the newest video coding standard finalized by the Joint Video Experts Team (JVET) of the ITU-T VCEG and ISO/IEC MPEG in July 2020. It is also known as H.266 and MPEG-I Part 3. VVC is designed to serve an ever-growing need for improved video compression as well as to support a wider variety of today's media content and emerging applications [18]. VVC achieves around 50% and 75% bit rate reduction over its predecessors HEVC and H.264/AVC, respectively [18]. VVC has a wide range of video applications, such as video with resolutions beyond standard- and high-definition, video with high dynamic range and wide color gamut, adaptive streaming with resolution changes, computer-generated and screen-captured video, ultra-low-delay streaming, 360° immersive video, and multi-layer coding [18].

AOMedia Video 1 (AV1)

AV1 is an open-source and royalty-free video coding standard developed by AOMedia. At the same video quality, AV1 achieves more than 30% bit rate reduction than its predecessor

VP9 [19]. The AV1 format is already supported by many web platforms, including Android, Chrome, Microsoft Edge, and Firefox. It has also been used by web-based video service providers, including YouTube, Netflix, Vimeo, and Bitmovin [19].

JPEG XS

The JPEG XS standard is designed for inter-operable, visually lossless low-latency (from less than one line to maximum 32 lines of image) lightweight image compression that allows for low-complexity real-time implementation in application-specific integrated circuit (ASIC), field-programmable gate array (FPGA), central processing unit (CPU), and graphics processing unit (GPU) [12]. JPEG XS targets the following applications: 1) transport over video links and IP networks, 2) real-time video storage, 3) frame buffer compression, 4) omnidirectional video capture and rendering, and 5) sensor compression [20].

VESA Display Stream Compression (DSC)

The VESA DSC [21, 22] video coding standard is widely accepted as a low-cost, visually lossless [23, 24] codec for use in display links [25]. DSC was first introduced in 2014 as version 1.0, which was deprecated due to a bug that causes buffer overflow and was updated to DSC version 1.1 in 2015. DSC version 1.1 supports 4:4:4 pixels of 8 and 10 bits per component (bpc). DSC version 1.2 was published in 2016 with support for larger component bit depths (14 and 16 bpc) and YCbCr 4:2:2 and 4:2:0 pixels in native mode. The latest version 1.2a was published in 2017 which fixed the bug for 4:2:0 pixels in version 1.2. DSC compresses video data to as low as 8 bits per pixel (bpp), which yields three times compression for 24-bit color. DSC has been widely adopted by publicly known standards such as the DisplayPort and Embedded DisplayPort of VESA, High-Definition Multimedia Interface (HDMI), and Display Serial Interface (DSI).

VESA Display Compression-M (VDC-M)

The VDC-M standard was introduced by VESA in 2018. It can achieve higher compression (6 bpp) than DSC with increased hardware cost. Similar to DSC, VDC-M also provides visually lossless compression quality and targets applications such as low-power mobile computing. VDC-M has been adopted by the MIPI Display Serial Interface 2 (DSI-2).

1.1.3 Compare DSC With Other Video Coding Standards

The VESA DSC video coding standard has four key attributes. First, DSC achieves visually lossless compression for various contents. Second, DSC uses a intra-frame, line-based coding algorithm, which results in very low latency for encoding and decoding. Third, DSC codecs can be implemented with low hardware costs. Since DSC performs intra-frame coding, it only requires a single picture line of buffering plus a small rate buffer, without the need of frame buffers. There is no computation intensive algorithms in DSC. Finally, DSC supports more color bit depths, including 8, 10, 12, 14, and 16 bpc.

Compared to the H.26x series of video coding standards, DSC achieves much lower hardware cost. The intra-only mode of these standards can be implemented with lower costs; however, they do not produce visually lossless quality for all contents. For example, the H.264/AVC intra-only mode can achieve visually lossless quality for most natural contents but is less effective with some content [25]. The screen content coding extension of HEVC (HEVC-SCC) [26] enhances the coding capability of HEVC for screen contents, but it has higher complexity compared to DSC [27]. The comparison between HEVC-SCC, VVC, AV1 and some other standards for their screen content coding capability is published [28].

Compared to Motion JPEG 2000, it does not achieve visually lossless quality for all types of content at 8 bpp with low hardware complexity for a real-time, high-throughput implementation [25]. The JPEG XS standard targets different set of applications which may not encounter certain types of content such as subpixel rendered text [25].

1.2 Research Motivation

Many applications require transmission of video data over display links, such as digital televisions, extended display of computers, automotive video systems, mobile devices (e.g. smartphone, tablet), and augmented reality (AR) and virtual reality (VR) headsets. The video stream in digital televisions and extended displays is transmitted using digital cables. For mobile devices, the video captured by camera is transmitted to the display using internal wires.

Traditionally, uncompressed video data are transmitted over display links, since it results in best visual quality and the bandwidth of the physical layer is sufficient. However, as ultra-high-definition (UHD) video resolutions such as 4K UHD (3840×2160), 8K UHD (7680×4320) and beyond

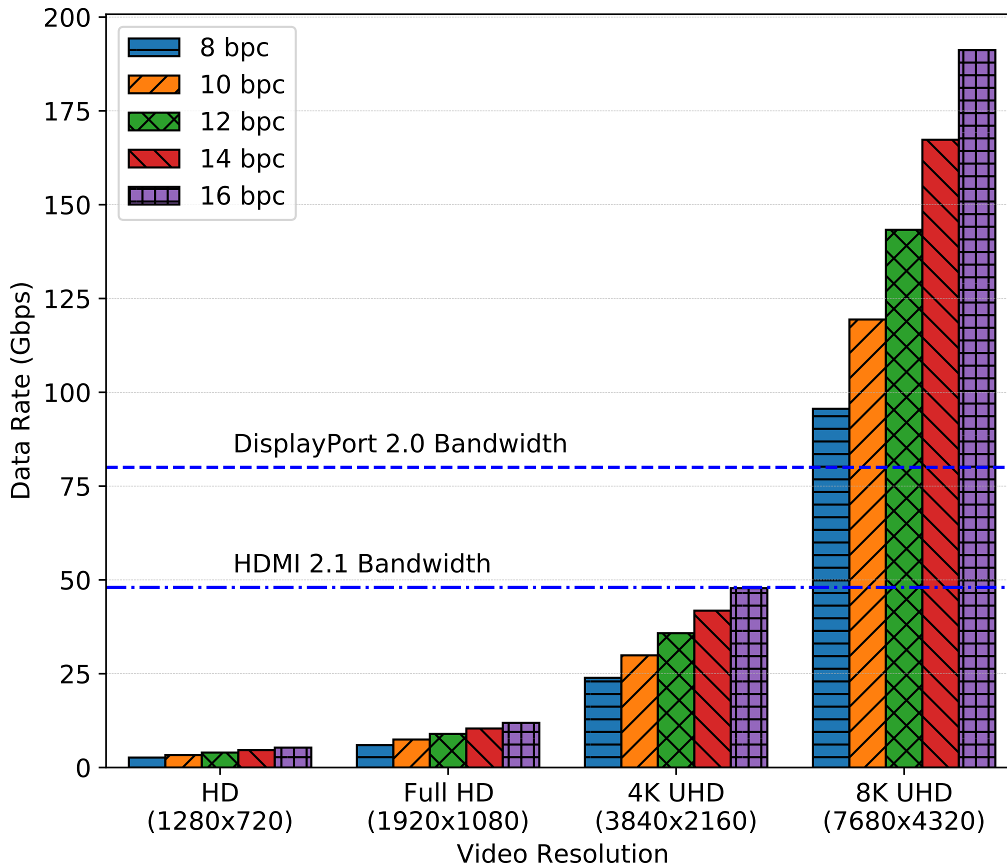


Figure 1.1: Uncompressed data rates for various video resolutions and component bit depths (8–16 bpc) at 120 frames per second.

are becoming prevalent (e.g., it is projected that 66% of connected flat-panel TV sets will be 4K [29]), the required data rate for transmitting uncompressed video dramatically increases. In addition, video frame rates such as 120 frames per second (fps) and beyond are more demanding. Moreover, new display technologies such as organic light emitting diode (OLED) and quantum dots have enabled wide color gamut (WCG) and high dynamic range (HDR) which motivates usage of higher color bit depth such as 10, 12, 14, and 16 bpc. As a result, the physical layer is no longer able to transmit raw video data. Figure 1.1 shows that the data rate increases exponentially with the increase of display resolution. Transmitting 8K UHD video at 120 fps requires 96 Gbps for 8 bpc and 191 Gbps for 16 bpc, which means two of the state-of-the-art display interfaces—DisplayPort (DP) 2.0 and High-Definition Multimedia Interface (HDMI) 2.1—are not able to support it.

To address the bandwidth disparity, one solution is to increase the physical layer bandwidth

by costly methods such as adding more wires and using more lanes. Although this approach can potentially fix the problem, there are several issues. First, adding more wires to display links can bring legacy compatibility issues. The existing devices may not be compatible with the new display link and thus huge costs will be incurred to upgrade the incompatible devices. Second, this solution increases power consumption. For mobile devices, higher power consumption means shorter battery life, which impacts user experience. Lastly, this approach is not scalable. The number of wires and lanes that can be added is limited. Even if it satisfies the current bandwidth demand, it will soon fall behind as the required data rate increases exponentially.

In contrast, a more compelling solution is to reduce the data rate by compressing the video data prior to transmission and decompressing them before display. As such, no hardware changes are needed to the current display links, and the pressure is alleviated without increasing power consumption. The Video Electronics Standards Association (VESA) has standardized the Display Stream Compression (DSC) [22,25] video coding standard to enable low-cost and low-latency hardware implementations of visually lossless video codecs over display links. DSC has been widely used in industry products. Specifically, DSC version 1.1 has been adopted by the Mobile Industry Processor Interface (MIPI) Alliance's Display Serial Interface (DSI) 1.2, DSI-2 1.0, and VESA Embedded DisplayPort (eDP) 1.4b; DSC version 1.2a has been adopted by HDMI 2.1, VESA DP 2.0, and VESA eDP 1.4b [30]. Thanks to the compression provided by DSC, HDMI 2.1 supports transmission of 4K at 120 fps and 8K at 60 fps.

There are some commercial DSC codecs available, such as the Hardent DSC IP cores [31]; however, the architecture details and performance results are not publicly available. A published DSC decoder design [32] focuses on optimizing the line buffer but does not include detailed architectures or post-layout results. Moreover, there are no existing work that demonstrates DSC codecs with measured data from silicon.

This dissertation investigates the design of DSC version 1.2a codecs (i.e., encoders and decoders) from the perspective of hardware architectural design, physical chip design, and fine-grained many-core software design. First, three hardware architectures are proposed for DSC encoders: a slice encoder architecture, a slice-interleaved encoder architecture, and a time-interleaved encoder architecture. A DSC encoder chip based on the time-interleaved architecture is designed and fabricated in TSMC 28 nm CMOS technology. Second, this dissertation discusses the design of

DSC decoders in two approaches: 1) application-specific integrated circuits (ASIC) design, and 2) software design on fine-grained multiple instruction multiple data (MIMD) many-core processor arrays. Four hardware architectures are proposed for the ASIC DSC decoders: a slice decoder architecture, a slice-interleaved decoder architecture, a parallel slice decoder architecture, and a parallel-interleaved decoder architecture. The many-core software design of the slice decoder and parallel slice decoders are discussed.

The proposed slice-interleaved encoder architecture, time-interleaved encoder architecture, and the slice-interleaved decoder architecture share computational logic and replicate only registers for different slices, which results in reduced hardware cost. The time-interleaved encoder architecture achieves increased throughput by utilizing idle cycles to process different slices. The parallel slice decoders achieve linearly increased throughput using replicated slice decoders which yields linearly increased area cost. The many-core software decoders demonstrate that fine-grained MIMD many-core computation platforms can achieve significant better performance and energy efficiency than coarse-grained general purpose processors. The ASIC decoders outperform the many-core software decoders in both performance and energy efficiency.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides a comprehensive overview of the VESA Display Stream Compression (DSC) video compression standard, focusing on the encoding and decoding algorithm. Chapter 3 presents three hardware architectures for DSC encoders. Chapter 4 discusses the design and measured results of a fabricated DSC encoder chip that supports real-time processing of 4K video resolution. Chapter 5 proposes four hardware DSC decoder architectures and physical design of six DSC decoders that are implemented based on the proposed architectures. Chapter 6 discusses the software design of DSC decoders for fine-grained many-core processor arrays. Chapter 7 concludes the dissertation and proposes future work.

Chapter 2

Display Stream Compression (DSC) Standard

This chapter overviews the VESA Display Stream Compression (DSC) standard [22], focusing on the system-level usage, concepts, and coding algorithms. A thorough analysis of the encoding and decoding processes is provided.

2.1 Overview

Figure 2.1 illustrates DSC usage in end-to-end systems, which consists of a source device, a display link, and a sink device. In the source device, an image source captures raw video data and sends them to the DSC encoder via an internal connection. The DSC encoder compresses the video data into a bitstream and sends it to the sink device over the display link. In the sink device, the DSC decoder receives the bitstream and decompresses it into raster-scan-ordered pixels, which are sent to a display through an internal connection. The display link can be either wired or wireless. Examples of the wired display links are MIPI Alliance's Display Serial Interface (DSI) Specification, DisplayPort (DP), Embedded DisplayPort (eDP), and High-Definition Multimedia Interface (HDMI).

DSC uses a line-based intra-picture video coding algorithm, which requires only current line and previous line reconstructed pixels, resulting in very low encoding and decoding latency. The main storage of DSC consists of a line buffer that stores one picture line of pixels, and a rate buffer that converts varying number of bits into constant bit rate. Unlike many other video coding

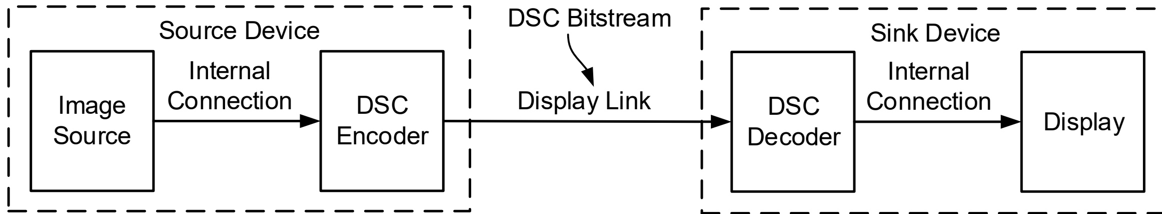


Figure 2.1: DSC usage in end-to-end systems [22].

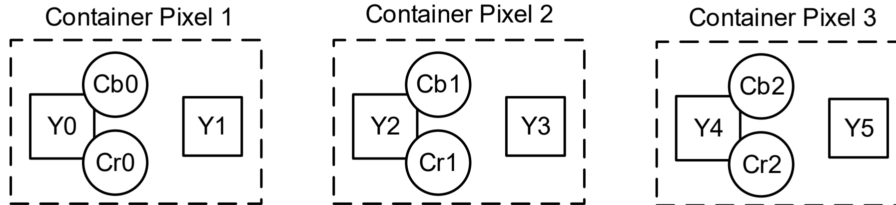


Figure 2.2: YCbCr 4:2:2 pixels packed into virtual 4:4:4:4 containers in native 4:2:2 mode. Odd-position luma (Y1, Y3, Y5) is treated as the fourth independent component in a container pixel.

standards, no off-chip memory is needed in DSC. As a result, DSC codecs can be implemented with very low hardware cost.

DSC version 1.1 supports 8, 10, and 12 bits per component (bpc); DSC versions 1.2 and 1.2a also support 14 and 16 bpc. All DSC versions support 4:4:4 pixels in both RGB and YCbCr formats. A YCbCr 4:4:4 pixel contains three *components*: one luma component (Y) and two chroma components (Cb and Cr). DSC version 1.1 supports YCbCr 4:2:2 pixel in *simple* mode only, where 4:2:2 pixels are upsampled to 4:4:4 format before encoding and downsampled to 4:2:2 after decoding. DSC versions 1.2 and 1.2a also support YCbCr 4:2:2 in *native* mode, in which two neighboring pixels are packed into a virtual, half-width 4:4:4:4 container and the even- and odd-position luma are treated as independent components, as shown in Figure 2.2. DSC version 1.2a adds support for YCbCr 4:2:0 pixel format in *native 4:2:0* mode, in which two neighboring pixels are packed into a virtual, half-width 4:4:4 container the even- and odd-position luma are treated as independent components, as shown in Figure 2.3. The width of pictures and slices are effectively halved in native 4:2:0 and 4:2:2 modes.

In DSC, a *picture* is defined as a frame for progressive format videos or a field for interlaced format videos. A picture is partitioned into one or more non-overlap rectangular identical-sized regions called *slices*, where each slice is independently coded. The number of columns of slices per

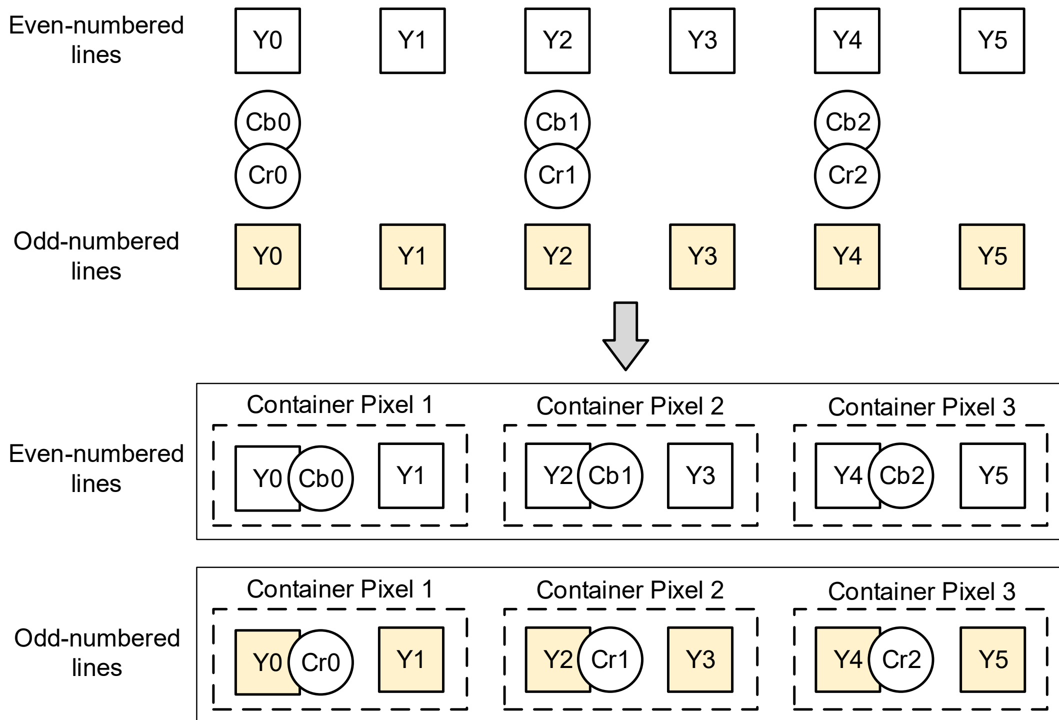


Figure 2.3: YCbCr 4:2:0 pixels packed into virtual 4:4:4 containers in native 4:2:0 mode. Odd-position luma (Y1, Y3, Y5) is treated as a independent component in the container pixel. Even- and odd-numbered lines contain Cb and Cr in the chroma, respectively.

picture is called the number of *slices per line*. A slice consists of a set of *groups*, each of which is consecutive (container) pixels of the same slice line. If slice width is not evenly divisible by three, the last group of each slice line contains fewer than three pixels and is called a *partial group*. Figure 2.4 shows some example slice configurations of a picture. In case the picture width is not evenly divisible by slice width, the rightmost column of slices extends beyond the right boundary of the picture and the value of the last pixel of each slice line is replicated for pixels outside the picture boundary. If the picture height is not evenly divisible by the slice height, the bottom rows of slices extend beyond the bottom of the picture, and midpoint values of each component are used for these regions. To minimize the amount of extra data to be sent over display links, it is desirable to choose configurations for which the picture width and height are evenly divisible by slice width and height, respectively.

The DSC algorithm is designed to allow encoders and decoders to be implemented with a throughput of one and three pixels per cycle, respectively. The throughput per pixel in native 4:2:0 and 4:2:2 modes are effectively doubled compared to simple 4:4:4 mode, since two pixels are

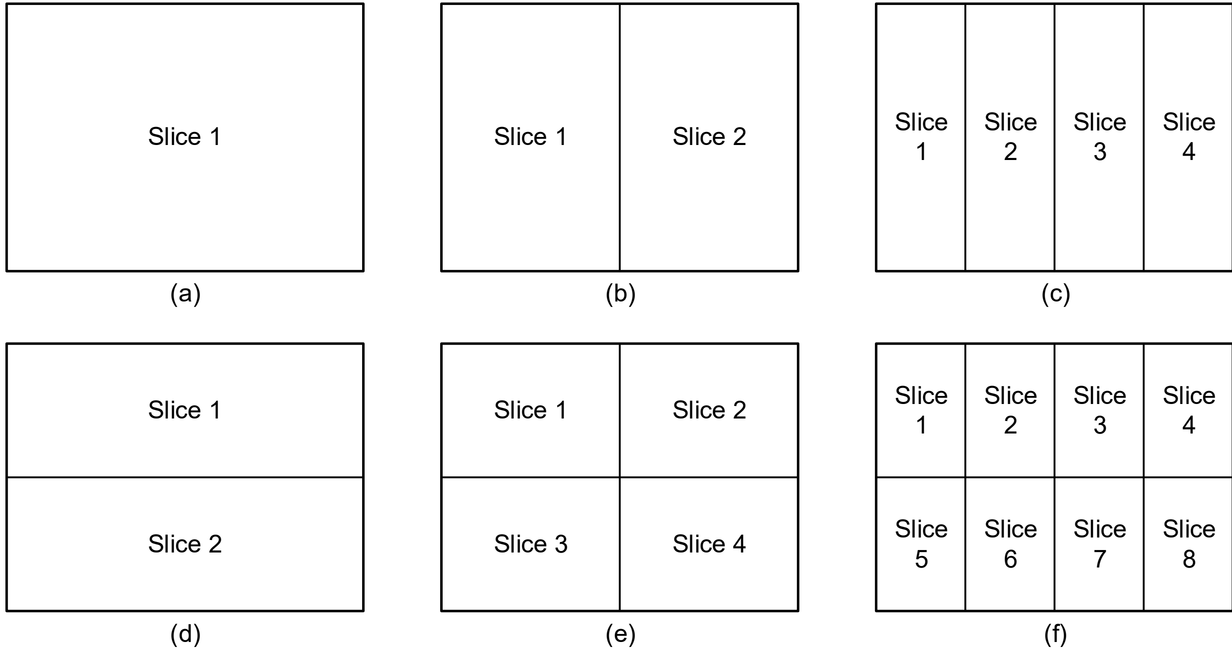


Figure 2.4: Example of pictures configured into: (a) one row and one column of slice; (b) one row and two columns of slices; (c) one row and four columns of slices; (d) two rows and one column of slices; (e) two rows and two columns of slices; (f) two rows and four columns of slices.

packed into one container. To achieve higher throughput, the DSC algorithm is easily parallelized by configuring pictures to more than one slice per line and processing each column of slices of a picture (Figure 2.4) with a separate instance of an encoder or decoder.

DSC uses a 128-byte picture parameter set (PPS) to store configuration information. PPS contains the information of DSC version, pixel format and color bit depth, specified bit rate, rate control parameters, and so on. Identical PPS should be used in the encoder and the decoder to ensure correct decoding; therefore, the encoder must communicate PPS to the decoder.

2.2 DSC Encoding Algorithm

2.2.1 DSC Encoding Process

Figure 2.5 illustrates the DSC encoding process. RGB input image data are converted to reversible YCoCg (YCoCg-R) [33] format before further encoding. A pixel buffer stores a small number of original pixels for look-ahead flatness determination. The encoding process is performed on a group basis, and every group is coded in either *predictive coding mode (P-mode)* or *indexed*

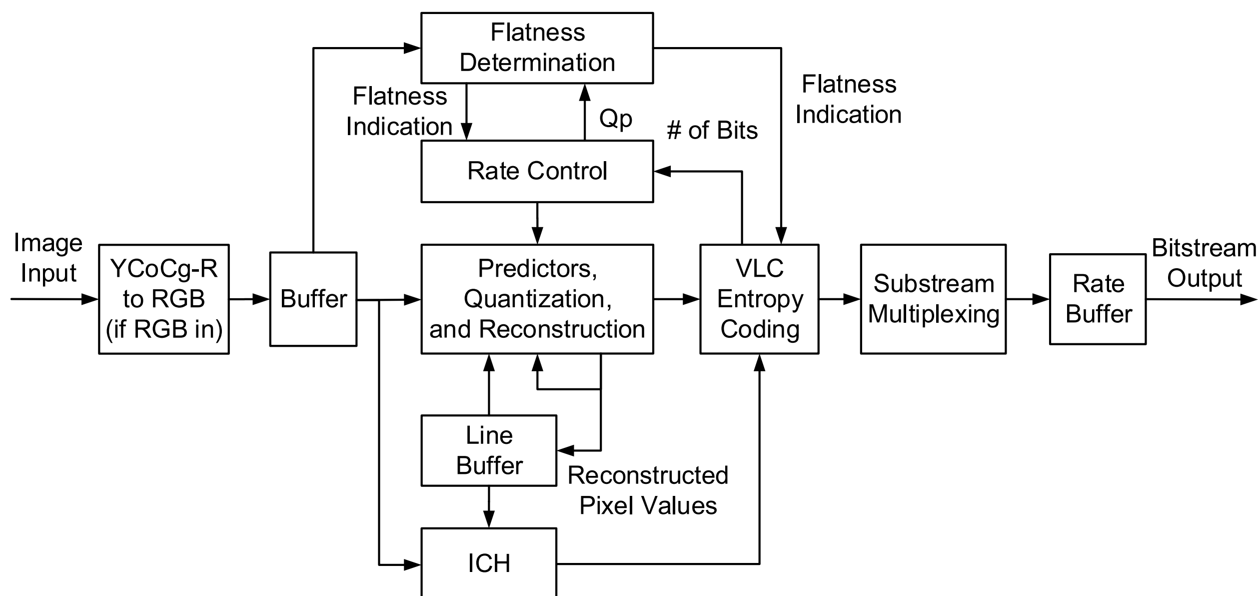


Figure 2.5: DSC encoding process [22].

color history (ICH) coding mode (ICH-mode). For P-mode-coded groups, the samples are predicted with one of three predictors: modified median-adaptive prediction (MMAP), block prediction (BP), and midpoint prediction (MPP). Residuals, each of which is calculated by subtracting the predicted value from the original value, are quantized before being coded in entropy encoder. Reconstruction is performed by adding the inverse quantized residual to the predicted value. ICH keeps a record of 32 recently reconstructed pixels, each of which is addressed by a 5-bit index. If ICH-mode is selected, the best-matched ICH pixel is selected for each pixel in current group, and the three indices of the selected entries are coded in entropy encoder. The final reconstructed pixels are stored in the line buffer and used for MMAP and ICH in the next slice line. The variable length coding (VLC) entropy encoding codes P-mode groups using delta size unit-variable length coding (DSU-VLC) scheme, and ICH-mode groups using ICH coding with a special escape code. Rate control algorithm manages the rate buffer fullness and dynamically adjusts the quantization parameter (Qp) in every group to maximize perceived quality. If the upcoming pixels are flat, the Qp is dropped quickly. Substream multiplexing combines the three or four substreams into a single bitstream. The rate buffer converts a variable number of bits input which are used to code each group into a constant rate bitstream output, which is specified as bits per pixel (bpp) in the PPS.

2.2.2 Color Space Conversion

DSC can accept both RGB or YCbCr color space inputs. RGB pixels are converted into YCoCg-R format, where chroma (Co and Cg) use one more bit than luma (Y) except in 16 bpc, where the least significant bit (LSB) is rounded off. YCbCr inputs are accepted without color space conversion; Cb and Cr are mapped to Co and Cg labels, respectively; Cb and Cr has the same bit depth as Y. The conversion is defined as follows [22]:

$$cscCo = R - B \tag{2.1}$$

$$t = B + (cscCo >> 1) \tag{2.2}$$

$$cscCg = G - t \tag{2.3}$$

$$Y = t + (cscCg >> 1) \tag{2.4}$$

Where:

- t is a temporary value.
- $cscCo$ and $cscCg$ are the intermediate values for Co and Cg, respectively.
- Y is value of the luma component.

The Co and Cg values are centered around the midpoint for 8, 10, 12, or 14 bpc:

$$Co = cscCo + (1 << bits_per_component) \tag{2.5}$$

$$Cg = cscCg + (1 << bits_per_component) \tag{2.6}$$

For 16 bpc, Co and Cg are rounded to 16 bits before centering around the midpoint:

$$Co = MIN(\text{0xFFFF}, ((cscCo + 1) >> 1) + \text{0x8000} \tag{2.7}$$

$$Cg = MIN(\text{0xFFFF}, ((cscCg + 1) >> 1) + \text{0x8000} \tag{2.8}$$

2.2.3 Prediction, Quantization, and Reconstruction

DSC uses three predictors: modified median-adaptive prediction (MMAP), block prediction (BP), and midpoint prediction (MPP), all of which are required for encoders. Every component of a group is predicted using one of the predictors. However, encoders can choose to disable BP if the decoder does not support it, or BP does not benefit the coding contents. In native 4:2:0 mode, BP applies to only luma components, and chroma components always select MMAP or MPP. In predictive coding (P-mode), a predicted value is calculated for each sample using the one of the three predictors, which is subtracted from the original sample value to produce the residual. The

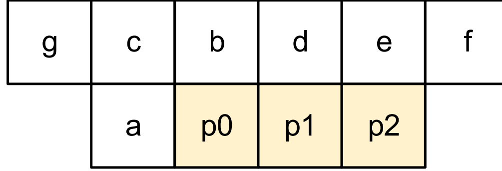


Figure 2.6: Pixels used in MMAP of current group pixels $(p0, p1, p2)$, modified from [22, Figure 6-1].

residuals are quantized and entropy-coded in the bitstream. Reconstruction produces the same reference pixels for prediction and ICH in encoders and decoders.

Modified Median-Adaptive Prediction (MMAP)

MMAP is the default prediction method in DSC. It predicts the current group using the reconstructed pixel values from the previous group and previous line that is above the current group. Figure 2.6 shows the pixels that are used in MMAP for the current group.

A horizontal low-pass filter [0.25, 0.5, 0.25] is applied to the previous line samples (c, b, d, e in Figure 2.6) and produces filtered values ($filter_c$, $filter_b$, $filter_d$, and $filter_e$). For example,

$$filter_c = (g + 2 * c + b + 2) >> 2 \tag{2.9}$$

In the first group of each slice line, g and c are outside the left boundary of the slice and both of them use the same value of pixel b. For the last group of each slice line, pixel f is outside the right boundary of the slice and it uses the value of pixel e.

The filtered values are blended with the original pixel values and produce blended values ($\tilde{c}, \tilde{b}, \tilde{d}$, and $\tilde{e}$). In the first group of each slice line, a and $\tilde{c}$ are set to the midpoint of the component. For example,

$$\tilde{c} = c + CLAMP(filter_c - c, -((1 << qlevel)/2), ((1 << qlevel)/2)) \tag{2.10}$$

where:

- $qlevel$ is the quantization level
- $CLAMP$ function is defined as:

$$CLAMP(x, min, max) = \begin{cases} min, & \text{if } x < min. \\ x, & \text{if } min \le x \le max. \\ max, & \text{if } x > max. \end{cases} \tag{2.11}$$

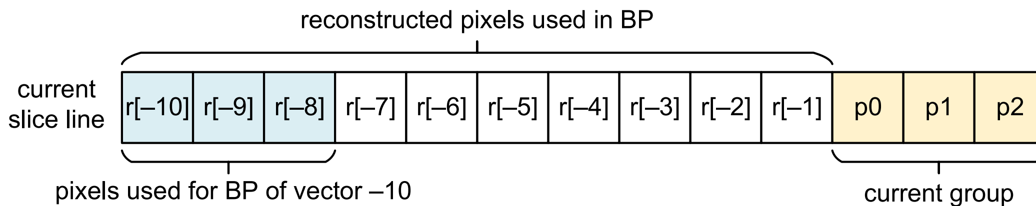


Figure 2.7: Block prediction example with BP vector of -10. The three pixels in current group $p0$, $p1$, and $p2$ are predicted as $r[-10], r[-9]$, and $r[-8]$, respectively.

The three pixels are predicted as:

$$p0 = CLAMP(a + \tilde{b} - \tilde{c},\ MIN(a, \tilde{b}),\ MAX(a, \tilde{b})) \tag{2.12}$$

$$p1 = CLAMP(a + \tilde{d} - \tilde{c} + R0,\ MIN(a, \tilde{b}, \tilde{d}), MAX(a, \tilde{b}, \tilde{d})) \tag{2.13}$$

$$p2 = CLAMP(a + \tilde{e} - \tilde{c} + R0 + R1,\ MIN(a, \tilde{b}, \tilde{d}, \tilde{e}), MAX(a, \tilde{b}, \tilde{d}, \tilde{e})) \tag{2.14}$$

where:

- $R0$ is the inverse quantized residual of the first sample in the group
- $R1$ is the inverse quantized residual of the second sample in the group
- MIN and MAX return the minimum and maximum values, respectively

For the first line (and the second line of chroma in native 4:2:0 mode) of a slice, there are no previous line pixels and the samples are predicted as follows (*cpnt_bit_depth* is the bit depth of the component and $((1 << \mathit{cpnt_bit_depth}) - 1)$ is the maximum value of the component).

$$p0 = a \tag{2.15}$$

$$p1 = CLAMP(a + R0,\ 0, ((1 << \mathit{cpnt_bit_depth}) - 1)) \tag{2.16}$$

$$p2 = CLAMP(a + R0 + R1,\ 0,\ ((1 << \mathit{cpnt_bit_depth}) - 1)) \tag{2.17}$$

The MMAP of $p1$ and $p2$ depends on the inverse quantized residual of pixels to their left, i.e., $p0$ and $p1$, respectively. Therefore, MMAP of $p1$ and $p2$ must wait until the prediction, quantization, and inverse quantization of $p0$ and $p1$ finish, respectively. In general, this data dependency limits the throughput of DSC encoders to not more than one pixel per cycle.

Block Prediction

Block prediction (BP) predicts the samples of the current group using reconstructed pixel values of previous groups in the current slice line. The offset from current sample to the pixel used for BP is called the *BP vector*, which is between -3 and -10; therefore, the three pixels within a

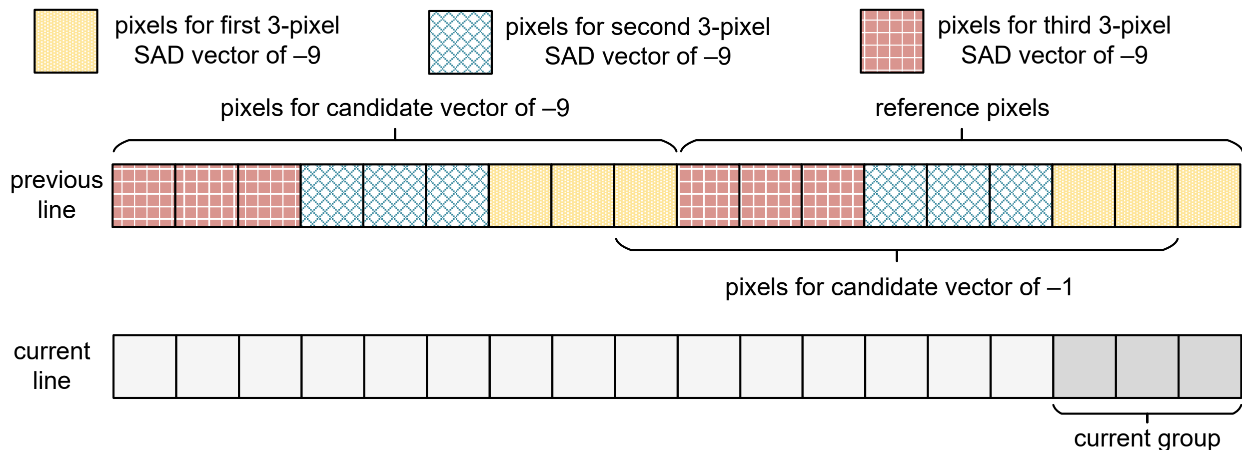


Figure 2.8: Pixels used in 3-pixel and 9-pixel sum of absolute differences (SAD) calculation for BP search. Current line pixels are not used in this calculation.

group can be predicted in parallel. Figure 2.7 shows an example of block prediction with BP vector of -10. BP helps improve the rendered quality of repeated patterns such as zone plates [25]. The selection of BP is for all pixels of all components in the group, except that BP in native 4:2:0 mode applies only to luma components, and chroma always select MMAP over BP.

Midpoint Prediction

The purpose of Midpoint prediction (MPP) is to limit the size of the quantized residuals to no larger than the quantized sample depth, i.e., the difference between the component bit depth ($cpnt_bit_depth$) and quantization level ($qlevel$). MPP predicts the sample as a value around the midpoint of the component, in which the $qlevel$ number of LSBs are copied from the reconstructed value of the rightmost pixel in the previous group ($prev_recon$).

$$p = (1 << (cpnt_bit_depth - 1)) + (prev_recon \mathbin{\&} ((1 << qlevel) - 1)) \tag{2.18}$$

Prediction Mode Selection

The decision between MMAP and BP is made once per group and only uses the reconstructed pixel values from the previous line, which means the decision can be made by up to one slice line pixel time in advance.

To determine the BP vector, the encoder calculates 9-pixel sum of absolute differences (SAD) between the reference pixels and the pixels to the left, in which nine candidate vectors

$(-1, -3, -4, -5, -6, -7, -8, -9, -10)$ are considered. Figure 2.8 shows the pixels used for calculating the 3-pixel SADs for candidate vector of -9. The 9-pixel SAD is calculated as the sum of three 3-pixel SADs, which are calculated by adding the 6-bit modified absolute differences between reference samples and the samples in the candidate vectors. In native 4:2:0 mode, only luma samples are included in the sum, whereas all samples are included in the sum for other modes. The modified absolute difference ($modified_abs_diff$) is computed by reducing the bit width of the absolute difference (abs_diff) to six bits.

$$modified_abs_diff = MIN(abs_diff >> (cpnt_bit_depth - 7), \text{0x3F}) \tag{2.19}$$

Three 10-bit 3-pixel SADs are clamped to nine bits and added together to form a 11-bit 9-pixel SAD, three LSBs of which are truncated. The vector with the smallest SAD is used as the BP vector. If BP vector is -1, MMAP is selected for this group and the BP counter is reset to zero; otherwise BP counter increases by one if the current group is not the first three groups of the slice line. Block prediction is selected if current group is not a partial group, and BP counter is at least equal to three, and less than three pixels have passed since an edge occurred.

The size of quantized residuals of BP/MMAP are calculated. If the maximum size is greater than or equal to the quantized sample depth, MPP is used for the component. In addition, the encoder is forced to select MPP to produce a minimum data rate that avoids rate buffer underflow.

Quantization, Inverse Quantization, and Reconstruction

The residual ($residual$) is produced by subtracting the predicted sample value ($pred_sample$) from the original sample value ($orig_sample$). Quantized residuals ($quantized_residual$) are controlled by the quantization level ($qlevel$) and are calculated as follows:

$$residual = orig_sample - pred_sample \tag{2.20}$$

$$round = MAX(0, (1 << qlevel)/2 - 1) \tag{2.21}$$

$$quantized_residual = \begin{cases} (residual + round) >> qlevel, & \text{if } residual \geq 0. \\ -((-residual + round) >> qlevel), & \text{if } residual < 0. \end{cases} \tag{2.22}$$

Inverse quantization is performed by shifting the quantized residual ($quantized_residual$) to the left by quantization level ($qlevel$) number of bits:

$$inverse_quantized_residual = quantized_residual << qlevel \tag{2.23}$$

Reconstructed samples are calculated by adding the inverse quantized residual to the pre-

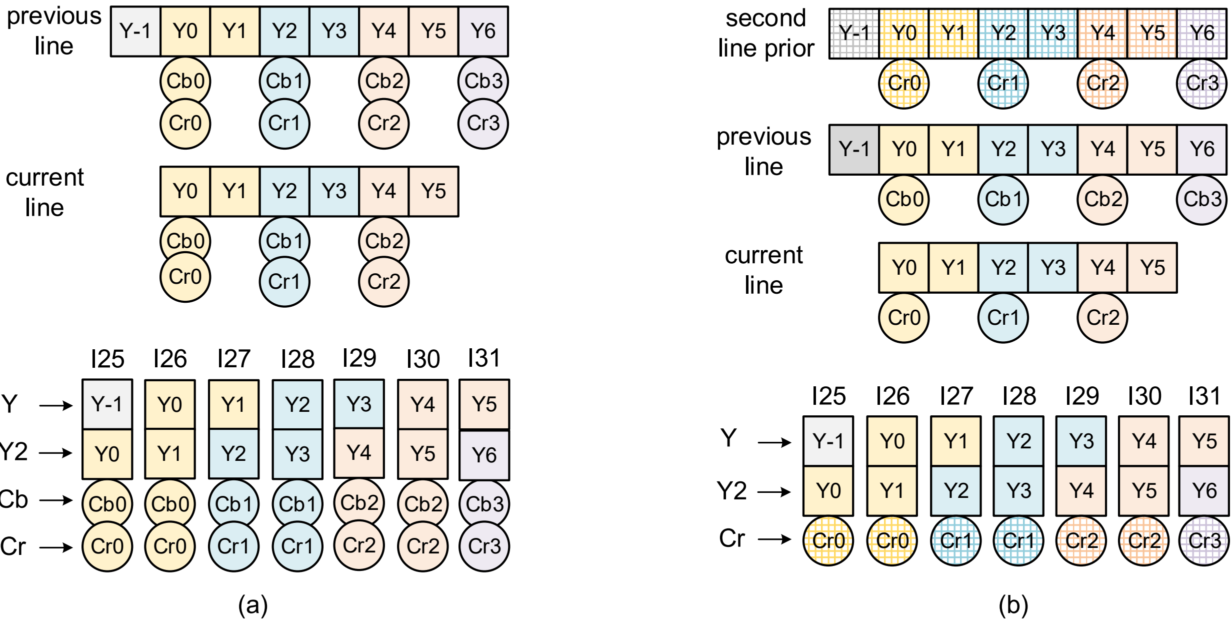


Figure 2.9: ICH entries pointing to previous line. (a) Native 4:2:2 mode. (b) Native 4:2:0 mode.

dicted value ($pred_sample$), and bound to zero and maximum value of the component (max_value).

$$recon_sample = CLAMP(pred_sample + inverse_quantized_residual, 0, max_value) \quad (2.24)$$

2.2.4 Indexed Color History (ICH)

ICH maintains 32 entries of recently reconstructed pixels (container pixels in native modes) from predictive coding. All 32 entries are actual history pixels for the first line (and second line in 4:2:0 mode) of each slice; otherwise, only the first 25 entries are actual history pixels and the last seven entries point to reconstructed pixel values in the previous line. In 4:4:4 mode, the seven ICH pixels in the previous line are centered around the horizontal position of current group. Figure 2.9 shows the ICH entries pointing to the previous line in native 4:2:2 and 4:2:0 modes. ICH is helpful in coding contents that have similar pixel values appearing in close positions, such as computer-generated text and graphics. Figure 2.10 shows the ICH in the encoder, which consists of three steps: 1) search for the best ICH entries; 2) decide between ICH-mode and P-mode; and 3) update the ICH entries.

In every group, the encoder searches over the 32 ICH entries and finds the best match for each pixel of the group based on smallest weighted SAD between the original pixels and ICH pixels

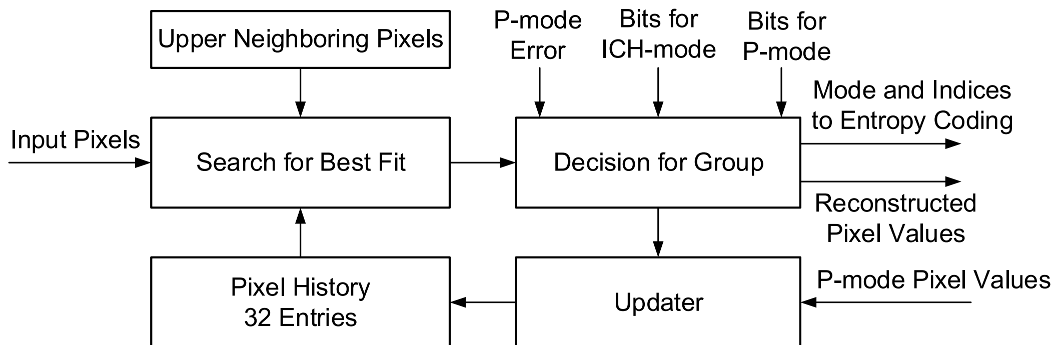


Figure 2.10: Indexed color history in the DSC encoder [22].

over all components. The weight (w_c) is equal to two for luma and one for chroma.

$$weighted_sad = \sum_c w_c * |orig_c - ich_c| \tag{2.25}$$

In case multiple indices have the smallest weighted SAD, the smallest index is selected for the pixel.

After the best ICH entries have been found for each pixel, the encoder decides between ICH-mode and P-mode, in which two factors are considered for ICH-mode to be selected: 1) for each pixel in the group, at least one ICH entry should exist such that the absolute difference between the pixel and the ICH entry of every component is within a threshold; 2) the rate-distortion (RD) cost of ICH-mode should be smaller than P-mode and and distortion of ICH-mode should also be less than P-mode if the next group is very flat. RD is the sum of the $log2$ of the maximum component-wise errors, whereas the total cost for ICH- and P-mode are the sum of the total number of bits used in that mode and the value of RD multiply by four.

With the coding mode selected, the ICH is updated at the end of the group time. Figure 2.11 shows an example of updating the ICH records in both P-mode and ICH-mode. In P-mode, the reconstructed pixels of the current group, denoted as P0, P1, and P2 from left to right, enter the top of the ICH. All other entries are shifted down by three, as shown in Figure 2.11(a). In ICH-mode-coded groups, the selected ICH entries are moved to the top of the ICH. In Figure 2.11(b), the selected indices are 1, 3, and 5 for the left, middle, and right pixels, respectively. Therefore, I5 becomes the MRU, while I3 is moved to the second entry and I1 is moved to the third entry. The original entries I0, I2, and I4 are shifted down by three, two, and one, respectively, while other entries remain unchanged. When the number of unique indices is less than three, as shown in Figure 2.11(c–d), only the last occurrence of a replicated index is counted in ICH update.

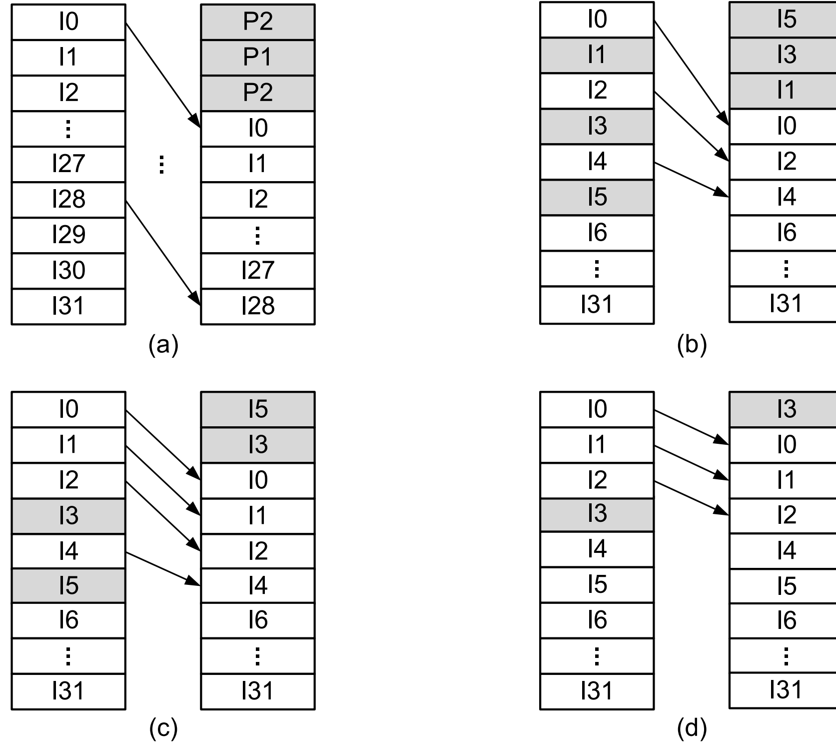


Figure 2.11: ICH update examples. (a) P-mode. (b) ICH-mode with indices (1, 3, 5). (c) ICH-mode with indices (5, 3, 5). (d) ICH-mode with indices (3, 3, 3).

2.2.5 Line Buffer

Line buffer stores the reconstructed pixel values of the previous line, which are used in ICH, BP search, and MMAP. The storage is one slice line per column of slices; therefore, the total storage requirement is one picture line of pixels, regardless of the number of slices per line.

In DSC, the decoder might use a bit depth that is smaller than the component's bit depth ($cpnt_bit_depth$) to reduce implementation cost. In this case, the decoder should communicate the maximum bit depth that it supports and the encoder should configure the $linebuf_depth$ in the PPS accordingly. The bit-width of the stored samples values ($stored_sample$) in the line buffer is reduced as follows:

$$shift = MAX(0, cpnt_bit_depth - linebuf_depth) \tag{2.26}$$

$$round = (shift > 0)\ ?\ (1 << shift - 1)) : 0 \tag{2.27}$$

$$stored_sample = MIN((sample + round) >> shift,\ (1 << linebuf_depth) - 1) \tag{2.28}$$

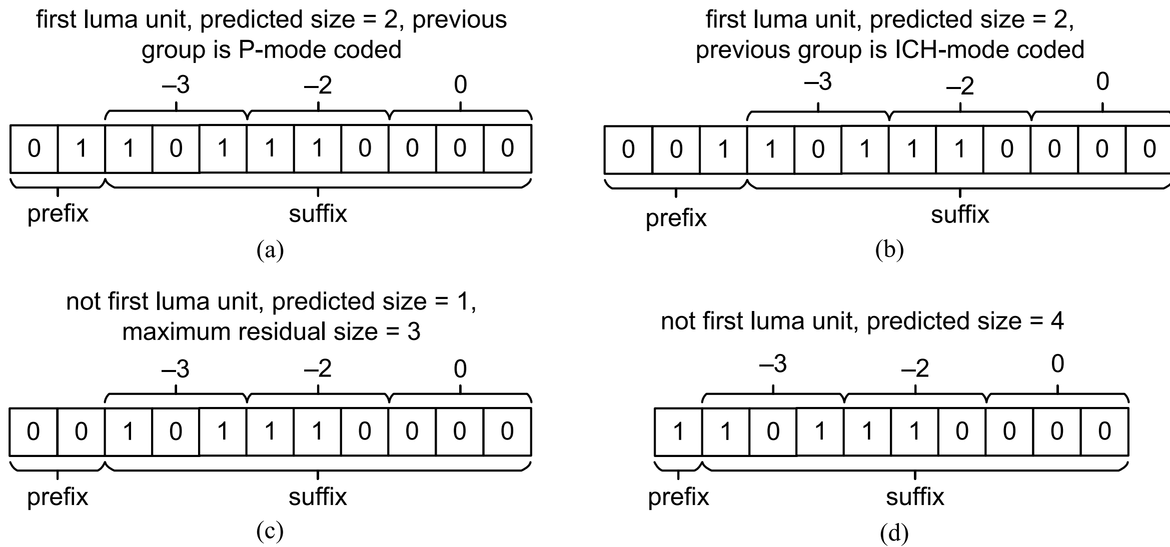


Figure 2.12: Examples of DSU-VLC codes with quantized residuals of $(-3, -2, 0)$ coded in the suffix using 2's complement. (a) The first luma unit and prefix indicates size increase of one. (b) The first luma unit and prefix uses two "0" bits to indicate size increase of one since the previous group is coded with ICH-mode. (c) Not first luma unit, the trailing "1" bit is omitted since the maximum residual size of the group is equal to maximum residual size of the component. (d) Not first luma unit, the predicted size is larger than the actual size of the quantized residuals, a single "1" bit in the prefix indicates no change in size.

The sample value that is read from the line buffer should be shifted to the left.

$$read_sample = stored_sample << shift \tag{2.29}$$

2.2.6 VLC Entropy Encoding

DSC uses delta size unit-variable length coding (DSU-VLC) for groups coded with P-mode, and indexed color history coding for ICH-mode-coded groups. The bits used to code each component of a group form a *syntax element*.

For every component of P-mode-coded groups, a residual size prediction and adjustment is performed using the change of quantization level from previous group to current group, and the size of quantized residuals ($required_size$) of the same component from the most-recent P-mode group.

$$predicted_size = (required_size[0] + required_size[1] + 2 * required_size[2] + 2) >> 2 \tag{2.30}$$

$$adj_predicted_size = CLAMP(predicted_size - qlevel_change, 0, cpnt_bit_depth - qlevel) \tag{2.31}$$

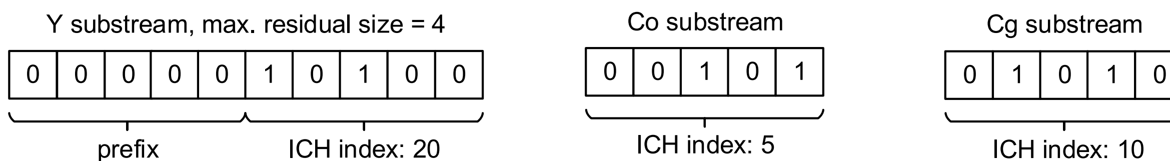


Figure 2.13: Example of an ICH coding for a group selecting ICH-mode with indices (20, 5, 10) for the leftmost, middle, and right pixels, respectively.

DSU-VLC for each unit includes a prefix which indicates the residual size, and a suffix that codes three quantized residuals. The prefix coding is a unary code which contains a number of "0" with a trailing "1" bit. The number of "0" bits indicates the size increase from the adjusted predicted size to the maximum residual size of the unit being coded in the current group. There are two cases that require modification: 1) if the previous group is coded with ICH-mode, a single "1" bit indicates ICH-mode is used again, "01" means no change in size, and "001" means size increase of one, etc; 2) when coding Y2, Co, and Cg units with maximum size, the trailing "1" bit is omitted since the decoder can infer it. Figure 2.12 shows four examples of DSU-VLC codes. The shortest DSU-VLC code for one unit is one bit, which contains a single "1" bit of prefix and no bits for suffix (all quantized residuals are zeros).

Indexed color history (ICH) coding uses a special escape code on the prefix of the first luma unit and no prefix for other units. If the previous group is coded in P-mode, the escape code indicates a prefix size that is one larger than the maximum length in P-mode, and the trailing "1" is omitted since the decoder can infer it. If the previous group is coded in ICH-mode, the prefix in the first luma unit contains a single bit of "1", which indicates ICH-mode is used again for the current group. The 5-bit ICH index for the leftmost pixel in the group is coded in the Y2 substream in native 4:2:2 mode and in the Y substream in other modes. The 5-bit ICH index for the middle and rightmost pixels in the group are coded in the Co and Cg substream, respectively. In ICH coding, a group requires at least 16 bits to code, which results in minimum bit rate of 5.333 bits per pixel. Figure 2.13 shows an example of ICH coding for a group not in native 4:2:2 mode.

2.2.7 Rate Control

The rate control (RC) uses an idealized rate buffer model to manage the rate buffer fullness, and dynamically adjusts the quantization level to maximize the perceived quality. It consists of a

buffer level tracker, linear transformation, long-term parameter selection, short-term quantization parameter (Qp) adjustment, and flatness Qp overrides.

The buffer level tracker updates the original fullness of the RC buffer model. In every group, the number of bits used to code the group, which is provided by entropy encoding, is added to the buffer fullness. In addition, after the initial transmission delay, the number of bits to be transmitted per group is subtracted from the buffer fullness. In the last group of a chunk being processed, the buffer level tracker determines the number of adjustment bits, which is between zero and eight, to make the chunk size an integer number of bytes. In constant bit rate (CBR) mode, it signals the usage of MPP to guarantee a minimum bit rate if underflow could occur in the next group. In variable bit rate mode (VBR), buffer fullness is clamped to zero if the value will be less than zero; the correction amount is accumulated and is used to determine the actual chunk size.

Linear transformation adjusts the original buffer fullness ($buffer_fullness$) to a RC model fullness ($rc_model_fullness$) which is a negative number and a value of zero means full.

$$rc_model_fullness = (rc_xform_scale * (buffer_fullness + rc_xform_offset)) >> 3 \qquad (2.32)$$

The offset value (rc_xform_offset) is modified in every group. It is designed to maintain constant quality during the initial delay, allocate extra bits for the first slice line (and also second slice line in native 4:2:0 mode), and ensure the slice is coded with the correct number of bits. The scale factor (rc_xform_scale) is used at the beginning and end of slice to maintain the RC model's usable range.

Long-term parameter selection uses 14 thresholds to define 15 ranges. The modified buffer fullness is compared against the thresholds and classified into one of the ranges. Each range has a minimum and a maximum quantization parameter, as well as an offset used for adjusting the targeted bits per group, all of which are used in short-term quantization parameter adjustment.

Short-term quantization parameter adjustment uses the parameters selected in long-term parameter selection, information provided in entropy encoding (i.e., predicted size, number of bits used to code the previous group), and quantization parameters of the previous two groups to make final adjustments of Qp for the next group.

The encoder performs flatness checks for the original pixels of each group in the upcoming *supergroup*, which is a set of four consecutive groups. Two flatness checks which use the current group pixels and the rightmost pixel in the previous group (first check) or the three pixels in the next group (second check) determines whether the group is "very flat" or "somewhat flat" by comparing the

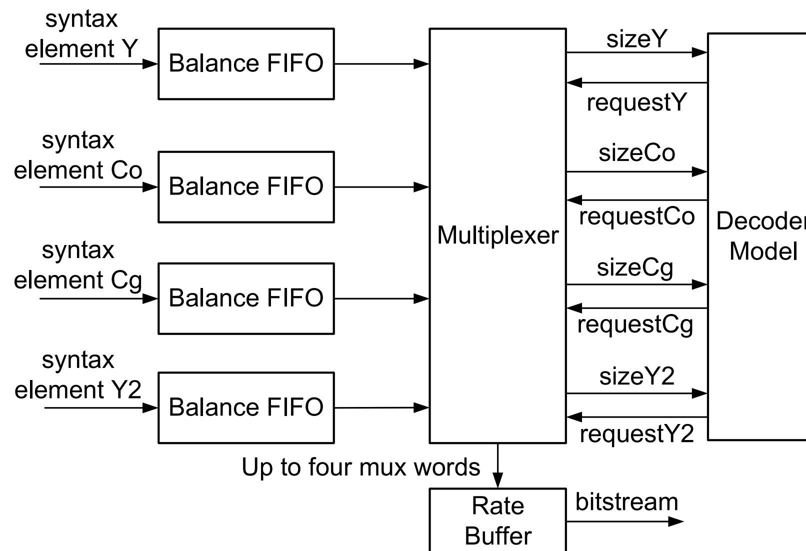


Figure 2.14: Substream multiplexer and rate buffer in the encoder, modified from [22, Figure 6-11]

differences of the maximum and minimum sample values with two thresholds. When transitioning from non-flat to flat, Qp is dropped in the first flat group. Qp drops by four in "somewhat flat" groups and uses a constant value for "very flat" groups. Qp is mapped to quantization level before being used in prediction, quantization, reconstruction, and entropy encoding.

2.2.8 Substream and Slice Multiplexing

The syntax element of each component forms a *substream*. The three or four substreams are combined into a single bitstream using the substream multiplexing process. In simple 4:4:4 mode, the three substreams are for Y, Co, and Cg, respectively; the substream order in native 4:2:2 mode is the same as 4:4:4 mode for the first three substreams, and Y2 is in the fourth substream. The even-position luma (Y), odd-position luma (Y2), and chroma (Cb or Cr) in native 4:2:0 mode are the first, second, and third substreams, respectively. The fourth substream is only used in native 4:2:2 mode and is disabled for other modes.

The rate buffer converts a variable number of bits used to code each group to a constant-rate bitstream output. Figure 2.14 shows the substream multiplexer and rate buffer of the encoder. The syntax elements are passed to the balance FIFOs from the entropy encoder. The purpose of the balance FIFOs is to accumulate the syntax elements and ensure that the multiplexer has at least one mux word ($muxWordSize$ bits) whenever there is a mux word request, which is generated in the decoder model when the remaining bits are not enough for a largest syntax element ($maxSeSize$

bits). The worst case scenario is that the syntax element contains only one bit of data in every group; therefore, the balance FIFO needs to accumulate $(muxWordSize + maxSeSize - 1)$ syntax elements at the beginning of the slice. On the other hand, in the accumulation stage, every group can be coded with a syntax element of the maximum size; therefore, a size of $(muxWordSize + maxSeSize - 1)$ entries of $maxSeSize$-bit words ensures that the balance FIFO does not overflow.

In every group, the multiplexer receives up to three (or four in native 4:2:2 mode) mux word requests. If there is no request in one group, the multiplexer does not send any data to the rate buffer; if there is more than one mux word requests in one group time, the multiplexer sends mux words to the rate buffer in the following order: 1) one mux word for Y, 2) one mux word for Co, 3) one mux word for Cg, 4) one mux word for Y2. In constant bit rate (CBR) mode, the substream multiplexer adds "0" padding bits at the end of slice to make sure the total number of bits for a slice is equal to the budget. The decoder model keeps track of the fullness of the funnel shifters, i.e., the number of remaining bits; it does not contain the actual funnel shifter buffers. In every group time, the funnel shifter fullness is reduced by the size of the syntax element, and increased by the size of a mux word when a new mux word request is issued.

If an encoder supports more than one slice per line, slice multiplexing merges the bitstreams of different columns of slices into a single bitstream output. The multiplexed bitstream consists of *chunks*. The first chunk is from the first slice of the first row of slices, and the second chunk is from the second slice of the first row of slices, and so forth for the remaining slices in the first row of slices. This pattern repeats for all chunks of the first row of slices and then repeats for the remaining rows of slices in the picture.

2.3 DSC Decoding Algorithm

2.3.1 DSC Decoding Process

Figure 2.15 illustrates the DSC decoding process. The bitstream enters the rate buffer at a constant bit rate specified in the PPS, and is read out as mux words at a variable rate in each group time. *Substream demultiplexing* splits the bitstream into three or four substreams which are temporarily stored in the *funnel shifters*. *VLC entropy decoding* processes substreams in parallel. The coding mode (P-mode or ICH-mode) of the group is decoded, followed by the

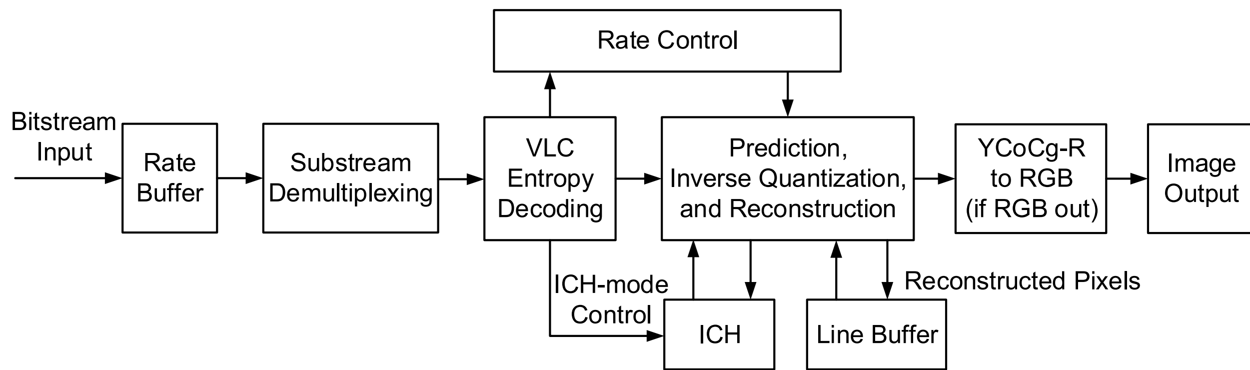


Figure 2.15: DSC decoding process [22].

quantized residuals for P-mode coded groups, and ICH indices for ICH-mode coded groups. The rate control uses two results from entropy decoding: 1) the actual number of bits used in coding the current group, and 2) the number of bits would be used if the sizes are accurately predicted. Optionally, flatness information is decoded. Rate control algorithm manages the fullness of an idealized rate buffer model and adjusts the quantization parameter (Qp) to maximize perceived quality. Qp is mapped to quantization level for luma and chroma components. For P-mode coded groups, sample values are predicted; quantized residuals are inverse quantized; decoded values are calculated by reconstruction, which uses the predicted values and inverse quantized residuals. For groups coded using ICH-mode, the ICH pixels pointed to by the selected indices are directly used as the decoded pixels. The decoded values are also stored in the line buffer and read out in the next line for MMAP and ICH. If RGB output pixel format is desired, decoded values are converted from YCoCg-R to RGB format.

2.3.2 Slice and Substream Demultiplexing

Slice demultiplexing splits the bitstream for different columns of slices when there are multiple slices per line; it is the inverse process of slice multiplexing.

Figure 2.16 shows the diagram of substream demultiplexing in the decoder. During the initial decoding delay period at the start of slices, the decoder accumulates bits in its rate buffer. Then the rate buffer sends up to three (or four in native 4:2:2 mode) mux words to the demultiplexer in every group time, depending on the number of mux word requests. When there are multiple requests, the order of the mux words is the same as that in the encoder. Every funnel shifter consists of a small buffer that can store at least a mux word and a syntax element with the size of one less

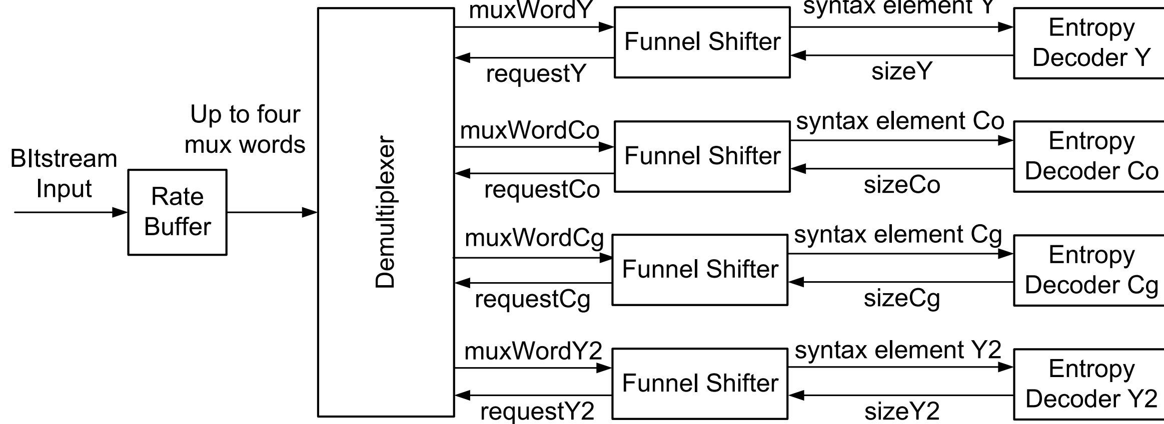


Figure 2.16: Substream demultiplexing in the DSC decoder, modified from [22, Figure 7-1].

bit than the maximum syntax element size. The funnel shifter removes the decoded bits and its fullness is decreased by the size of the decoded syntax element of the unit in each group time. If the resulting fullness is less than the maximum syntax element size, a mux word request is sent to the demultiplexer, so that the funnel shifter always has enough bits to construct a syntax element for the entropy decoder. Then the funnel shifter fullness is increased by the mux word size upon receiving the mux word. At the end of slice, the decoder shall flush the "0" padding bits if it runs in constant bit rate (CBR) mode.

2.3.3 VLC Entropy Decoding

In every group time, the entropy decoder parses syntax elements of the three or four units, which can be processed in parallel. The coding mode decision between P-mode and ICH-mode is decoded in the first luma unit in every group. If ICH-mode is used, three ICH indices are decoded and passed to ICH module of the decoder. If P-mode is selected, the entropy decoder first determines whether MPP is used for the unit by comparing the decoded residual size of the three samples in the group with the maximum residual size of the unit. The quantized residuals are decoded; their required sizes are calculated and used for size prediction in the next group time. The quantized residuals and MPP selection are passed to the prediction and reconstruction block. For each unit, the size of the syntax element is used for updating the funnel shifter and its fullness. The number of bits used in each component and the group and the total number of bits if the sizes have been optimally predicted are decoded and passed to rate control. In the third and fourth groups of a

supergroup, flatness information is decoded and passed to rate control.

2.3.4 Rate Control

The decoder uses the same rate control algorithm as the encoder. In every group time, the decoder RC buffer model adds the number of bits decoded in the group to its fullness, and subtracts the same number of bits as the encoder. The rate control algorithm produces the same quantization parameter (Qp) at every group. The decoder does not perform flatness checks since the flatness information (first flat group and flatness type) is decoded in the entropy decoder. The Qp adjustment process is exactly the same as the encoder.

2.3.5 Line Buffer

The decoder uses a line buffer to store a picture line of reconstructed pixel values of the previous line, which are used in MMAP, ICH, and BP search if BP is supported. To reduce implementation cost, the decoder line buffer can use a bit depth that is smaller than the size of the pixel components. The decoder is required to communicate the line buffer bit depth to the encoder if it is smaller than the component bit depth. The same bit depth reduction method is used in the decoder and encoder.

2.3.6 Prediction, Inverse Quantization, and Reconstruction

The decoder uses the same predictors as the encoder, i.e., modified median-adaptive prediction (MMAP), block prediction (BP), and midpoint prediction (MPP). BP is optional for the decoder. If the decoder does not support BP, the decoder does not need to perform BP search, since MMAP is always selected over BP. Otherwise, the decoder follows the same process as the encoder to select between BP and MMAP, since the bitstream does not explicitly specify the predictors used for each group. The decision between BP/MMAP and MPP is decoded from the syntax element by the entropy decoder.

In the decoder, the quantized residuals of all three pixels within the group are available at the beginning of the group. There is no data dependency in MMAP of different pixels within the group; therefore, the decoder can process three pixels in one clock cycle. Inverse quantization and reconstruction in the decoder are the same as in the encoder.

2.3.7 Indexed Color History

The structure and update process of the ICH in the decoder is identical to that of the encoder. The coding mode decision and the three ICH indices of each group are decoded by entropy decoding; therefore, the decoder does not need to search for the best ICH entries or make coding mode decisions. If a group is coded using ICH-mode, the pixel values of the ICH entries pointed to by the selected indices are directly used in the decoder's output pixel streams.

2.3.8 Color Space Conversion

If the decoder is configured to produce YCbCr outputs, no color space conversion is performed; otherwise, the decoded pixels are converted from YCoCg-R to RGB. The Co and Cg values are modified to be centered around zero. For 8, 10, 12, and 14 bpc:

$$cscCo = Co - (1 << bits_per_component) \tag{2.33}$$

$$cscCg = Cg - (1 << bits_per_component) \tag{2.34}$$

For 16 bpc, the conversion is different.

$$cscCo = (Co - 0\text{x}8000) << 1 \tag{2.35}$$

$$cscCg = (Cg - 0\text{x}8000) << 1 \tag{2.36}$$

Then, the color space conversion is defined as:

$$t = Y - (cscCg >> 1) \tag{2.37}$$

$$cscG = cscCg + t \tag{2.38}$$

$$cscB = t - (cscCo >> 1) \tag{2.39}$$

$$cscR = cscCo + cscB \tag{2.40}$$

The final values of R, G, and B components are calculated by bounding the values of $cscR$, $cscG$, and $cscB$ between 0 and the maximum value $(1 << bits_per_component - 1)$ using the $CLAMP$ function, which is defined in Equation 2.11. For example:

$$R = CLAMP(cscR,\ 0,\ (1 << bits_per_component - 1)) \tag{2.41}$$

2.4 Summary

This chapter highlights the key concepts and algorithms of the VESA Display Stream Compression (DSC) standard. The encoding and decoding processes and the operations of all sub-processes are summarized and analyzed.

Chapter 3

Hardware Architectures for Display Stream Compression Encoders

This chapter presents Display Stream Compression (DSC) encoder architectures, including: 1) a slice encoder architecture which utilizes the lowest amount of hardware and supports one slice per line at a throughput of one pixel per clock cycle, 2) a slice-interleaved encoder architecture that supports serially processing multiple slices per line with shared computation resources, and 3) a time-interleaved architecture that supports parallel and serial slice encoding while sharing computation resources.

3.1 Slice Encoder

The slice encoder implements the DSC encoding algorithm with support of one slice per line at a throughput of one pixel per cycle in simple 4:4:4 mode and two pixels per cycle in native 4:2:2/4:2:0 modes.

The input pixels in RGB format are converted to YCoCg-R within one clock cycle and stored in a register. The registered values are used for two purposes: 1) flatness check to determine whether the groups are flat; 2) temporarily stored in the pixel buffer, which is implemented as a shift register and can store 21 pixels. Pixels exit the pixel buffer after a delay of seven group times from the time they enter the pixel buffer; they are used in prediction, quantization, reconstruction, and indexed color history (ICH).

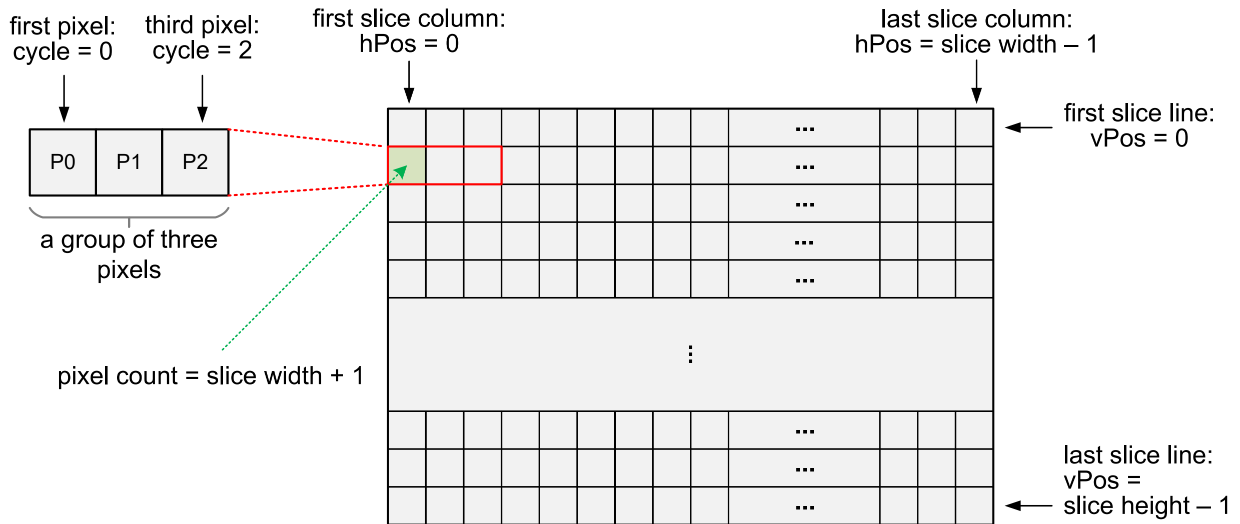


Figure 3.1: Illustration of the cycle counter, pixel counter, and position controller in a slice. Each square represents one pixel in the slice.

3.1.1 System Controllers

The slice encoder has eight controllers that are used during the encoding process.

1. A 2-bit cycle counter indicates the current cycle is the first, second, or third cycle within the 3-cycle group. The majority computation is used in one cycle out of the three cycles; therefore, the cycle controller is used to enable the registers that are updated.

2. A pixel counter that counts the number of pixel times that have passed since the start of encoding. This counter determines the initial transmission delay period for the rate buffer model in the rate control module.

3. A 2-bit group counter that indicates the order of the current group within a supergroup. It is used in flatness determination.

4. A horizontal ($hPos$) and vertical ($vPos$) position controller indicating the position of the current pixel time within the slice for prediction, ICH, and entropy encoder. If the current cycle has valid pixel input, $hPos$ increases by one, whereas $vPos$ increases by one if one slice line has been processed.

5. A group counter that indicates the number of groups that have passed and determines the starting group time for the read of balance FIFOs and size FIFOs.

6. A horizontal (*hPos_bf*) and vertical (*vPos_bf*) position controller indicating the position of the current pixel time within the slice for the read of balance FIFOs and size FIFOs.

7. A cycle counter that records the number of valid cycles that have passed since the balance FIFO read started. It determines the initial transmission delay period, after which the rate buffer starts removing bits.

8. A horizontal (*hPos_rb*) and vertical (*vPos_rb*) position controller indicating the position of the current pixel time within the slice for the read of the rate buffer.

Figure 3.1 shows the cycle counter, pixel counter, and position controller in one slice. The value of the horizontal position (*hPos*, *hPos_bf*, *hPos_rb*) counters starts from zero for pixels in the leftmost column of the slice, and increases by one per column. It is equal to (slice width – 1) for the last column of pixels. The value of the vertical position (*vPos*, *vPos_bf*, *vPos_rb*) counters starts from zero for the first row of pixels in the the slice, and increases by one per row. For the last row of pixels, their vertical position value is (slice height – 1). The cycle counter of the three pixels within one group are equal to zero, one, and two, respectively. The pixel counter starts with value of one; therefore, the first pixel in the second slice line has a pixel counter value of (slice width + 1).

3.1.2 Prediction, Quantization, and Reconstruction

The data dependency that the prediction of the current group pixels requires the reconstructed values of the previous group forms a feedback loop, which means a slice encoder may not process more than one group per cycle. In a straightforward implementation that processes one group per cycle, the critical path of the entire encoder is in the prediction loop, which includes: 1) filter and blend calculations, 2) serial MMAP for the three pixels, 3) predictor selection (first between BP and MMAP, then BP/MMAP and MPP), and coding mode decision between predictive and ICH mode.

In the proposed slice encoder, the prediction loop is pipelined such that it processes one group per three clock cycles, which yields an average throughput of one pixel per cycle. The pipelined prediction loop is shown in Figure 3.2. The task partitioning in the pipeline tries to balance the load among stages. Note that the prediction, quantization, and reconstruction of the three or four components are calculated in parallel; therefore, four instances of the corresponding

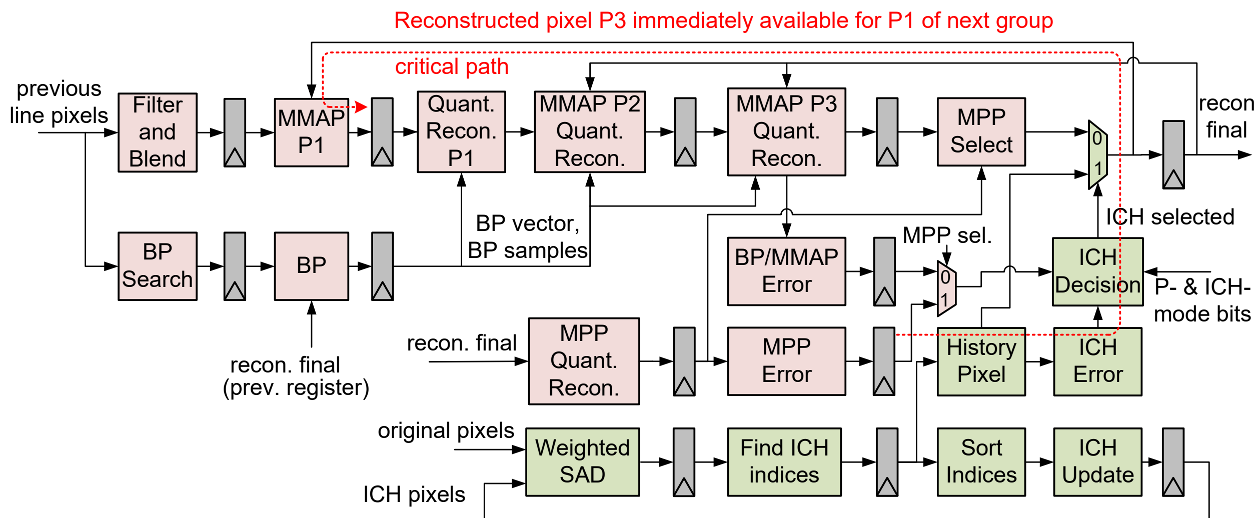


Figure 3.2: Pipelined block diagram of prediction, quantization, reconstruction (pink), and ICH (green) in the proposed slice encoder. Details of BP search are omitted.

blocks (except BP search) in Figure 3.2 are used in the encoder.

The filter and blend operations for MMAP and block prediction search use the reconstructed pixels from the previous line, which are naturally available before the current group. Therefore, they are computed in the previous group time and thus moved out of the critical path. Computation is pipelined into five stages, where the last two stages of the current group overlap with the first two stages of the next one for concurrent computation of two groups. The quantization, inverse quantization, and reconstruction of MMAP/BP and MPP are performed separately. The MMAP of the first pixel and BP of the entire group are done in the second stage. The third stage calculates the quantized residuals of the first pixel after the decision between MMAP and BP, which is inverse quantized before being used for the MMAP of the second pixel in the group. Other calculations performed in the third stage include quantization and reconstruction of the second pixel, MPP and its quantization and reconstruction for the entire group. The fourth stage calculates the MMAP, quantization and reconstruction of the third pixel in the group, as well as the size of the quantized residuals. The component-wise maximum errors for MMAP/BP and MPP are also calculated, which are used in the coding mode decision in the fifth stage. The encoder makes the decision between BP/MMAP and MPP in the last stage, and then decides between predictive mode and ICH mode for the current group.

Figure 3.3 shows the 4-stage pipeline of BP search in the proposed slice encoder. For all

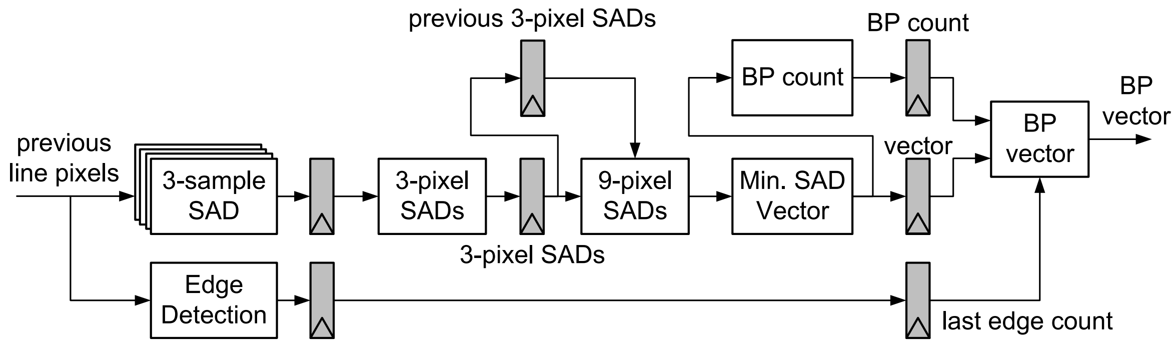


Figure 3.3: Four-stage pipeline of the BP search.

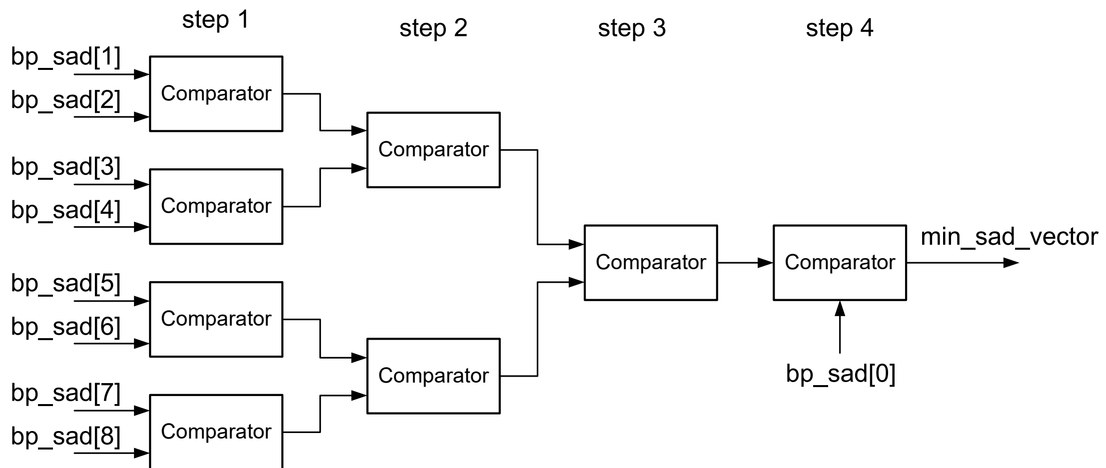


Figure 3.4: Four steps to find the BP vector with the smallest sum of absolute differences.

nine vectors, the component-wise 3-pixel sum of absolute differences (SAD) are calculated in the first stage and then added together to form the 3-pixel SAD containing all components. The clamped 3-pixel SADs are added with two 3-pixel SADs of the previous two groups to form the 9-pixel SADs in the third stage. The BP vector is calculated in the fourth stage and sent to BP. To reduce latency, the vector with minimum 9-pixel SAD is found using a four-step comparison in one cycle, as shown in Figure 3.4. In the first step, eight 9-pixel SADs ($bp_sad[1]$–$bp_sad[8]$, corresponding to vector -3, -4, ..., -10) are compared using four comparators, and the smaller SADs and their vectors are passed to the second step. The second step finds the smallest two SADs and the corresponding vectors. The minimum SAD and its vector of the eight 9-pixel SADs are found in step 3, which is compared to the 9-pixel SAD of vector -1 to find the final vector.

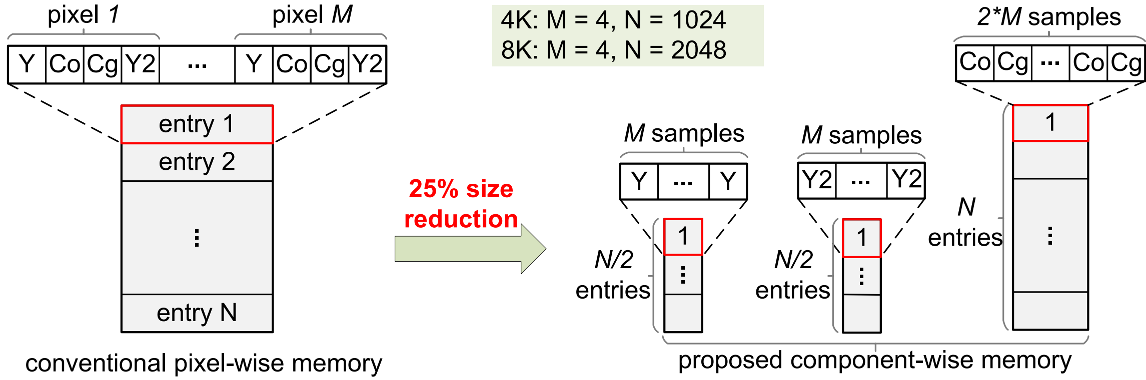


Figure 3.5: Component-wise memory architecture for the line buffer.

3.1.3 Indexed Color History

The indexed color history (ICH) module is pipelined into three stages, as shown in the bottom half of Figure 3.2. First, the weighted sum of absolute differences (SAD) between the original pixels and the 32 ICH pixels are calculated in parallel in the first cycle of the group. In the second cycle, the three ICH indices are found based on the smallest weighted SADs. The ICH pixels that are selected for the current group are addressed from the ICH registers in the third cycle. The errors between the selected ICH pixels and the original pixels are calculated, which are used together with the P-mode errors for the coding mode decision. The estimated number of bits required for P-mode and ICH-mode to code the current group is produced by the entropy encoder. In the third cycle of the group, the three selected indices are sorted and then used to update the ICH pixels and their valid signals.

3.1.4 Line Buffer

The three reconstructed pixels of each group are written to the line buffer in the first cycle of the next group time. The bit-width reduction is performed in parallel on all 9 or 12 samples of the group using the method in Section 2.2.5. The pixels in the last slice line (as well as the second last line for chroma samples of native 4:2:0 mode) are not written to the line buffer, since these pixels are not required for any following slice line. The same pixel values are read out in the third group before the group in the same position of the next line. Therefore, at the end of a group time, the previous line pixels of the next three groups are available. An exception is that the chroma samples in native 4:2:0 mode are delayed by one more slice line, since they used for line $i + 2$ rather

than $i + 1$ (where i is the current line number). The pixel values that are read out from the line buffer memory are stored in a shift register, which produces the values used in MMAP, BP search, and ICH.

The line buffer stores one picture line of pixels, which is 1920, 3840, and 7680 pixels for 1080p, 4K, and 8K resolutions, respectively. To make the memory depth equal to a power of 2, the storage is scaled up to 2048 pixels for 1080p, 4096 pixels for 4K, and 8192 pixels for 8K. Then, each line buffer entry stores four pixels instead of three. Two small buffers are used in the write and read side to handle the mismatch between number of pixels per group and per entry.

A straightforward line buffer implementation is to use a pixel-wise memory architecture, which requires storage of four components per pixel. However, the fourth component is only used in native 4:2:2 mode, which means the storage is wasted in other modes. In addition, the picture width is effectively halved in native 4:2:2 and 4:2:0 modes, which means smaller memory size is sufficient. To reduce the memory size, a component-wise memory architecture is used in the line buffer memory, as shown in Figure 3.5. In this architecture, half-depth memories are used for Y and Y2 components, and a full-depth memory is used for chroma (Co and Cg), resulting in a 25% memory size reduction. The Y2 memory stores the fourth component in 4:2:2 mode and is reused as an extended Y memory in other modes.

Native 4:2:0 mode contains only one chroma component and requires an additional slice line of delay, which means it needs to be stored separately. To adapt to the proposed component-wise architecture, the pixel components in native 4:2:0 mode are reordered before being written to the memory and converted back to the original order after being read out from memory. Specifically, the chroma component is stored in the Y and Y2 memory, and luma samples are stored in the Co and Cg memory.

3.1.5 Rate Control

The rate control module is pipelined into four stages, as shown in Figure 3.6. The offset and scale factors for the linear transformation are calculated using the horizontal and vertical slice position controllers, and the pixel count as of the previous group. Linear transformation is partitioned into two parts. In the first part, the offset factor is adjusted before being added to the original buffer fullness in the first cycle of a 3-cycle group time. The result is multiplied with

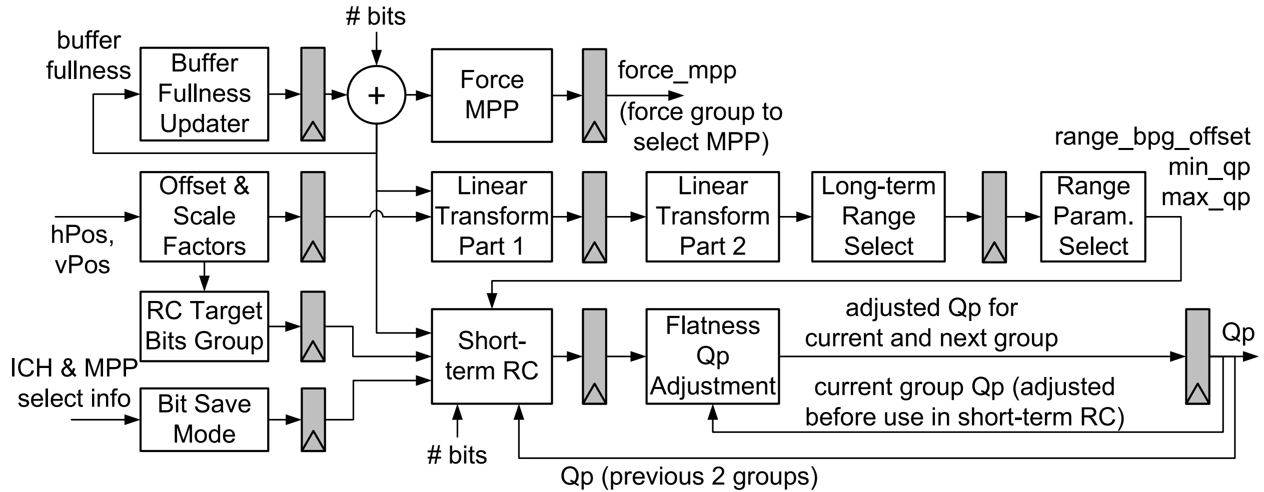


Figure 3.6: Pipelined block diagram of rate control in the slice encoder.

the transform scale factor to generate the final result of linear transformation in the second part. The transformed buffer fullness is compared to the 14 threshold values to determine the range for long-term rate control parameter selection. The range calculated from the information of current group is delayed by a group time and used to select range parameters for the short-term RC for the next group.

Short-term RC uses the buffer fullness, number of bits (from entropy encoder), ICH decision, and flatness information of the current group. However, the parameters are selected by the range from the previous group, which allows one additional group time to perform linear transformation and long-term range selection. The bit save mode is calculated in the third cycle of the current group, and the short-term rate control quantization parameter (Qp) adjustment is performed in the next cycle. The Qp from short-term RC is further adjusted based on the result of the flatness determination and is used as the final Qp for next group. The Qp of the current group is also adjusted by the flatness adjustment before being used in the short-term RC in the next group time.

The Qp is mapped to quantization levels for the luma and chroma components, which are used for prediction, quantization, reconstruction, and entropy encoding of the next group.

3.1.6 VLC Entropy Encoder

The three or four components of every group are encoded into syntax elements in parallel in the third cycle of the group. The entropy encoder estimates the number of bits that would be

used to encode the current group using P-mode and ICH-mode. This information is passed to coding mode decision. After the encoder has selected a coding mode, the entropy encoder codes each component of the group into a syntax element in the same cycle. The syntax elements and their sizes are stored and sent to the balance and size FIFOs in the next cycle, respectively.

3.1.7 Substream Multiplexing

The substream multiplexing controls the writes and reads of the balance FIFOs, size FIFOs, and the rate buffer. The writes for balance FIFOs and size FIFOs are performed in the first cycle of the next group time, compared to the entropy encoder. The reads of the balance FIFOs and size FIFOs are delayed by $(MuxWordSize + maxSeSize)$ number of group time compared to the writes, which makes sure the balance FIFOs always have enough bits whenever there is a mux word request. The write to the rate buffer is delayed by one group time from the read of balance FIFO. In the proposed slice encoder, the read of rate buffer does not start until the following three conditions are satisfied:

1. The ideal initial transmission delay period after the start of balance FIFO read has passed;
2. If the encoder runs in variable bit rate (VBR) mode, the size of the first chunk has been determined in the rate control algorithm, so that the encoder knows how many bits to transmit for the chunk;
3. The balance FIFO read is completed before the last chunk of the slice starts to be read from the rate buffer, so that the size of the last chunk can be calculated before transmitting bits of it. This constraint is satisfied as long as the rate buffer read waits until one slice line of data has been read from the balance FIFO.

Synchronous FIFO

Both balance FIFO and size FIFO are designed as synchronous FIFOs, which use the same clock for write and read. Figure 3.7 illustrates the circuit of the synchronous FIFO in the slice encoder. A dual-port memory is used as the storage element, which contains a write port and a read port that supports one read and one write in one cycle. The memory is indexed by the write address (*wr_address*) and read address (*rd_address*), the bit-widths of which are determined by the

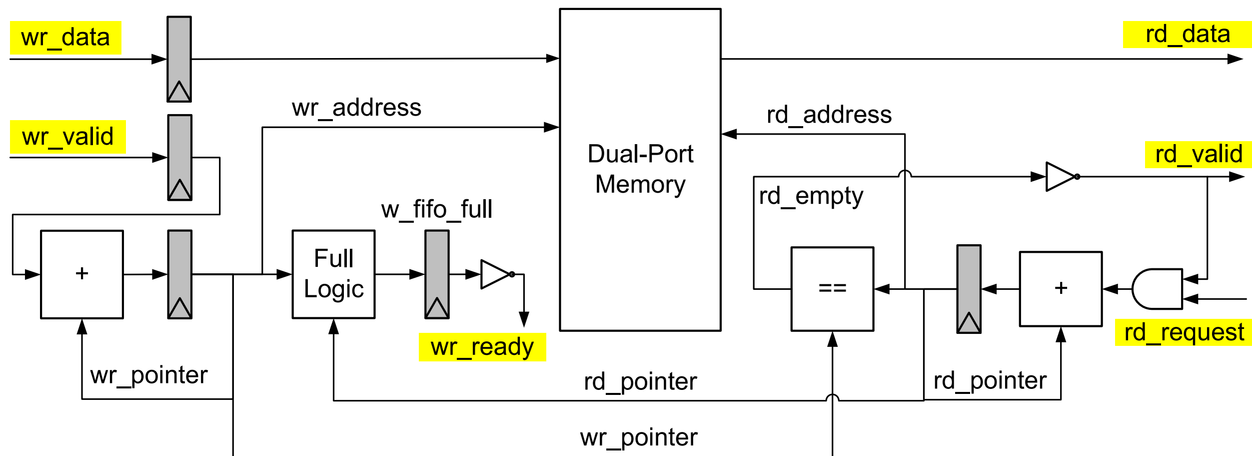


Figure 3.7: Synchronous FIFO circuit.

number of entries in the memory (*fifo_depth*):

$$N = \lceil log_2(fifo_depth) \rceil \tag{3.1}$$

A write pointer and a read pointer which have one more bit than the addresses determine whether the FIFO is full or empty. The write (read) address is equal to the write (read) pointer without its MSB.

The write pointer always points to the next memory location to be written to. When valid data enter the FIFO, they are written to the memory location indexed by the write address. The write pointer is increased by one and points to the next location. If the write pointer already reaches the last location of the FIFO before the increment, it is moved to point to the first location of the FIFO memory. If the write pointer and read pointer only differ in their MSB, the FIFO is full.

The read pointer always points to the next memory location to be read from. When the FIFO receives a read request and it is not empty, the valid flag (*rd_valid*) of the read data is asserted indicating the data are valid. The read pointer is increased by one and points to the next memory location, which is the first memory location if it points to the last memory location in the current cycle. The FIFO is empty if the read pointer is equal to the write pointer.

When returning from the last FIFO address to the first address, the LSBs of the pointer containing the FIFO address must be reset to zero, and the MSB should be flipped. If the FIFO depth is a power of two, a simple increment is sufficient to perform both operations.

The ready signal (*wr_ready*) is set to zero if the FIFO is full, which means the data source should stop sending new data. In case it takes multiple cycles for the data source to stop sending

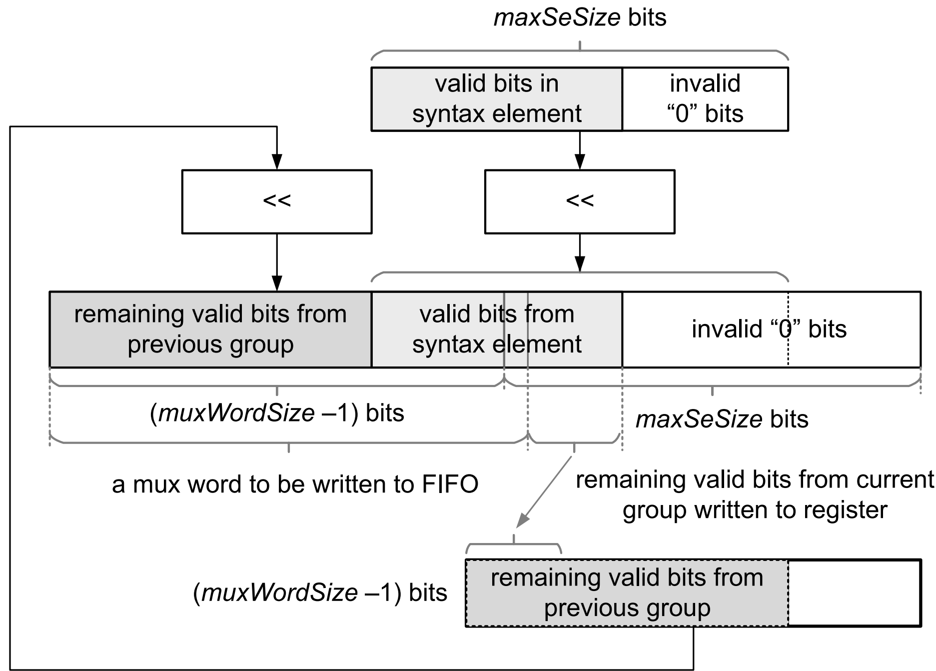


Figure 3.8: Mux word construction from syntax element for balance FIFO.

new data after the FIFO is full, the FIFO should set the ready signal to zero if the number of remaining entries in the FIFO is equal to or smaller than the number of cycles for the ready signal to propagate to the data source.

Balance FIFO

The slice encoder contains four parallel balance FIFOs, one for each substream. In every group time, a syntax element is generated by the entropy encoder for each component. The syntax elements are packed into mux words before being written to the balance FIFO. Figure 3.8 illustrates the steps of constructing mux words from the syntax elements. A register containing one less bit than a mux word stores the remaining bits after each group. When some bits of the register are not valid (i.e., the number of remaining bits is less than the register bit-width), the invalid bits are filled as zeros in the lower bits of the register. In the next group, the bits in the register are shifted left by $maxSeSize$ bits; the new syntax element is shifted left such that its MSB is next to the lowest valid bit of the register. If the total number of valid bits is enough for a mux word, the top $muxWordSize$ bits form a mux word and are written to the FIFO, while the remaining bits are written to the register. In the last group of a slice, the top mux word size number of bits are

Table 3.1: Size FIFO Word Width and Depth

Bits per Component	8	10	12	14	16
Word Width (bits)	6	6	6	6	7
FIFO Depth (Y, Co, Cg)	84	92	116	124	128
FIFO Depth (Y2)	80	88	112	120	128

written to the FIFO even if there are not enough bits for a mux word.

In the read side, a mux word is read out and sent to the substream multiplexer if there is a mux word request. When the horizontal and vertical position controller of the balance FIFO read reaches the end of the slice, all syntax elements in the slice have been read out. If more mux words are requested, "0" bits are used for any extra bits.

Size FIFO

Table 3.1 summarizes the minimum word width and memory depth of the size FIFO for different component bit depths. Every entry stores the size of a syntax element, the minimum word width is calculated as:

$$width = \lceil log_2(maxSeSize) \rceil \tag{3.2}$$

The minimum FIFO depth is equal to:

$$depth = muxWordSize + maxSeSize - 1 \tag{3.3}$$

Decoder Model

The decoder model updates the funnel shifter fullness of each substream at the end of every group time. The fullness is reset to zero at the beginning of encoding. At the beginning of a group, if the fullness is less than the maximum size of a syntax element, a mux word request is generated and sent out. Then the fullness is increased by the size of a mux word. Then, one data is read from the size FIFO, which is the amount that the fullness should decrease. The final shifter fullness of the group is stored in a register and used for next group.

Rate Buffer

The mux words are written to and read from the memory in the first and second cycle of the 3-cycle group time, respectively. In any cycle, there is maximum one memory access, which means the rate buffer can be implemented as a single-bank, single-port memory. Since the substream multiplexer sends up to four mux words in native 4:2:2 mode in one group time, each rate buffer memory entry stores four mux words, which is 192 bits in 8 and 10 bpc, and 256 bits in 12, 14, and 16 bpc. The rate buffer is guaranteed to never overflow in all modes by using sufficient storage capacity, and never underflow in constant bit rate (CBR) mode by adding sufficient initial delay before read starts. When rate buffer underflow occurs in variable bit rate (VBR) mode, no bits are sent to the output.

In the buffer write side, mux words are accumulated and once there are four mux words present, a rate buffer write is enabled. The remaining mux words are stored in a register and used for the next group time. Once the rate buffer read starts, the specified number of bits per pixel are sent to encoder output. A rate buffer read request is generated if the number of remaining bits is not enough for bitstream output in the next group time. The newly read mux words are concatenated with the remaining bits from the previous group time using a similar approach as the mux word construction in the balance FIFO.

In VBR mode, the size of chunks vary and are determined in the buffer level tracker of the rate control algorithm. The exception is the size of the last chunk of each slice, which is calculated by subtracting the total bytes of the mux words from the total size of all previous chunks in the slice. The last chunk is an empty chunk if the result of the subtraction is less than or equal to zero.

Since the rate buffer read has a delay compared to the write side, after all mux words of the current slice have been written to the rate buffer, the write for the next slice starts; however, the encoder needs more time to finish sending all the bits of the current slice to the encoder output. Therefore, the number of rate buffer writes and reads for the current slice are recorded, and the rate buffer read is disabled if all the mux words for the current slice have been read out, but the encoder still needs to send more bitstream (padding “0” bits) to make sure the total amount of bits is equal to the budget in CBR mode.

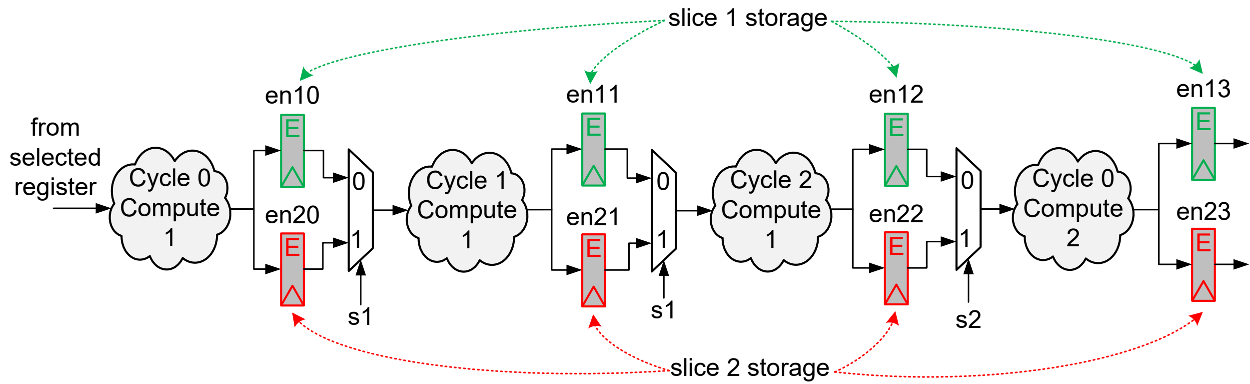


Figure 3.9: Datapath of a slice-interleaved encoder architecture that supports two slices per line. Two sets of registers that are enable-able (noted by "E" in the registers) are used as the storage for the two slices.

3.2 Slice-Interleaved Encoder

The *slice-interleaved* encoder architecture [34] supports encoding multiple slices per line with minimal additional hardware compared to the slice encoder. Figure 2.4 shows examples of two and four slices per line. The slice-interleaved encoder processes one slice at a time, which results in throughput of one pixel per cycle—the same as the slice encoder architecture.

3.2.1 Architecture

The slice-interleaved encoder is built based on the slice encoder. It supports multiple slices per line by sharing the combinational logic of the computation by all the slices in the same row, and replicating only registers for each slice. Figure 3.9 shows four computation stages in a slice-interleaved encoder that supports two slices per line.

A multiplexer selects the source register of the current slice being processed; the result is written to the register of the same slice. The registers are enable-able and only the ones belonging to the current slice are active. Every slice in the row is selected for one slice line number of cycles. The reason that each slice needs their own registers is that the encoder switches to the next slice after processing one line of the current slice, which will overwrite the encoding state of the current slice and cause encoding error if both slice use the same set of registers.

Every column of slices requires a dedicated set of balance FIFOs and size FIFOs for their syntax elements and sizes. Therefore, to support up to four slices per line, 16 balance FIFOs and 16

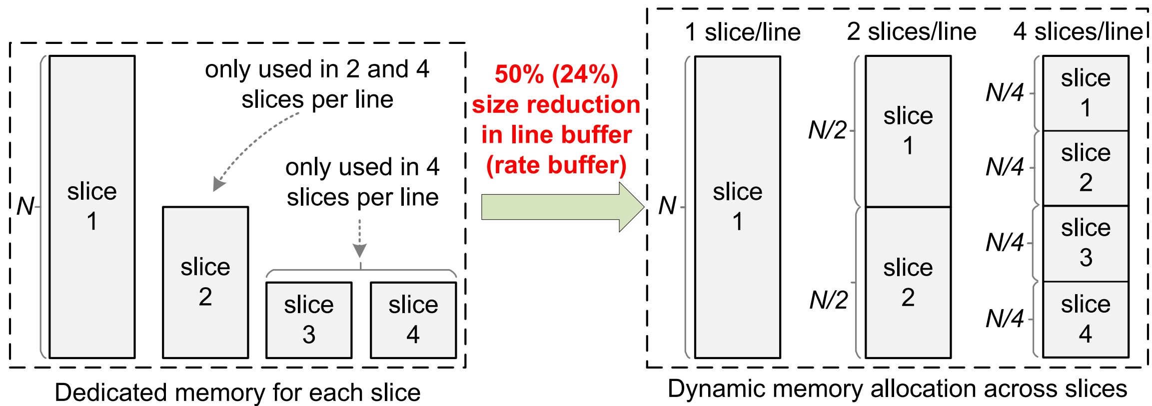


Figure 3.10: Dynamic line buffer and rate buffer memory allocation across slices in the slice-interleaved encoder architecture.

size FIFOs are required. The same pixel buffer is used for all columns of slices. Since the ICH is invalidated after the end of a slice line when the picture is configured into more than one slice per line, one ICH memory is used for all columns of slices.

Slice multiplexing is performed in the rate buffer read side. One chunk of bitstream of each slice from the leftmost slice to the rightmost slice is sent to output. In the rate buffer read side, two counters count the number of pixel times of the chunk and the number of chunks processed in the slice. The pixel time counter is reset to zero at the start of a chunk, and increased by one if valid bitstream is sent to output in the current cycle. The end of a chunk is indicated by a counter value equal to the slice width, after which a chunk from the next slice is started. When a chunk of the rightmost slice is completed, the chunk counter is increased by one to indicate that the same number of chunks has been processed in all slices in the same row. When the chunk counter's value reaches slice height, all chunks of the current row of slices are done. This process repeats for the slices in the future rows until the encoding is done.

3.2.2 Dynamic Memory Allocation Across Slices

Since the slice-interleaved encoder processes one slice at any given point of time, the line buffer and rate buffer have up to one read or write access per cycle; therefore, same as the slice encoder, they can be implemented as single-bank and single-port memory. To reduce the memory size while supporting different numbers of slices per line, a dynamic memory allocation scheme

is applied to the line buffer and rate buffer memory, as shown in Figure 3.10. When the encoder processes one slice per line, the entire memory is used for the only slice in the row. The memory is divided into two and four portions when processing two and four slices per line, respectively, in which every slice is allocated a portion of the memory. As every slice requires one slice line's pixel storage in the line buffer, the total size of the line buffer is always equal to the size of one picture line by using the dynamic allocation scheme, regardless of the number of slices per line. However, the size of the rate buffer increases due to additional buffering to support more than one slice per line (see Section 3.2.3). Nevertheless, the dynamic allocation scheme reduces the line buffer and rate buffer size by 50% and 24%, respectively, to support one, two, and four slices per line compared to using a dedicated memory for each column of slices.

3.2.3 Slice Encode Timing and Extra Buffering

Figure 3.11(a) shows the slice timing for idealized and slice-interleaved encoding of two slices per line. Pixels enter the encoder at a rate of one pixel per cycle in raster-scan order, which means one line of slices 1 and 2 enter the encoder alternately: one line of slice 1 followed by the same line of slice 2. In the idealized encode timing, each slice is processed only during the time when the pixels of the slice enter the encoder, as if the other slice does not exist. The encoded bitstream is transmitted to output after the initial delay period (assumed to be less than one slice line pixel time in this example), which includes the encoder pipeline delay, balance FIFO delay, and initial transmission delay of the rate buffer. In the idealized encode timing, each chunk is split into two parts that are not continuous. The slice-interleaved encode timing adds extra delay on the rate buffer read side such that each chunk is read out continuously and the bitstream output follows the order that one chunk of slice 1 and then one chunk of slice 2. The amount of extra delay is determined by the constraint that bitstream is not read out before they would be in the idealized timing. This constraint is satisfied by making the second part of chunk 1 in slice 1 start no earlier than it would be in the idealized timing. Therefore, the start time of the first part of this chunk is not earlier than the start time of the first part of chunk 1 in slice 2. Overall, the first part of chunk 1 in slice 1 is delayed by one chunk time.

Figure 3.11(b) shows the slice timing for idealized and slice-interleaved encoding of four slices per line, assuming that the initial delay is larger than one slice line number of cycles. In

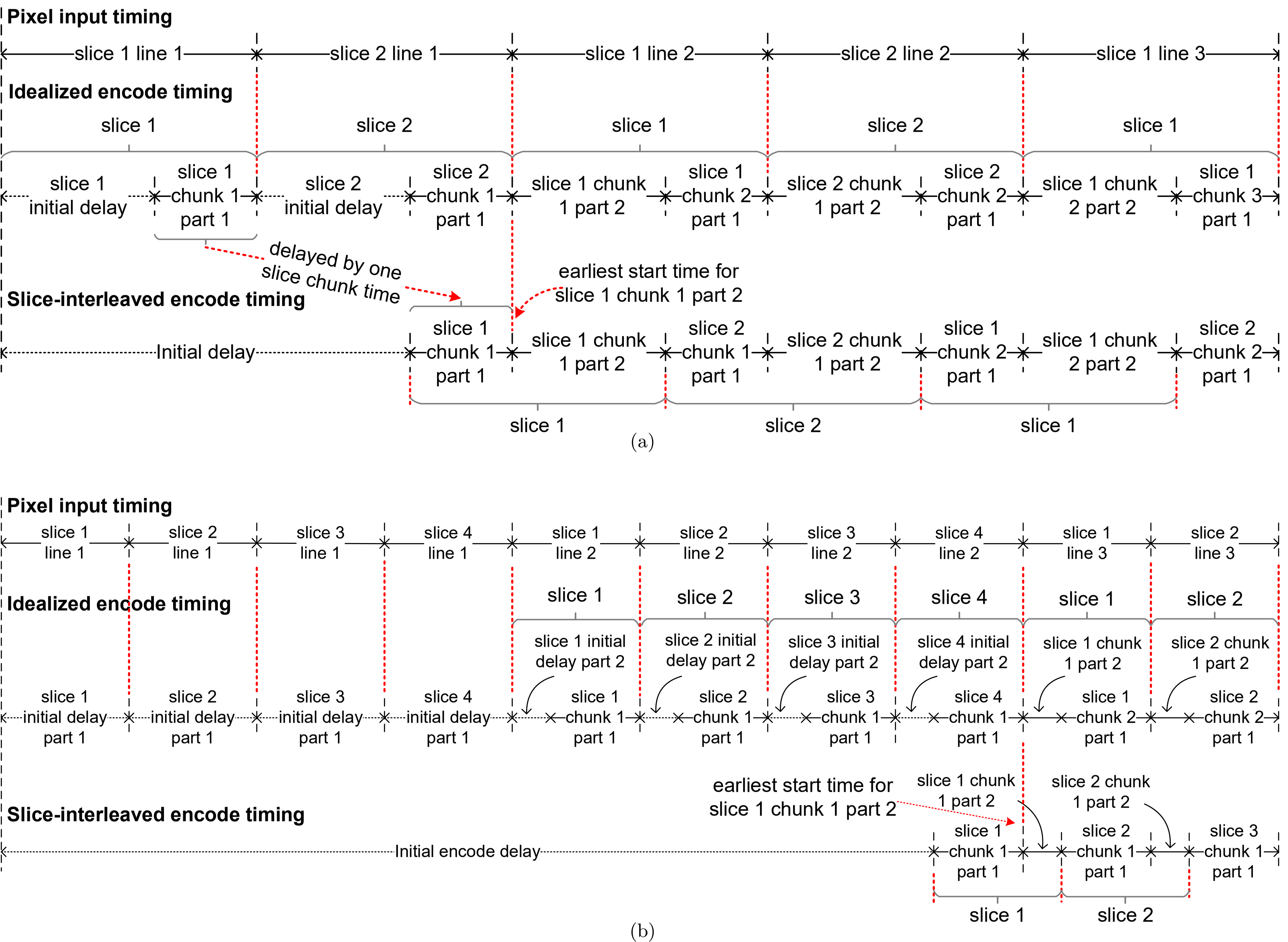


Figure 3.11: Slice-interleaved encode timing. (a) Two slices per line. (b) Four slices per line.

this example, the initial delay is split into two parts, where the first part is one slice line number of cycles. Additional delay is applied to the rate buffer read to make sure that the chunks are continuous. The same timing constraint as two slices per line should be satisfied, i.e., the bitstream is not read out earlier than it would be in the idealized timing. This constraint is satisfied if the first chunk of slice 1 is not read out from the rate buffer until the starting point of the first chunk of slice 4 in the idealized timing.

The additional delay in the rate buffer read brings the requirements for extra buffering. In the case of two slices per line, the first part of chunk 1 in slices 1 and 2 are required for extra buffering. For four slices per line, the first part of chunk 1 in all four slices are required for extra buffering. In both cases, a conservative upper bound for the amount of extra buffering is one picture line's worth of bitstream.

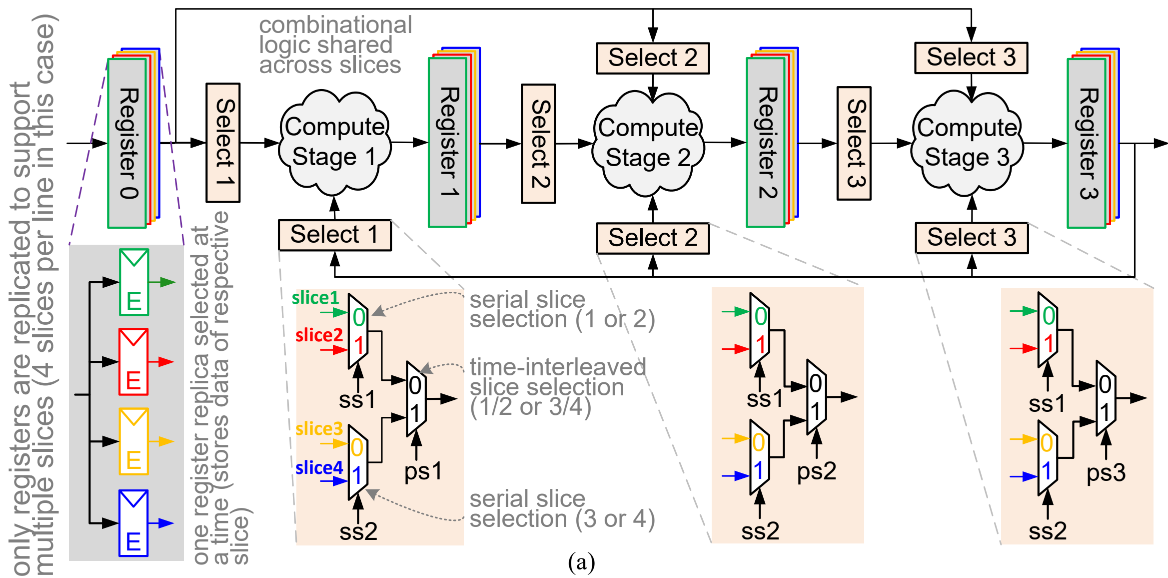
3.3 Time-Interleaved Encoder

3.3.1 Time-Interleaved Encoding Architecture

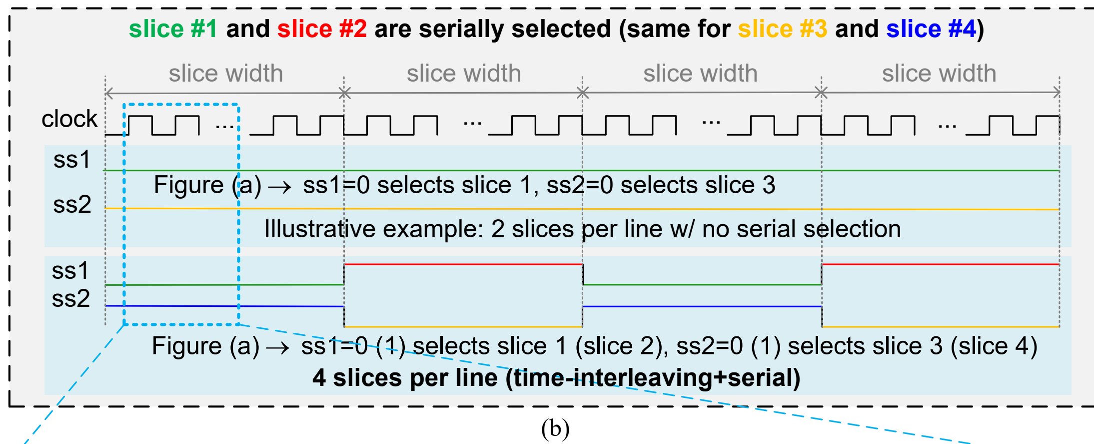
The time-interleaved encoder architecture is built on the slice encoder architecture and supports multiple slices per line using two techniques that share computation resources across slices: 1) group-level parallel processing of two or three slices, 2) serial slice processing to support more slices per line.

In the slice encoder, the computation of every pipeline stage is used only one cycle per 3-cycle group, resulting in two idle cycles. The time-interleaved encoder makes use of the idle cycles to process other slices of the same row; therefore, by using one or two idle cycles, two or three slices can be processed in the 3-cycle group, respectively. As a result, the time-interleaved encoder achieves double or triple throughput per cycle compared to the slice encoder. The combinational logic is shared by the two or three parallelly encoded slices, whereas the registers are replicated for each slice and multiplexed to support multiple slices per line. To avoid conflict, the slices that are processed in parallel at the group-level are always in different cycles of the 3-cycle group.

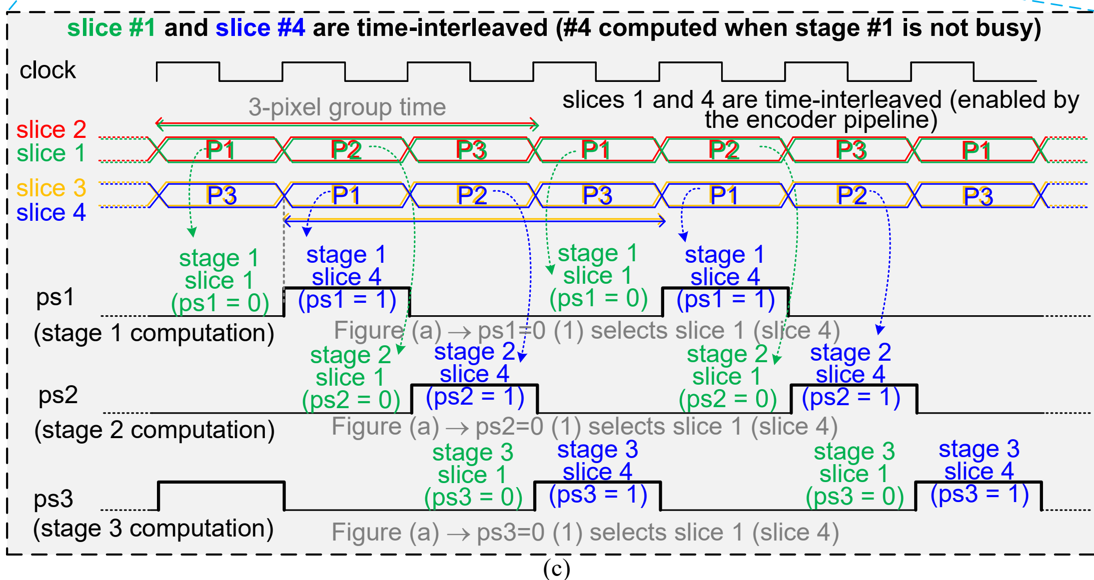
When the number of slices per line is not larger than the maximum supported parallelly encoded slices, all slices of the same row are encoded in parallel; otherwise, the time-interleaved encoder divides the slices into multiple portions, and performs a serial slice selection before the



(a)



(b)



(c)

Figure 3.12: Time-interleaved encoding architecture. (a) A generic 3-stage group data-path. (b) Serial slice selection timing (ss1, ss2). (c) Parallel-processing slice selection timing (ps1, ps2, ps3).

parallel slice processing. For a time-interleaved encoder that supports processing two slices in parallel and the picture is configured into $2 * N$ slices per line, one slice of the first N slices and another slice of the second N slices are serially selected. The two selected slices are processed in parallel. The serial slice selection uses the same mechanism as the slice-interleaved encoder.

Figure 3.12 shows an example of the time-interleaved encoding architecture that processes two slices in parallel and supports one, two, and four slices per line. Every register in the computation pipeline is replicated into four instances, one for each column of slice. Two levels of register selection are included: 1) serial slice selection by the serial selectors ($ss1, ss2$); 2) parallel slice selection by the parallel selectors ($ps1, ps2, ps3$). When the encoder processes pictures configured into one slice per line, the first register in every stage is always selected (all selectors are equal to zero), and the other three registers are not used. In this case, the time-interleaved encoder behaves as a slice encoder. To encode in two slices per line, the two serial selectors are always equal to zero (see the top half of Figure 3.12(b)), and the registers for slices 1 and 3 in Figure 3.12 store the coding states for the two slices in the same row for parallel processing. Registers for slices 2 and 4 are unused. When the encoder works in four slices per line, the encoding state of the four slices are stored in the four registers of each pipeline stage. At the slice level, slices 1 and 2 (also slices 3 and 4) are sequentially selected by selection signals $ss1$ and $ss2$, alternately computing odd and even slices, as shown in the bottom half of Figure 3.12(b). At the 3-pixel group level, the two serially selected slices are then time-interleaved, with the parallel slice selectors ($ps1, ps2, ps3$) sequentially selecting the proper slice registers, as shown in Figure 3.12(c). This time-interleaved encoder achieves two pixels per cycle throughput in 4:4:4 mode and four pixels per cycle in native 4:2:2 and 4:2:0 modes when encoding two or four slices per line.

3.3.2 Memory Architecture

The support of processing N ($N = 2,\ 3$) slices in parallel in the time-interleaved encoder architecture brings additional requirements on the memories of the encoder, including: 1) one instance each of ICH and pixel buffer are required for each parallelly processed slice; 2) N memory banks are required in the line buffer and rate buffer to enable N writes and/or reads per cycle; 3) an N-bank input buffer is required to split the input pixels for each column of slice.

The line buffer and rate buffer are built using N-bank single-port memory, where parallelly

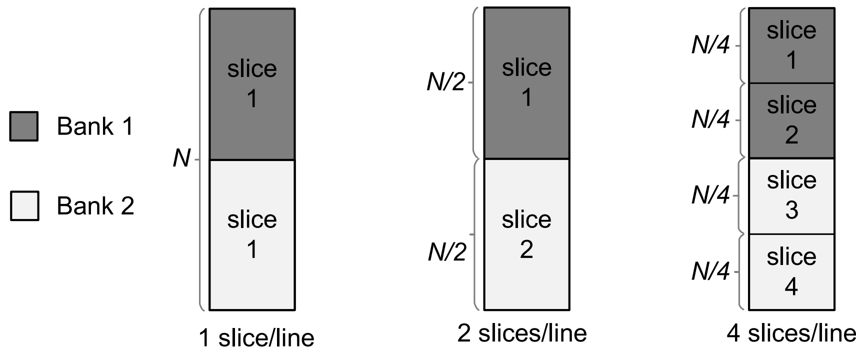


Figure 3.13: Dynamic memory allocation across slices for dual-bank line buffer and rate buffer in the time-interleaved encoder architecture. In this example, memory contains N entries that are divided into two memory banks.

processed slices are allocated with different memory banks, which can be written and read independently. The dynamic memory allocation scheme is applied to both line buffer and rate buffer. Figure 3.13 shows an example of the dynamic memory allocation with dual-bank memories when processing one, two, and four slices per line. In case of one slice per line, all memory banks are used for the only slice in the row, whereas one bank and half a bank is allocated for each slice in two or four slices per line, respectively. The total size requirement for the line buffer is one picture line of pixels, regardless of the maximum supported number of slices per line, whereas the rate buffer size increases as the number of slices per line.

An input buffer temporarily stores the raster-scan-ordered input pixels in different portions of the memory to enable multiple reads by multiple slices. To support encoding N slices in parallel, the input buffer requires one write and N reads per cycle. Therefore, the input buffer is built as a N-bank dual-port memory. Since every parallelly processed slice is processed at a throughput of one pixel per cycle, N pixels are read from the input buffer and processed per cycle in total. To match the read rate, N pixels are written to an entry in one of the banks in every cycle (i.e., every entry stores N pixels). The input buffer also utilizes the component-wise memory architecture to reduce the required buffer size. Unlike the line buffer, no component switching is needed in native 4:2:0 mode, since all components behave the same in the input buffer.

The minimum size for the input buffer is half a picture line of pixels to support processing two slices in parallel. This can be explained from the input buffer write and read timing in Figure 3.14. The maximum memory occupancy for bank 1 of the input buffer occurs when line 1 of

slice 1 finishes entering the input buffer, which contains half a slice line of pixels of slice 1. Similarly, the maximum occupancy for bank 2 occurs when line 1 of slice 2 finishes entering the input buffer and half a slice line of pixels of slice 2 remains in the buffer. Therefore, both banks should be able to store at least half a slice line of pixels, which means a total of half a picture line of storage is required. The input buffer size to support processing three slices in parallel is two thirds of a picture line of pixels, which can be derived using similar analysis on the input buffer read and write timing. The number of serially selected slices has no impact on the input buffer size requirement.

Regardless of the number of slices processed in parallel, four balance FIFOs and four size FIFOs are needed for every column of slices, which is the same as the slice-interleaved encoder architecture.

3.3.3 Slice Encode Timing and Extra Buffering

Figure 3.14 shows the slice timing of the time-interleaved architecture encoding two slices per line in parallel. The input pixels are written to the input buffer at two pixels per cycle, in which the pixels of slices 1 and 2 are written to memory banks 1 and 2, respectively. Pixels are read from both banks at a rate of one pixel per bank per cycle. The pixels of slice 2 are read from input buffer half slice line number of cycles later than the pixels of slice 1. Both slices are encoded in parallel at a throughput of one pixel per cycle per slice. The encoded bitstream of slices 1 and 2 are written to banks 1 and 2 of the rate buffer after the initial delay period, respectively. The initial delay period consists of the pipeline delay in the encoder, the balance FIFO delay, as well as the initial transmission delay of the rate buffer. Bitstream is read out from the rate buffer after the initial encode delay at a rate of two pixels' worth of bits per cycle. The encoder reads one chunk of slice 1 from bank 1, then reads one chunk of slice 2 from bank 2; this process repeats until all chunks are read. The earliest starting time that bitstream can be read from the rate buffer is the time when the first chunk of slice 2 is written to the rate buffer—this constraint makes sure that bitstream can be sent to output continuously.

Figure 3.15 shows the slice timing of the time-interleaved architecture in Figure 3.12 processing four slices per line. The pixel inputs are written to the input buffer at a rate of two pixels per cycle, where the pixels of the first and second (third and fourth) slices are written to the first and second half of the first (second) memory bank. In every cycle, one pixel is read from

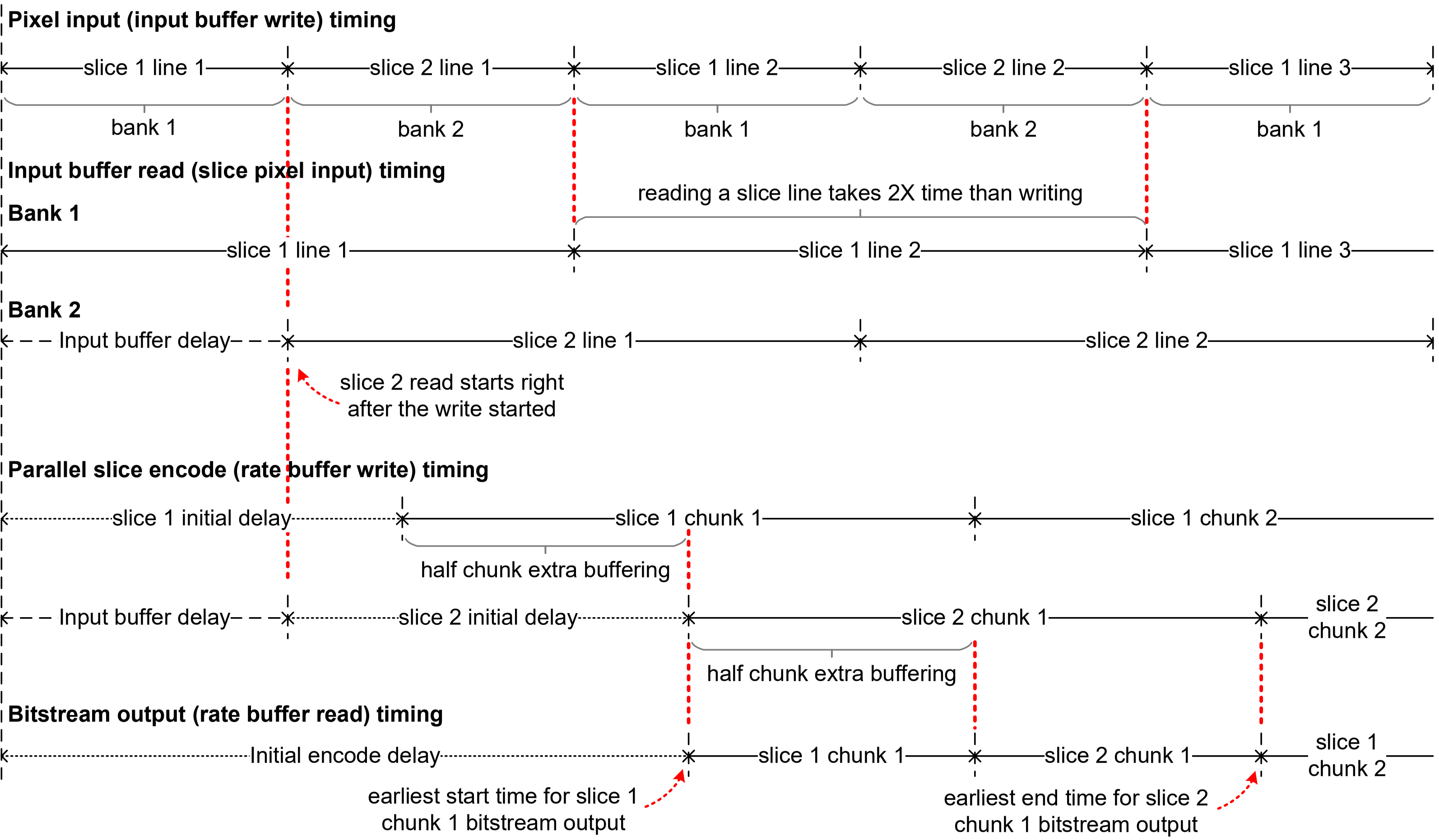


Figure 3.14: Time-interleaved slice encode timing for two slices per line.

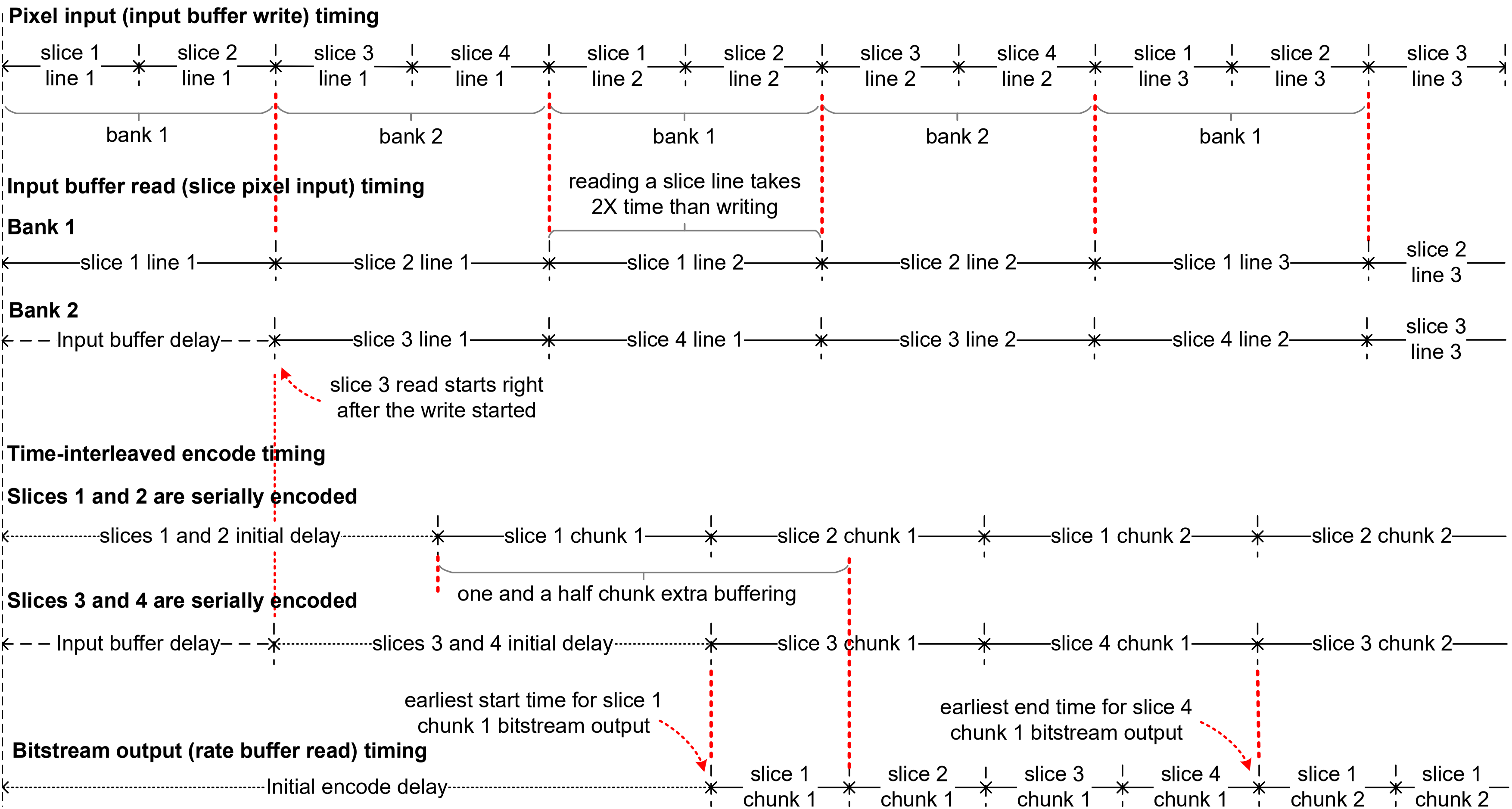


Figure 3.15: Time-interleaved slice encode timing for four slices per line.

each bank of the input buffer and sent to the encoder. The pixels of slice 3 are read after the input buffer delay, which is one slice line number of cycles. Slices 1 and 2 (and slices 3 and 4) are serially encoded which has the same slice timing as the slice-interleaved encode timing of two slices per line. After the initial encode delay period, the bitstream is read from the rate buffer and sent to output at a rate of two pixels' worth of bits per cycle. To ensure there is always enough bitstream in the rate buffer, the first chunk of slice 1 is read no earlier than the starting time of chunk 1 of slice 3.

In both two and four slices per line, additional rate buffer storage compared to the idealized size of the slice encoder is required to make sure the rate buffer does not overflow. The amount of additional buffering can be derived from the extra delay from rate buffer write to read. In two slices per line encoding, half chunk of slice 1 and half chunk of slice 2 are required for additional buffering; therefore, the total amount of extra buffering is half a picture line's worth of bitstream. In case of four slices per line, the amount of additional buffering on top of the slice-interleaved rate buffer size is: 1) an entire chunk of the first slice and half of a chunk of the second slice are required for extra buffering of the first bank in rate buffer; 2) an entire chunk of the third slice and half of a chunk of the fourth slice are required for extra buffering of the second bank in rate buffer. The total amount of extra buffering is three chunks, i.e., three quarters of a picture line's worth of bitstream.

3.4 Summary

This chapter presents three Display Stream Compression (DSC) encoder architectures. The design of a slice decoder is discussed. The slice-interleaved encoding architecture is built on top of the slice encoder architecture using shared computation logic and replicates only registers to support serial processing of multiple slices per line. The time-interleaved encoding architecture further explores group-level parallel encoding by sharing the computation logic while achieving higher throughput.

Chapter 4

A Display Stream Compression Encoder Chip in 28 nm for 4K Video Resolution

A Display Stream Compression (DSC) version 1.2a encoder chip [35] which is designed and fabricated in TSMC 28 nm LP9M CMOS technology is presented. To the best of the author's knowledge, this chip is the first published DSC encoder chip. The encoder utilizes time-interleaved encoding architecture which is capable of processing two slices in parallel and achieves a throughput up to two pixels per cycle.

4.1 Design Goals and Key Features

The design goal of the DSC encoder chip is to achieve area-efficient, high-performance, and energy-efficient real-time encoding of 4K videos. It is used to evaluate the time-interleaved encoding architecture presented in Chapter 3.

The chip fully complies with DSC version 1.2a and is backward compatible with DSC versions 1.2 and 1.1. It supports the following features.

- Video resolution up to 4K UHD (3840×2160)
- Uncompressed video color bit depth of 8 and 10 bits per component (bpc)
- 4:4:4 pixels in RGB and YCbCr format

- YCbCr 4:2:2 pixels in simple 4:2:2 and native 4:2:2 modes
- YCbCr 4:2:0 pixels in native 4:2:0 mode
- Compressed data rate of 8–16 bits per pixel (bpp) in both constant bit rate (CBR) and variable bit rate (VBR) modes
- One slice per line at a throughput of one pixel per cycle, and two and four slices per line at a throughput of two pixels per cycle

4.2 Chip-Level Architecture

Figure 4.1 shows the chip-level architecture with details of the logic outside the encoder core. The chip contains two clock domains: the core clock domain and the I/O clock domain. The I/O clock comes from off-chip input, whereas the core clock is generated using an on-chip clock generator, which is configurable to provide different frequencies. Both the I/O and core clock are de-skewed before being used. A scan module is used for loading the configurations into the chip, and scanning out the cyclic redundancy check (CRC) values and some internal signals for debugging. Pixel input and bitstream output use double data rate (DDR) logic to reduce the data width by 50%. Two dual-clock FIFOs pass the input pixels and output bitstream across I/O and core clock domains. One-bit signals cross the clock domains by synchronizers in the destination domain, which are implemented using three flip-flops that are clocked by the destination clock. An on-chip test generator produces pseudo-random pixel values in the core clock domain which can be directly fed to the encoder core for full-speed testing. For testing purposes, the chip is configurable for selecting different sources for the core pixel input and output bitstream.

4.2.1 Double Data Rate I/O

Figure 4.2(a) shows the DDR input logic for the input pixels. In the first stage, a negative-edge-triggered register (upper-left in the figure) captures the input data when clock is high, and a positive-edge-triggered register (bottom-left in the figure) captures the input data when clock is low. The second stage registers synchronize both half cycle data to the rising edge of the clock.

Figure 4.2(b) shows the DDR output logic for the output bitstream. The input data are divided into two halves and are synchronized with the rising edge of the clock. Then, the clock is used as the select signal of the multiplexer, which selects the upper half data to the output when

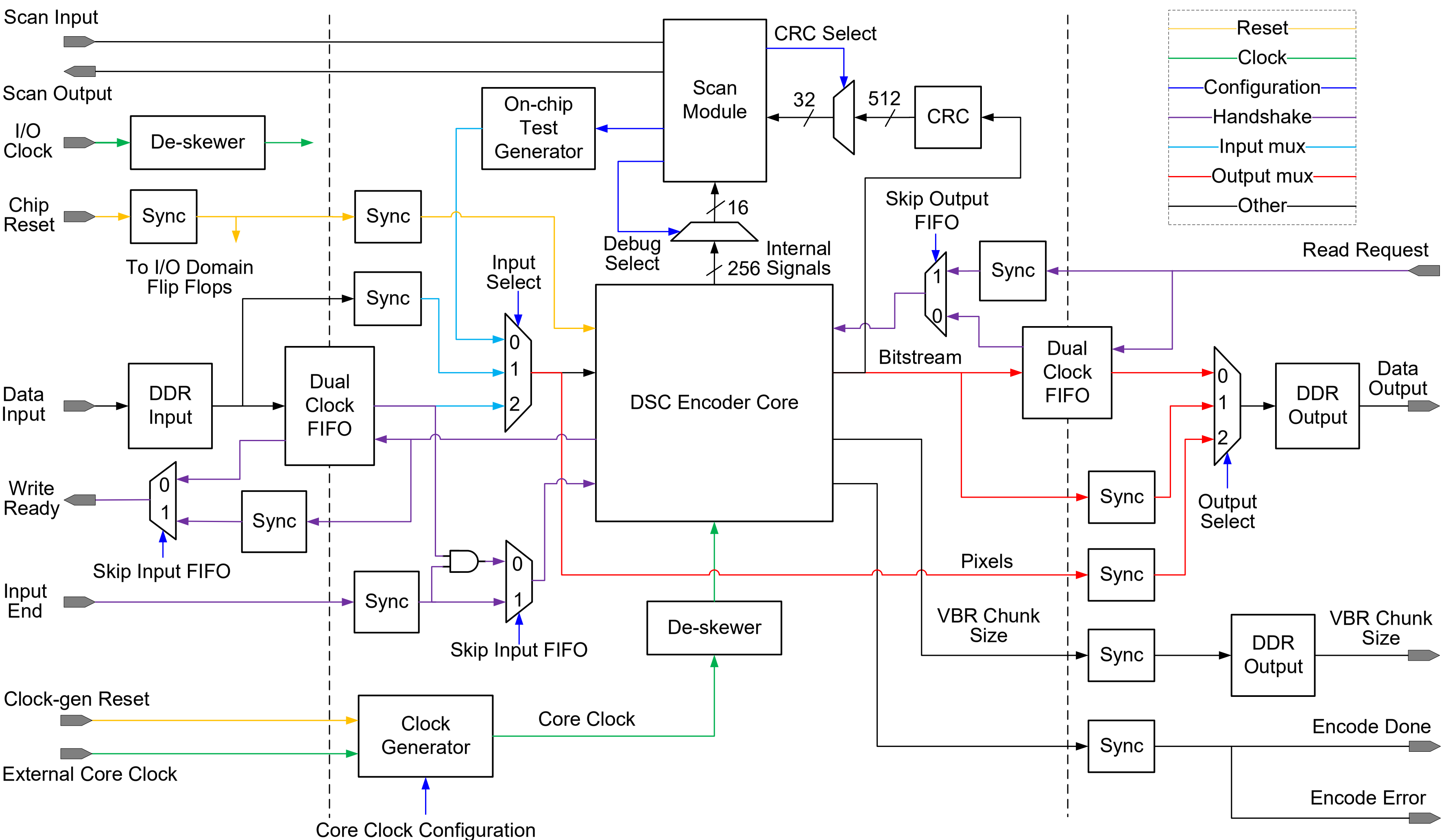


Figure 4.1: Chip-level block diagram of the DSC encoder chip.

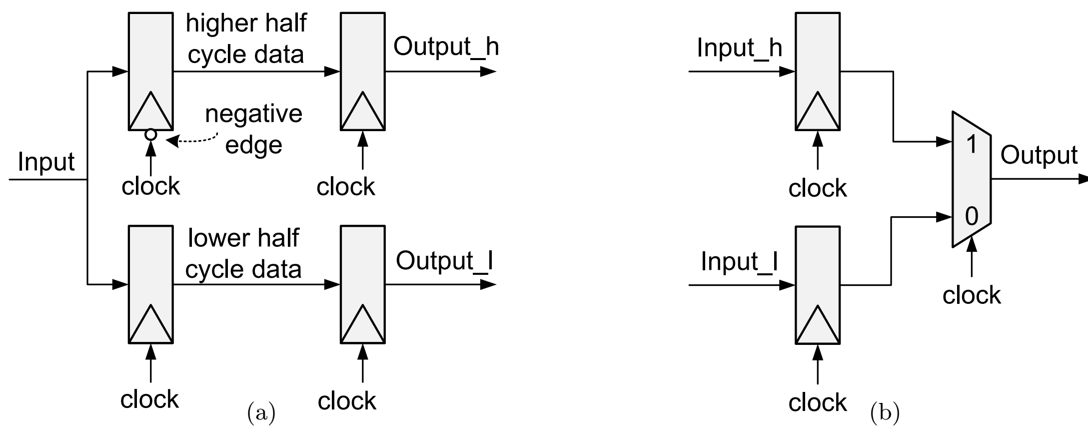


Figure 4.2: Double data rate I/O. (a) Input logic. (b) Output logic.

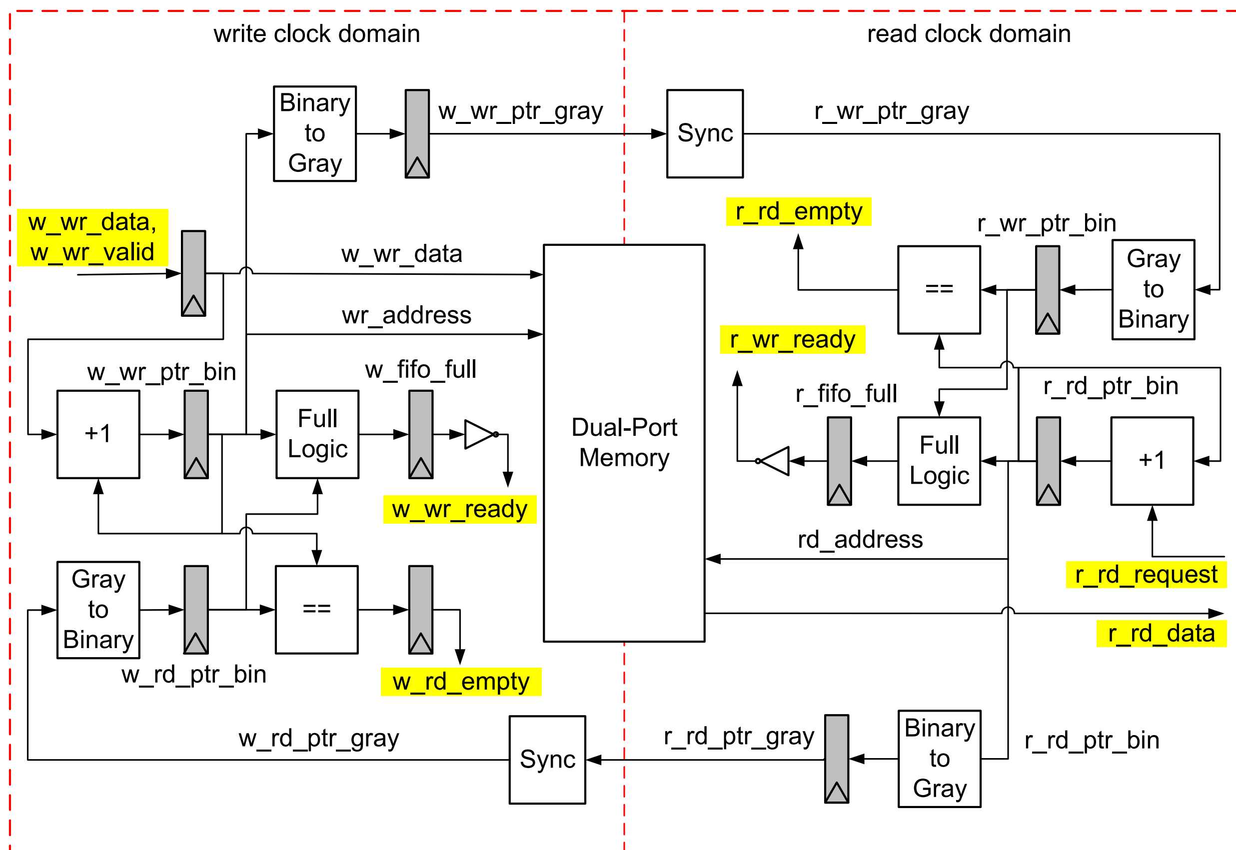


Figure 4.3: Dual-clock FIFO circuit.

clock is high, and lower half data to the output when clock is low.

4.2.2 Dual-Clock FIFO

The dual clock FIFO transmits multiple-bit data safely across two clock domains. Figure 4.3 illustrates the circuit of the dual-clock FIFOs used in the DSC encoder chip. The dual-clock FIFO has several similarities to the synchronous FIFO in Section 3.1.7: 1) a dual-port memory is used as the FIFO storage which supports a write and a read at the same time; 2) write and read pointers are one bit wider than the addresses and are used for FIFO full and empty checks. On the other hand, the dual-clock FIFO differs from the synchronous FIFO in two main aspects: 1) the write side is clocked by the source clock, while the read side is clocked by the destination clock, which is asynchronous to the source clock; 2) the read and write pointers are converted to gray code [36] before being synchronized to the other clock domain, and converted back to binary format before being used for FIFO full and empty checks. After a value is written to the FIFO memory, it will not be read out until the new write pointer value reaches the read side after synchronization, which gives sufficient time for the write to complete; therefore, although the read is in an asynchronous clock domain, it always reads the correct value out.

Table 4.1 shows the binary and gray codes for 3-bit numbers. The gray codes for two consecutive numbers differ by one bit only. The read and write pointers either stay the same or increment by one in a cycle; therefore, the synchronized gray pointer in the destination clock domain is either the unchanged value or the new incremented value. The conversion from binary code to gray code is calculated by shifting the binary code to the right for one bit and then using a bit-wise XOR operation with the original binary code:

$$gray_code = binary_code \; XOR \; \{1'b0, \; binary_code[W-1:1]\} \tag{4.1}$$

where:

- W is the bit-width of the code.

The gray code to binary code conversion is performed as:

$$binary_code[W-1] = gray_code[W-1] \tag{4.2}$$

$$binary_code[W-2:0] = gray_code[W-2:0] \; XOR \; binary_code[W-1:1] \tag{4.3}$$

4.2.3 On-Chip Test Generator

The on-chip test generator produces PPS and pseudo-random pixel values for the encoder core. The PPS can be configured for: 1) different pixel modes and component bit depths; 2) one,

Table 4.1: Binary Code and Gray Code for 3-bit Numbers

Number	Binary Code	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

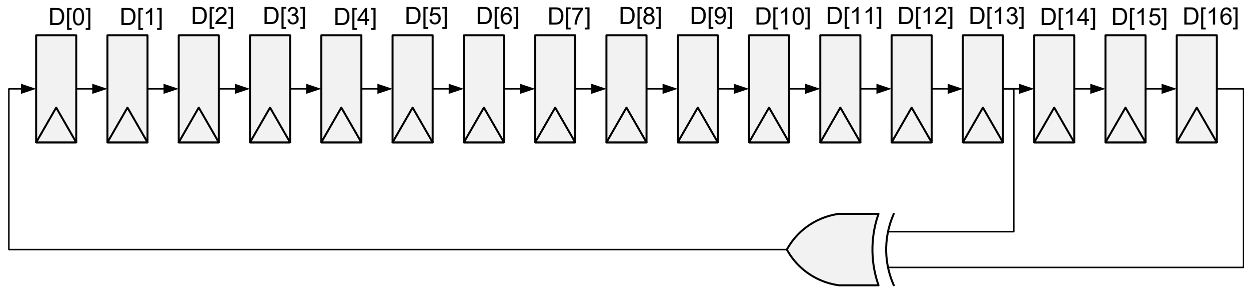


Figure 4.4: Circuit of the 17-bit linear-feedback shift register. The reset logic is omitted.

two, and four slices per line; 3) constant and variable bit rate modes. All other PPS variables are hard-coded. The frame size is hard-coded to 4K. The pixel values are generated using a maximum-length 17-bit linear-feedback shift register (LFSR) [37], which is able to generate $2^{17} - 1$ different combinations of 17-bit values. The feedback polynomial of this LFSR is:

$$f = x^{17} + x^{14} + 1 \tag{4.4}$$

Figure 4.4 shows the circuit implementation of this LFSR. In the reset stage, all the bits are set to zero, except the fifth ($D[4]$) to eighth ($D[7]$) bits, which are set to the value of a configurable 4-bit seed. After the reset stage, the bits shift right by one flip-flop per cycle, and the new value of the leftmost bit is an XOR of the previous value of the 14th ($D[13]$) and 17th ($D[16]$) bits. In each cycle, two pixels are constructed and sent to the encoder core. The component width number of

bits is taken from the LFSR value for each sample. Since the LFSR value is pseudo-random, the pixel values can be constructed in different ways while being pseudo-random. In this design, the value of the two pixels are constructed as follows. In native 4:2:2 mode, the two pixels ($p0$ and $p1$) of 10 bpc generated per cycle are constructed as:

$$p0 = \{D[12:3], D[11:2], D[10:1], D[9:0]\} \tag{4.5}$$

$$p1 = \{D[16:7], D[15:6], D[14:5], D[13:4]\} \tag{4.6}$$

In other modes, the fourth component is set to zero and the two pixels of 10 bpc are constructed as follows:

$$p0 = \{10'b0, D[11:2], D[10:1], D[9:0]\} \tag{4.7}$$

$$p1 = \{10'b0, D[16:7], D[15:6], D[14:5]\} \tag{4.8}$$

4.2.4 Memories

Table 4.2 shows the information of all memories in the DSC encoder chip. The input buffer is implemented with dual-bank, dual-port LVT SRAM, whereas the line buffer and rate buffer are implemented with dual-bank, single-port LVT SRAM. All other memories use flip-flops. The total memory size is 50 KB, including 35.5 KB SRAM.

The input buffer and line buffer are implemented using the component-wise memory architecture. The input buffer stores half a picture line of original pixels, where each bank of the two luma components consists of 256×20-bit words—each word stores two luma samples. Each bank of the chroma components stores 512×40-bit words, where each entry stores four chroma samples. Therefore, the total size of the input buffer is:

$$input_buffer_size = (512 \times 40 + 256 \times 20 + 256 \times 20) \times 2 = 61,440\ bits = 7.5\ KB \tag{4.9}$$

The line buffer consists of four 256×40-bit words for the two banks of luma components, as well as two 512×88-bit words for chroma components. Since the proposed encoder supports up to 10 bpc, the maximum bit-width of a chroma sample is 11 bits, which is one more than the luma sample. The total size of the line buffer is:

$$line_buffer_size = (256 \times 40 + 512 \times 88 + 256 \times 40) \times 2 = 131,072\ bits = 16\ KB \tag{4.10}$$

The 12 KB rate buffer consists of two banks of 256×192-bit words, where each word stores four 48-bit mux words.

The balance and size FIFOs for different columns of slices have different writes and reads

Table 4.2: Memory Blocks in the DSC Encoder Chip

Memory Name	Memory Type	Size	Information
Dual-Clock FIFO (Pixel)	flip-flop	32×80-bit (0.3125 KB)	original pixels, 2 pixels per entry
Dual-Clock FIFO (Bitstream)	flip-flop	32×32-bit (0.125 KB)	encoded bitstream, 32 bits per entry
Pixel Buffer	flip-flop	2×21×42-bit (0.226 KB)	21 look-ahead pixels per parallel slice for flatness determination
ICH	flip-flop	2×32×42-bit (0.344 KB)	32 history pixels per parallel slice
Balance FIFO	flip-flop	16×128×48-bit (12 KB)	syntax elements
Size FIFO	flip-flop	16×128×6-bit (1.5 KB)	size of syntax elements
Input Buffer	dual-port SRAM	7.5 KB	original pixels
Line Buffer	single-port SRAM	16 KB	previous line reconstructed pixels
Rate Buffer	single-port SRAM	12 KB	mux words

due to different encode timing between slices; therefore, they cannot be merged together and thus must be implemented as independent memories. Within each each column of slice, the size FIFOs of the three or four substreams always have the number of writes and reads and can be merged together. However, the balance FIFOs of the three or four substreams have different number of writes and reads due to two reasons: 1) in the write side, the size of syntax elements can vary significantly and write only occurs when there is enough bits for one mux word; 2) a read occurs only when there is a mux word request, which varies between different substreams. As a result, a separate pair of balance FIFO and size FIFO is used for each substream; therefore, in total 16

balance FIFOs and 16 size FIFOs are used to support four slices per line. The number of entries for both FIFOs is set to 128. All balance and size FIFOs are implemented as flip-flops, which results in smaller total area than SRAM.

The component-wise memory architecture results in 25% memory size reduction for input buffer and line buffer. Compared to the conventional approach that uses dedicated memory for different slices, the dynamic memory allocation in line buffer and rate buffer reduces memory size by 50% and 24%, respectively. In total, 48% memory size reduction is achieved in these three memories.

4.3 Physical Design

The proposed encoder chip is implemented using a standard-cell-based design flow. Standard threshold voltage (Vt) cells are used for logic, and low Vt cells are used for SRAM modules. The physical design of the encoder chip uses the following tools: 1) Cadence Innovus 18.13 for automatic placement and routing, 2) Calibre for design rule checking (DRC) and layout versus schematic (LVS) checks, and 3) Cadence Virtuoso for final layout edits.

4.3.1 Physical Design Flow

Figure 4.5 illustrates the physical design flow of the DSC encoder chip, which consists of seven major steps: 1) placement and routing of the encoder core which generates a GDSII layout and macro block with optimized area and clock frequency after the setup and hold timing requirements are met; 2) DRC and LVS on the core layout; 3) chip-level placement and routing which uses the core macro block and I/O cells; 4) DRC and LVS on the chip-level layout; 5) adding dummy fills to the chip layout so that it can meet density requirements, and inserting a sealring to the outer boundary of the chip; 6) DRC check on the final design which includes the chip layout, dummy fill, and sealring; 7) chip tapeout and fabrication.

The core placement and routing flow includes: floorplanning, power planning, clock tree synthesis (CTS), detailed routing, and layout export. The design is optimized before and after clock tree synthesis, and after detailed routing. A single netlist from synthesis that instantiates the encoder core and the SRAM macro blocks is read into Innovus. In the floorplanning step, the core is set to approximately square with its size fixed during the run, SRAM macros are placed in the corners, the core clock generator (oscillator) and clock de-skewers are placed inside the fences on the

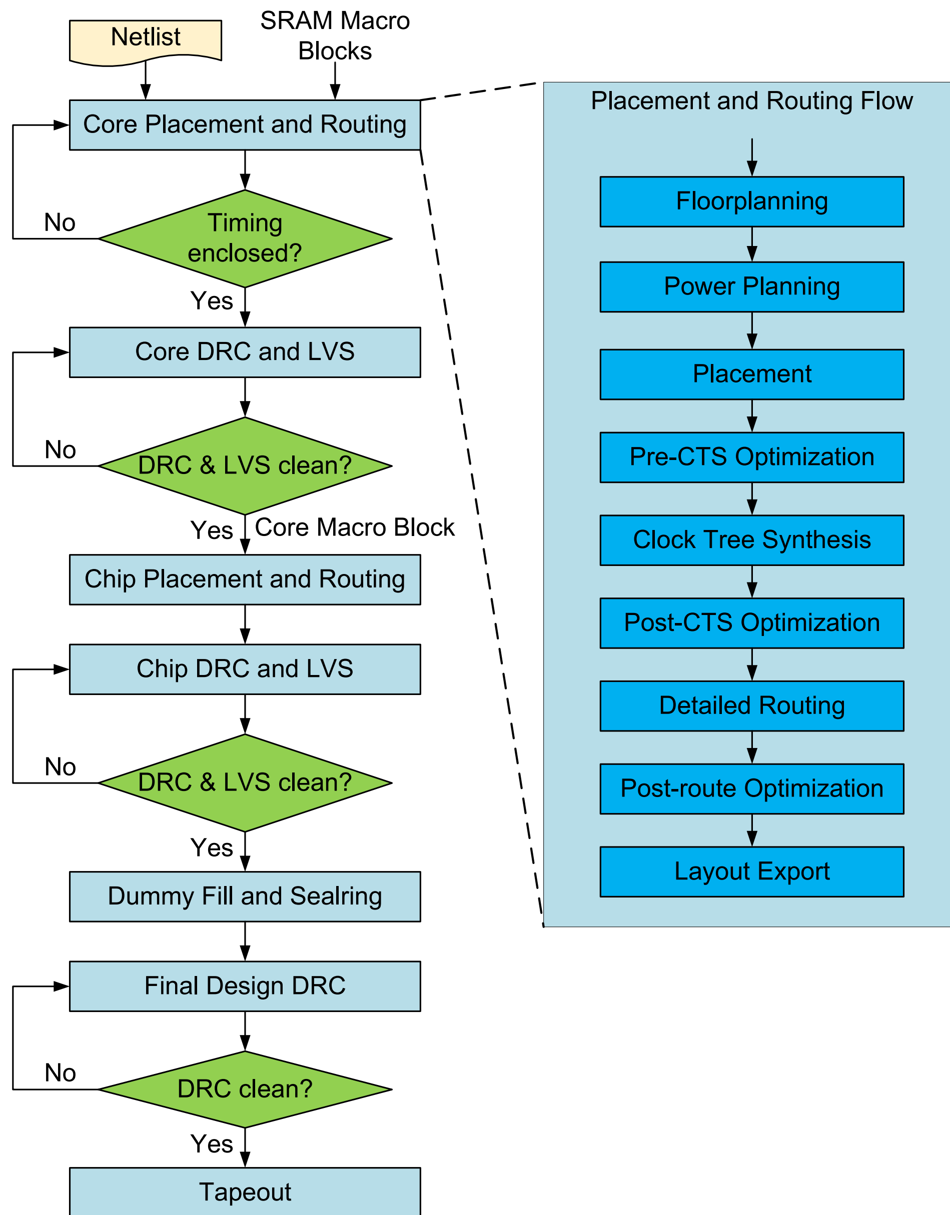


Figure 4.5: Physical design flow of the DSC encoder chip.

top of the core. Regarding power planning, the top two layers of metals are used for power grid since they are thicker and can be wider than lower level metals, which means smaller resistance and thus smaller voltage drop across the chip. The topmost level metal is dedicated for power distribution and not used for regular signal routing; therefore, it is made wide enough to reduce resistance. On the other hand, the second topmost metal is connected to lower level metals all the way down to the first layer of metal; therefore, the width of power stripes of this metal is set smaller to avoid blocking signal routing. In clock tree synthesis, a metal width larger than the minimum metal width

is used for clock tree routing to make it more robust. Timing and core area are optimized through multiple iterations of placement and routing with reduced core clock period at a stepsize of 0.1 ns and core dimension at a stepsize of about five microns, in which clock frequency has priority over core area. After the optimal placement and routing result is achieved, a GDSII layout and a macro block in LEF abstract format, as well as gate-level netlists are exported.

The core GDSII layout is imported to Cadence Virtuoso and checked for DRC and LVS using Calibre. The layout is edited manually to fix DRC violations. Once the layout is DRC clean, a LVS check is performed to make sure the layout matches the netlist exported from Innovus after placement and routing. Whenever the layout is edited, a new iteration of DRC and LVS check is required to make sure the design is DRC and LVS clean.

The chip-level placement and routing assembles the encoder core and I/O cells and pads. It places the encoder core as a macro block in the center of the layout. The I/O cells and pads are placed evenly on the four edges of the chip. Two pairs of power rings are inserted for power distribution: 1) a pair of block ring is placed outside the encoder core macro block, 2) a pair of core power ring for core power domain of the entire chip is added between the core area and the I/O cells. Since the I/O cells form a ring for the I/O power and ground, there is no need to add extra power rings for the I/O power domain. The chip core area outside the encoder core macro block is filled with filler cells and decoupling capacitor (decap) cells, as well as output drivers for the output I/O cells. Since there are no sequential elements, clock tree synthesis and timing enclosure are not included in the chip placement and routing. The chip layout in GDSII format is exported from Innovus after placement and routing, which is then imported to Virtuoso for DRC and LVS check.

In order for the design to meet density requirements, dummy fills are inserted. In addition, a sealring is added to the outer boundary of the chip. A final round of DRC check is performed on the final layout to make sure the design is DRC clean. Finally, the design is ready for tapeout and sent for fabrication.

4.3.2 Physical Design Results

Figure 4.6 shows the encoder core floorplan in Innovus. The SRAM modules for input buffer, line buffer, and rate buffer are placed in the bottom-left, upper-right, and bottom-right corner of the core. The on-chip oscillator for core clock is placed in the middle of the top edge of

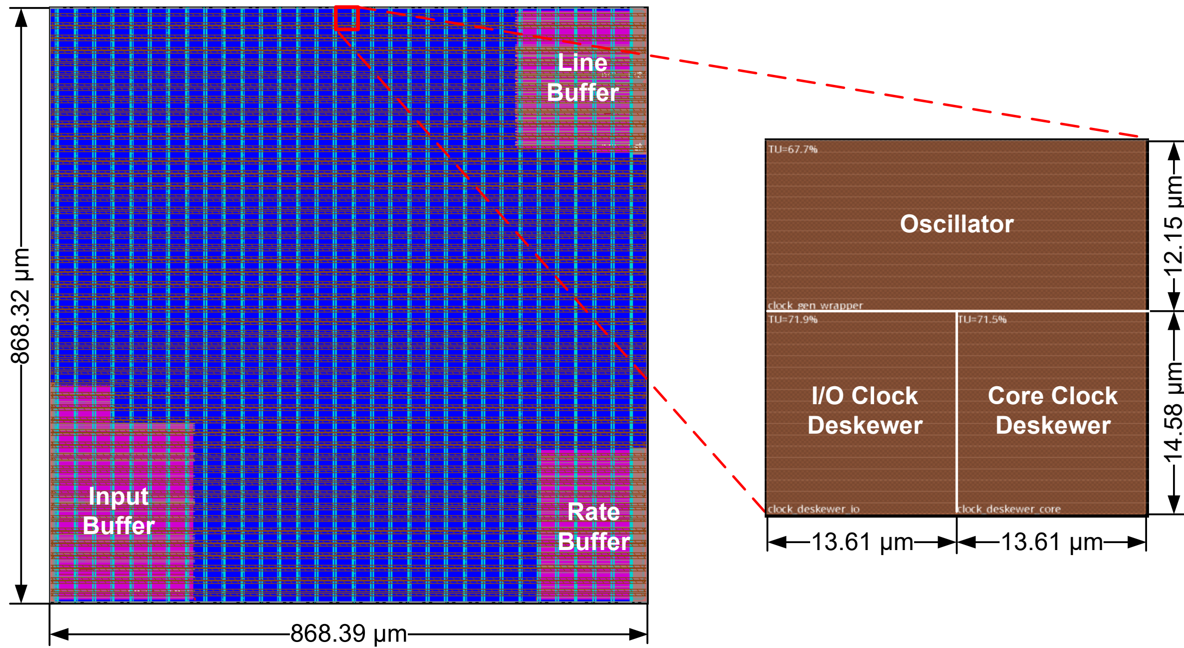


Figure 4.6: DSC encoder core floorplan. The SRAM modules of input buffer, line buffer, and rate buffer are placed in the bottom-left, upper-right, and bottom-right corners, respectively.

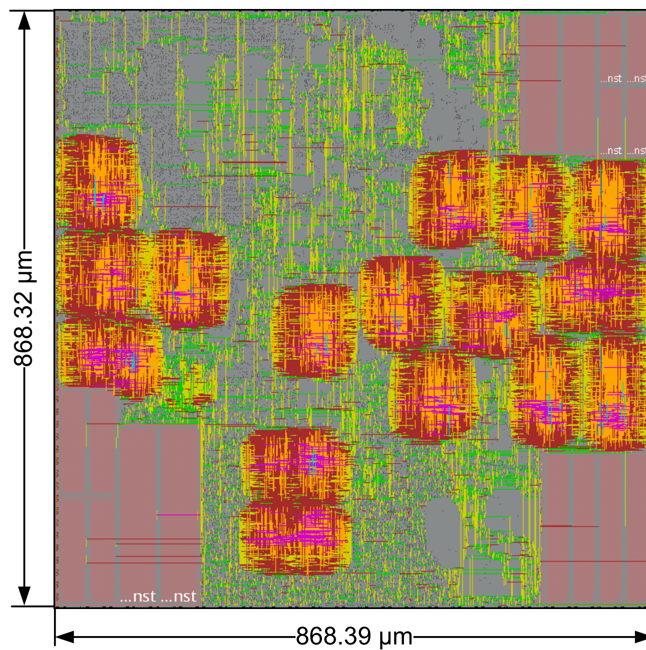


Figure 4.7: DSC encoder core layout showing only the clock nets. The clusters in the middle and bottom of the layout are clock nets for the balance FIFOs, which are implemented as flip-flops.

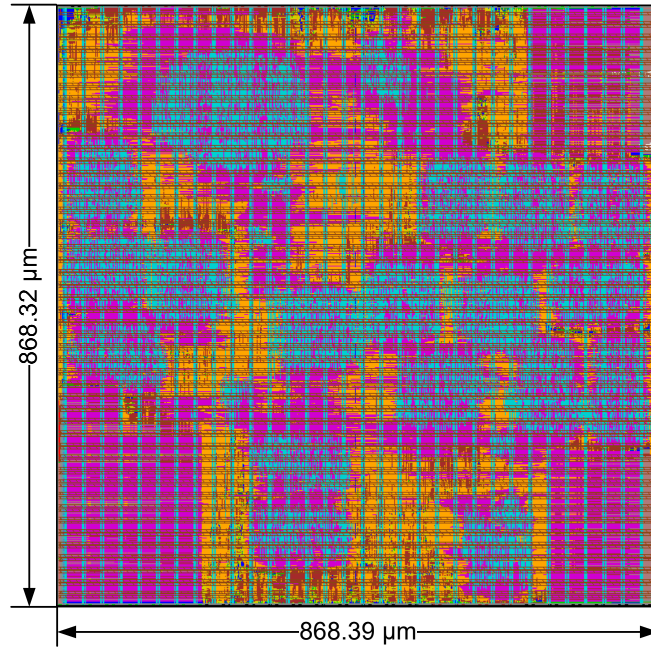


Figure 4.8: DSC encoder core layout.

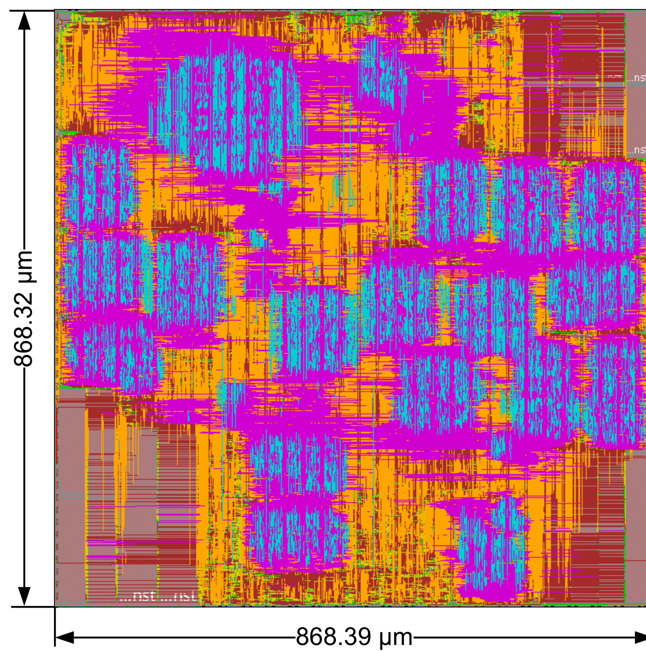


Figure 4.9: DSC encoder core layout with power and ground nets hidden.

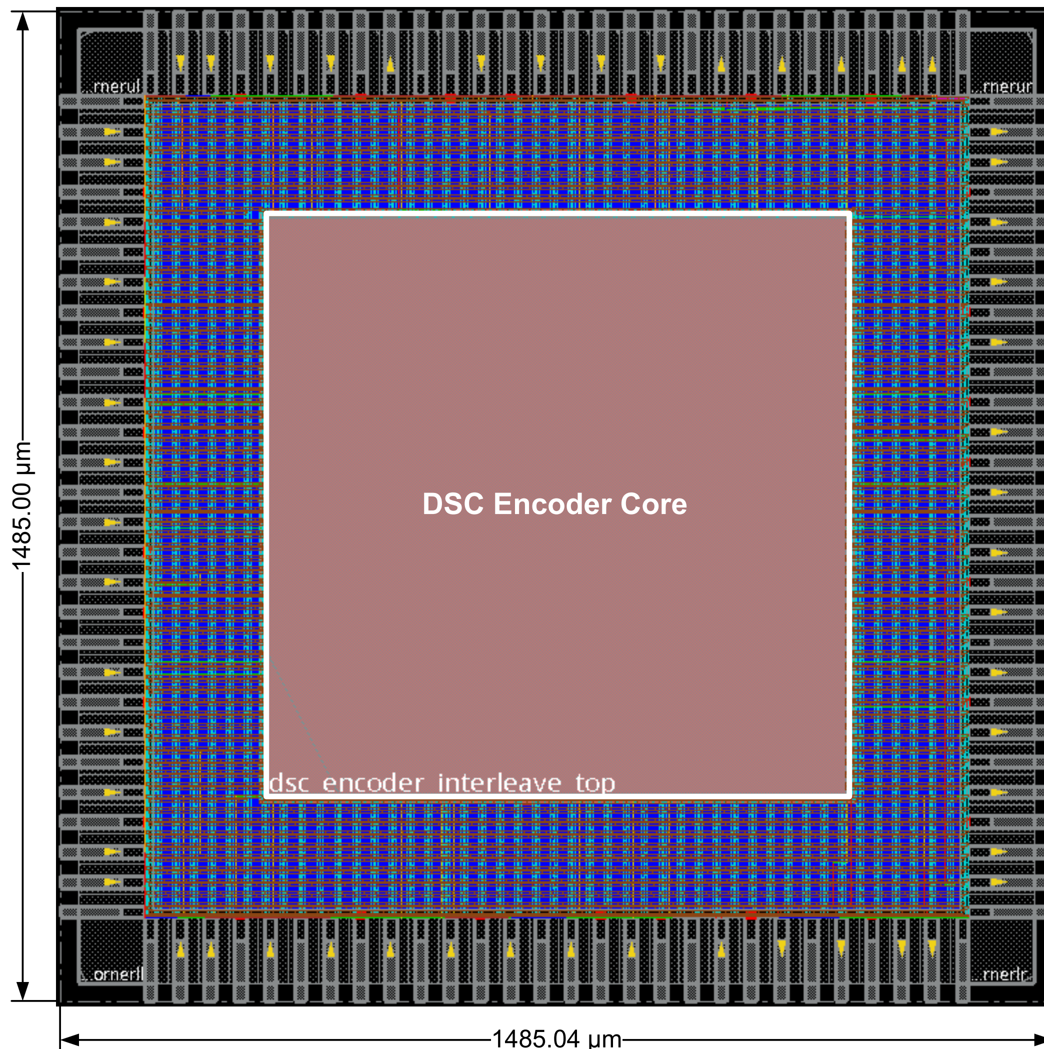


Figure 4.10: DSC encoder chip layout. The I/O pads are not shown in this plot. The sealring is not included in this plot.

the core, with two clock de-skewers below it. Figure 4.7 shows Innovus plot of the encoder core with only the clock signals visible. Figure 4.8 shows the final layout of the encoder core. Figure 4.9 shows the encoder core layout with the power and ground nets hidden. Figure 4.10 shows the layout of the entire chip. The pads are placed on top of the I/O cells and are not shown in this plot.

Figure 4.11 shows the logic area breakdown of the encoder core. The picture parameter set (PPS) module occupies less than 2% of the total logic area, whereas the indexed color history (ICH) module is the largest module and occupies 30.8% of the total logic area. One of the main contributors of the large logic area utilization is the search for the best ICH entries for all three pixels within the group, which is calculated in parallel in the first cycle of each group. The total

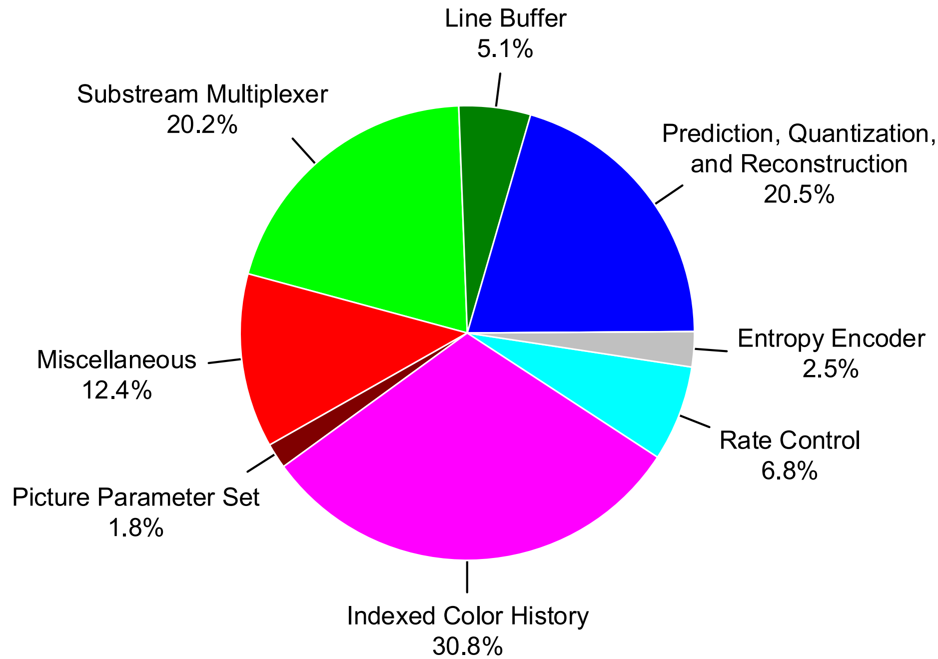


Figure 4.11: DSC encoder core logic area breakdown.

logic area of ICH, prediction, quantization, and reconstruction accounts for over half of the total core logic area. The logic area for line buffer includes the bit-width reduction logic for every sample, and the memory read and write interfaces. The miscellaneous logic includes RGB to YCoCg-R conversion, mapping quantization parameter to quantization level, core controllers and parameter calculations, etc.

Compared to a conventional DSC encoder that uses four separate slice encoders to support four slices per line, the combinational logic sharing in the proposed encoder reduces logic area by 1.75 times.

4.4 Chip and Testing

4.4.1 Chip

The DSC encoder chip is fabricated in TSMC 28 nm LP9M CMOS technology. Figure 4.12 shows the die micrograph of the chip. The core clock generator, memory blocks, and logic are annotated. The chip occupies 2.34 mm^2 die area, with a core area of 0.754 mm^2. The chip contains a total of 112 I/O pads which are evenly placed at the four edges. The chip is encapsulated using a CQFP144 package with pads on the top of the die.

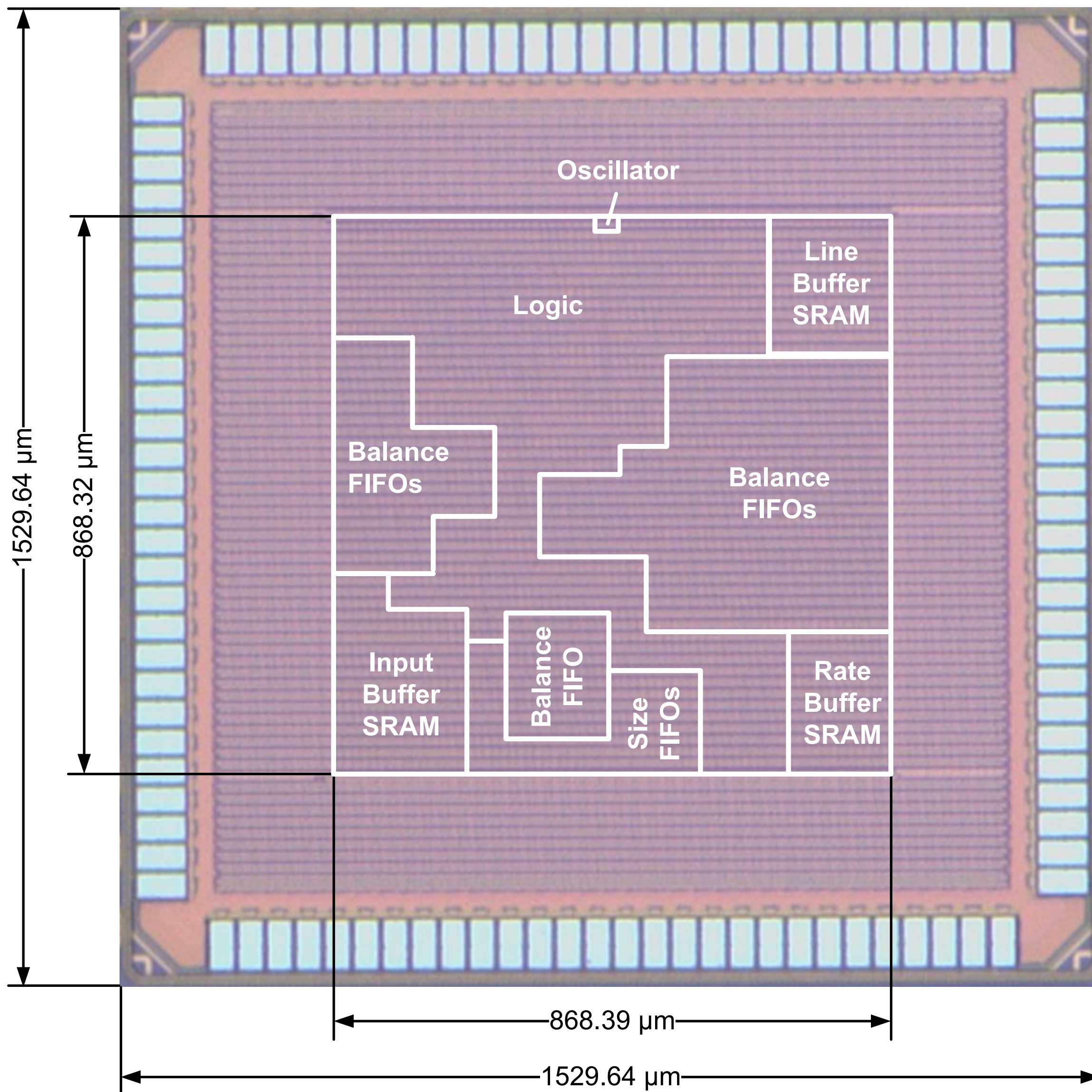


Figure 4.12: DSC encoder chip die micrograph.

4.4.2 Testing

Figure 4.13 illustrates the testing setup for the DSC chip, which consists of a desktop computer (PC), a Xilinx VC-709 FPGA board, a PCB with the packaged chip, two external power supplies, and an oscilloscope. The packaged chip is attached to the PCB through the socket. The FPGA board is used for interfacing between the PC and the PCB. The PC communicates data with the FPGA board through a PCI express (PCIe) connection, whereas the FPGA board communicates with the PCB board through its FPGA Mezzanine Card (FMC) interface. Since the PCB does not contain a FMC interface, a FMC adapter is used as the bridge between the FPGA and PCB. Two

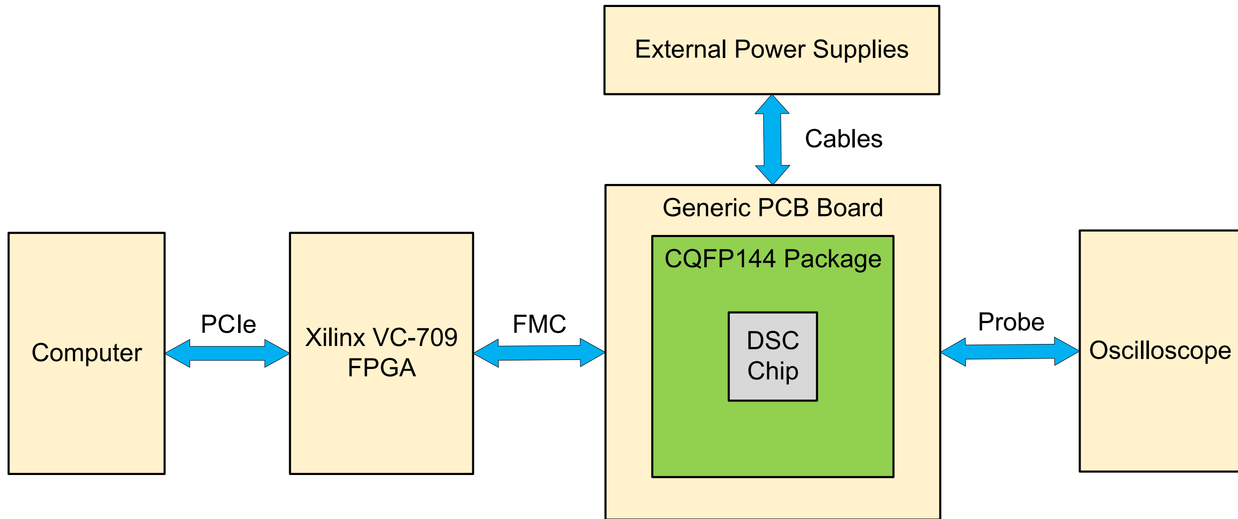


Figure 4.13: Testing setup for the DSC encoder chip.

external power supplies are connected to the power (I/O and core) and ground pins on the PCB. An oscilloscope probes signals and measures core clock frequency.

The chip is tested using the test cases generated by the on-chip test generator, which guarantees the chip can operate at its maximum clock frequency. The testing process includes six main steps: 1) the PC sends configuration parameters to the FPGA, which passes them to the chip through the scan module one bit at a time; 2) the clock generator is reset, after which the core clock is active; 3) the chip is reset and then the on-chip test generator sends the pseudo-random pixels to the encoder core; 4) the chip encodes the pixels into a bitstream; 5) Once the encoding is done, the CRC values are scanned out from the chip and sent back to the PC; 6) the CRC value is compared with the pre-computed reference value to determine whether the outputs are correct. The oscilloscope measures the frequency of a divided core clock, from which the actual core frequency is calculated. The I/O voltage is fixed to 1.80 V, whereas the core voltage is varied from 1.15 V down to 0.80 V with a stepsize of 0.05 V. The corresponding maximum frequency and current are measured for one, two, and four slices per line at 4:4:4, 4:2:2, and 4:2:0 pixel modes.

4.5 Measured Results

The measured clock frequency and current data are used to calculate throughput and energy per pixel. Throughput is calculated based on one pixel per cycle for one slice per line and

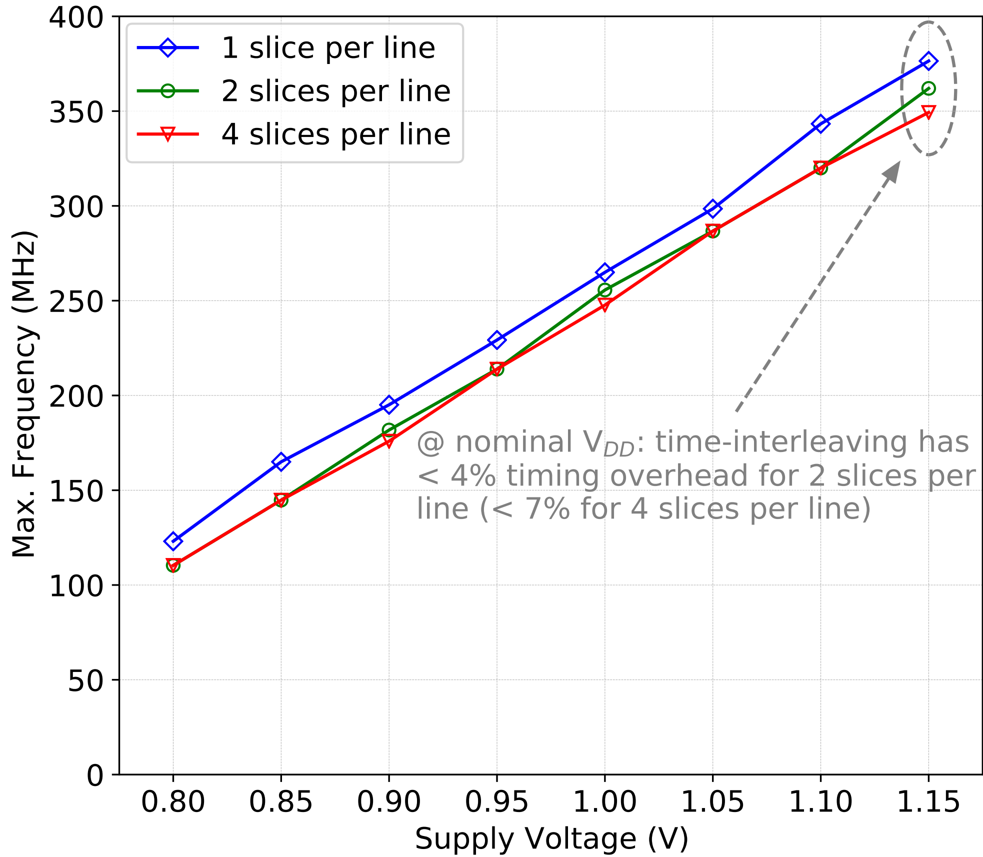


Figure 4.14: Maximum core clock frequency versus supply voltage of the DSC encoder chip.

two pixels per cycle for two and four slices per line. Energy per pixel is calculated by dividing the power consumption by the number of pixels encoded per second.

Although the pseudo-random images may not contain as much spatial and temporal correlations as other images (e.g., natural images), the encoder performance for both type of images are expected to be similar. First, DSC performs intra-frame compression which does not consider temporal correlations. Second, regardless of image contents, DSC compresses each image on a group-by-group basis, which includes the same operations and requires the same amount of processing time.

4.5.1 Throughput

Figure 4.14 plots the maximum achievable frequencies at various voltages for one, two, and four slices per line, which are 376 MHz, 362 MHz, and 349 MHz at 1.15 V, respectively. Due to the

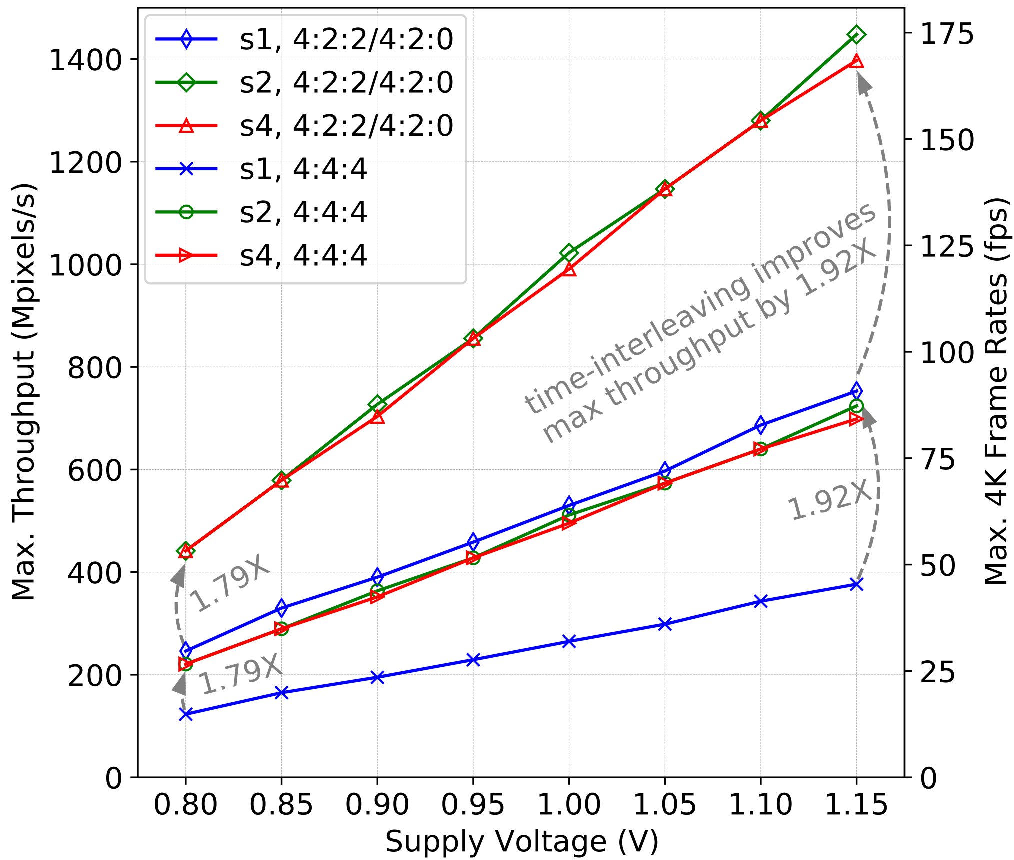


Figure 4.15: Maximum throughput versus supply voltage of the DSC encoder chip.

register selection multiplexers in the time-interleaved architecture, the maximum achievable clock frequency is smaller compared to that of the non-interleaved architecture. The timing overhead (i.e., percentage of maximum frequency reduction) is less than 4% and 7% for two and four slices per line, respectively.

Figure 4.15 shows the maximum throughput and the corresponding frame rates at 4K. The throughput at 1.15 V while encoding 4K videos is up to 724 megapixels per second (Mpixels/s) (i.e., 87 frames per second (fps)) for simple 4:4:4 mode, and 1448 Mpixels/s (i.e., 174 fps) in native 4:2:0/4:2:2 modes. Time-interleaved encoding at two slices per line improves the throughput by 1.79 times at 0.80 V and 1.92 times at 1.15 V.

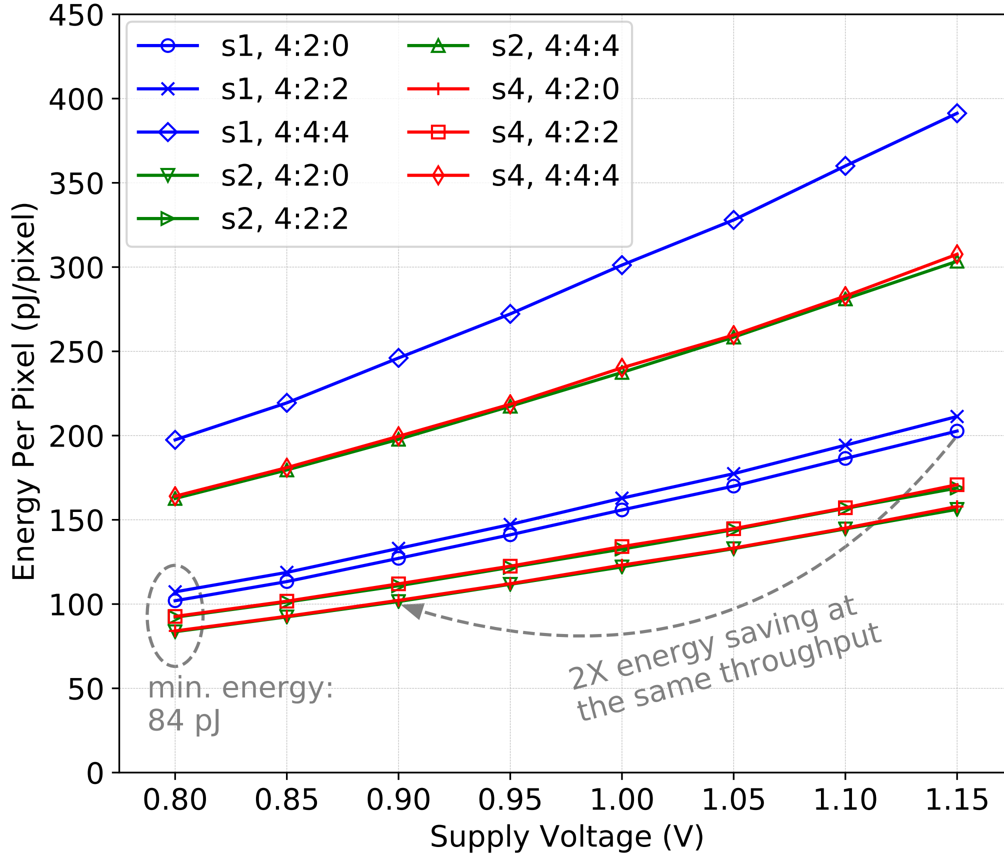


Figure 4.16: Energy per pixel versus supply voltage of the DSC encoder chip.

4.5.2 Energy Efficiency

Figure 4.16 plots the energy per pixel for one, two, and four slices per line in 4:2:0, 4:2:2, and 4:4:4 modes. The minimum energy of 84 pJ, 92 pJ, and 163 pJ per 4:2:0, 4:2:2, and 4:4:4 pixel is achieved at 0.80 V. When supply voltage is reduced from 1.15 V to 0.80 V, the energy per pixel is reduced by 1.97–1.99 times, 1.83–1.87 times, and 1.84–1.88 times for one, two, and four slices per line, respectively. At the same voltage, encoding a 4:4:4 pixel consumes 1.93–1.95 times more energy than a 4:2:0 pixel since a 4:2:0 container pixel includes two original pixels, whereas a 4:2:2 pixel consumes 1.04–1.10 times more energy than a 4:2:0 pixel since a 4:2:2 container pixel has one more component than a 4:2:0 container pixel. Thanks to the computation logic sharing, time-interleaved encoding in two slices per line consumes 14%–23% lower energy per pixel than non-interleaved encoding (i.e., one slice per line) at the same voltage. The serial slice processing in four slices per line incurs less than 1.5% energy overhead compared to two slices per line. For a

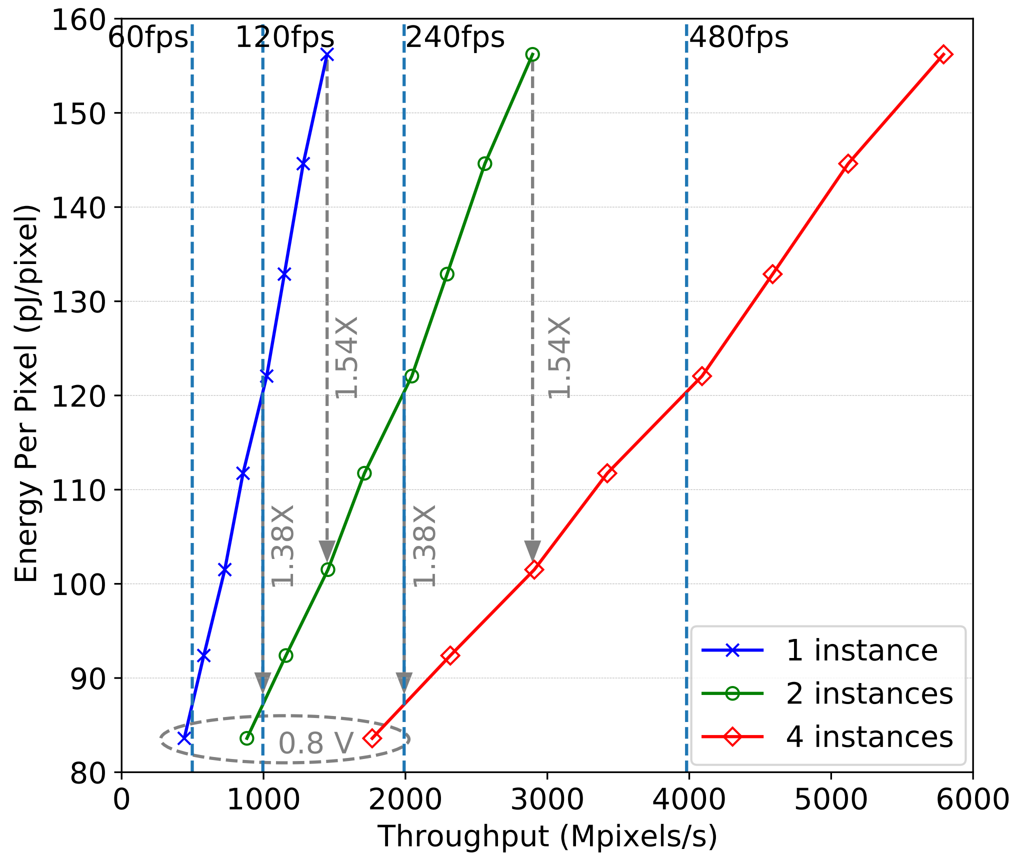


Figure 4.17: Multiple encoder instances scalability with 1, 2, and 4 encoder instances. Each curve plots the energy per 4:2:0 pixel versus throughput at various voltages. The lowest data points are for 0.8 V, whereas the highest data points are for 1.15 V.

given throughput target, the time-interleaved encoding (two and four slices per line) further reduces energy dissipation compared to non-interleaved encoding through voltage scaling. For example, at the same throughput, time-interleaved encoding lowers energy per 4:2:0, 4:2:2, and 4:4:4 pixel by 1.96, 1.87, and 1.94 times at 0.91 V compared to non-interleaved encoding at nominal voltage of 1.15 V.

4.5.3 Multi-Instance Scalability

Multiple DSC encoder instances encoding different columns of slices in parallel can achieve higher throughput or higher energy efficiency by voltage scaling. For example, two and four encoder instances can be used to parallelly decode four and eight columns of slices, respectively. Figure 4.17

plots the energy per 4:2:0 pixel versus throughput for one, two, and four parallel instances of the proposed encoder chip, each of which is configured to process two slices in parallel. At 1.15 V, one encoder instance achieves 1448 Mpixels/s with 156 pJ per pixel. With two encoder instances, the same throughput can be achieved at 0.9 V while only dissipating 101 pJ per pixel, which leads to 1.54 times energy saving compared to one encoder instance. For a target throughput of 120 frames per second at 4K, one encoder instance is required to run at 0.99 V with energy efficiency of 120 pJ per pixel, whereas two encoder instances meet the target at 0.82 V with 87 pJ per pixel—1.38 times lower than one encoder instance. The same energy savings are achieved by increasing the number of encoder instances from two to four. Therefore, regardless of the specific throughput target, the energy per pixel reduces by 1.38–1.54 times when doubling the number of encoder instances.

4.6 Comparison With Video Encoder Chips

Table 4.3 compares the proposed DSC encoder with published H.264/AVC and High Efficiency Video Coding (HEVC) video encoder chips that support video resolutions of at least 4K. The proposed DSC encoder has 2.7–33 times lower core area, 1.1–13.9 times lower logic gate count, and 1.6–37 times better throughput per area while being visually lossless.

4.7 Summary

This chapter presents a VESA Display Stream Compression (DSC) encoder chip which complies fully with DSC version 1.2a and supports 4K UHD video resolution. The proposed encoder chip utilizes the time-interleaved architecture that processes two slices in parallel. It achieves one pixel per cycle throughput at one slice per line, and two pixels per cycle at two and four slices per line. The chip-level architecture, physical design flow and results, chip and testing, and measured results are presented. The component-wise memory architecture with dynamic allocation reduces the total required buffer capacity for input buffer, line buffer, and rate buffer by 48%. Area efficiency improvements of up to 1.75 times are achieved compared to conventional parallel multi-slice architectures. Time-interleaved encoding improves energy efficiency by 1.87–1.96 times. Doubling the number of parallel encoder instances further increases energy efficiency by 1.38–1.54 times. This DSC encoder chip achieves throughput of 1448 Mpixels/s at 1.15 V and energy dissipation down to 84 pJ/pixel at 0.8 V, enabling integration in battery-powered portable and wearable systems.

Table 4.3: Comparison of Video Encoders with 4K Resolution or Higher

	This Work	VLSIC'12 [38]	VLSIC'13 [39]	JSSC'16 [40]
Compression Standard	DSC	H.264/AVC	HEVC	HEVC
Type of Compression	**visually lossless**	lossy	lossy	lossy
Coding	Intra-frame	Intra-frame	Intra- & Inter-frame	Intra- & Inter-frame
Technology	28 nm	65 nm	28 nm	28 nm
Voltage	0.8–1.15 V	0.8–1.2 V	NA	0.9 V
Max. Resolution	3840×2160	7680×4320	8192×4320	4096×2160
Core Area	**0.75** mm^2	2.07 mm^2	25 mm^2	2.16 mm^2
Gate Count	**627.7 Kgate**	678.8 Kgate	8350 Kgate	3558 Kgate
On-chip SRAM	35.5 KB	27.1 KB	7.14 MB	308 KB + DRAM
Max. Clock Frequency	376 MHz	260/330 MHz*	312 MHz	494 MHz
Max. Throughput	1448 Mpixels/s	1991 Mpixels/s	1062 Mpixels/s	250 Mpixels/s
Throughput/Area	**1920** $Mpps/mm^2$	962 $Mpps/mm^2$	42.5 $Mpps/mm^2$	115.7 $Mpps/mm^2$
Min. Energy/Pixel	84 pJ @ 0.8 V	24.5–53.6 pJ	667 pJ[†	320 pJ

* $Freq_{Intra}/Freq_{CABAC}$.

† Core power divided by throughput

Chapter 5

Hardware Architectures and Physical Design of Display Stream Compression Decoders

This chapter presents the hardware architectural and physical design of Display Stream Compression (DSC) decoders. Four architectures that explore one or more slices per line, and throughput-area trade off are discussed. Six decoders that are implemented in 28 nm using a standard cell design flow are used to evaluate the proposed architectures.

5.1 Slice Decoder

The *slice decoder* is to decode pictures configured into one slice per line. It achieves a throughput of three pixels per cycle (six pixels per cycle in native 4:2:2 and 4:2:0 modes), i.e., one group is processed per cycle. Figure 5.1 shows the block diagram of the slice decoder including major blocks and memories (rate buffer and line buffer).

The controllers in the slice decoder are similar to that of the slice encoder. A main difference is that the slice decoder does not use a cycle counter within the group, since it processes one group per cycle.

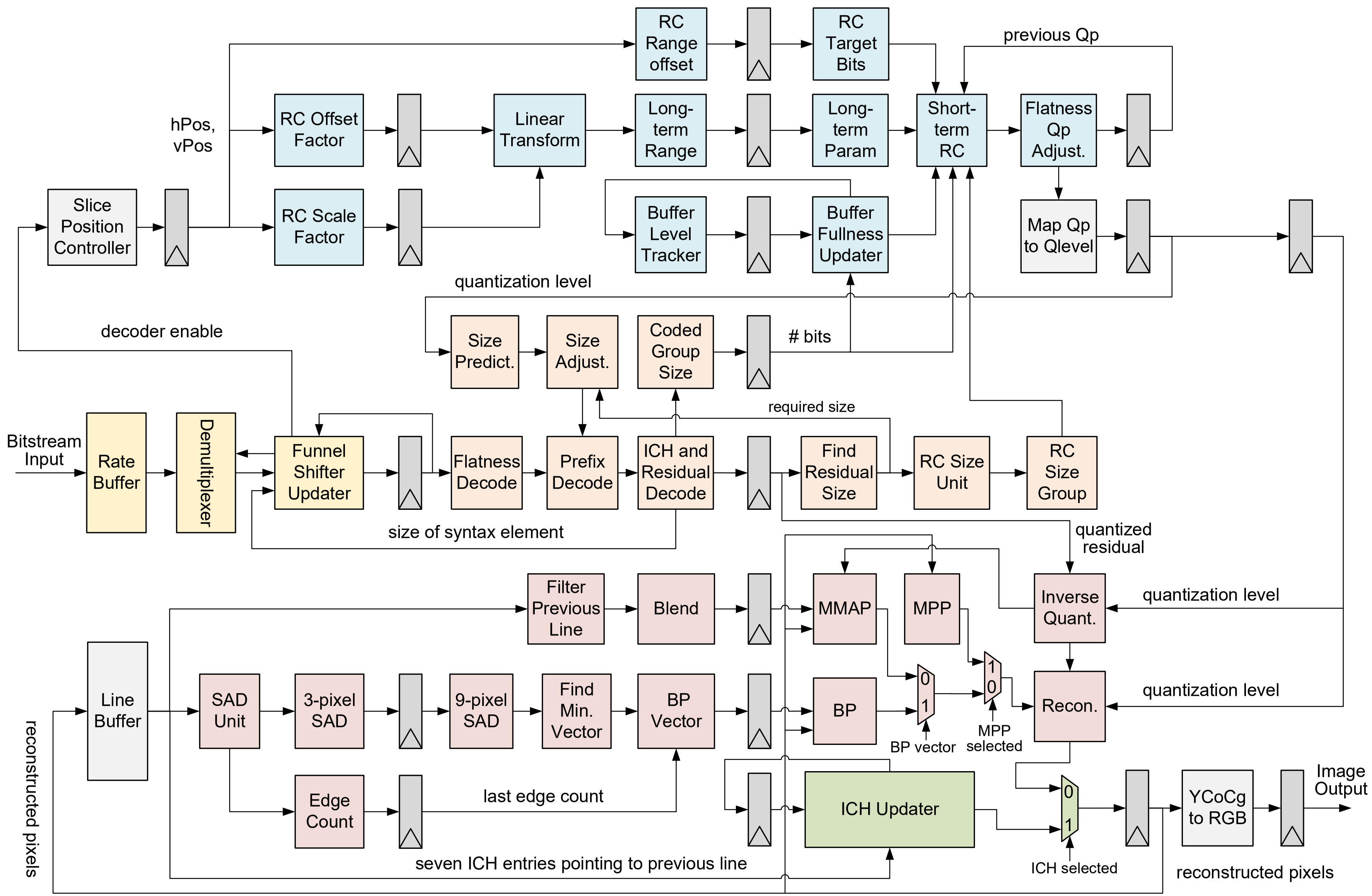


Figure 5.1: Pipelined block diagram of the slice decoder.

5.1.1 Substream Demultiplexing

The rate buffer memory has the same dimension as the slice encoder. It requires one write and one read per clock cycle and can be implemented as single-bank, dual-port memory. In every cycle, the budgeted number of bits per group enters the decoder. They are concatenated with the remaining bits from the previous cycle. If the total number of bits is enough for four mux words, they are written to the rate buffer memory. To accumulate enough data in the rate buffer to prevent buffer underflow, the read does not start until the initial decoding delay period has passed.

The slice decoder contains four parallel funnel shifters to temporarily store the bitstream to be decoded. The funnel shifter for the fourth substream is only used in native 4:2:2 mode. For each substream, a 7-bit register records the number of remaining valid bits in each funnel shifter. In every cycle, each funnel shifter sends the entropy decoder the most-recent undecoded bits whose length is equal to the maximum syntax element size. After a syntax element is decoded, the entropy decoder returns the number of bits that is used to code the component of the current group. If the number of valid bits in the funnel shifter is equal to or greater than the number of bits decoded, the decoding is successful. After a successful decoding, the funnel shifter shifts out the decoded bits, and its fullness is reduced by the number of decoded bits. If the remaining bits are not enough for a maximum size syntax element, the funnel shifter sends a mux word request to the demultiplexer. A mux word enters the funnel shifter in the same cycle and the the fullness is increased by the size of a mux word.

5.1.2 Entropy Decoder

The entropy decoder is pipelined into two stages. The three or four components are decoded in parallel using replicated hardware for each component. The decoded ICH indices, quantized residuals, and number of bits coding the group are used in the next clock cycle. The sizes of the quantized residuals are calculated and used for the size prediction for the next group.

5.1.3 Rate Control

Rate control generates the quantization parameter (Qp) for the second next group. The offset and scale factors for linear transformation are calculated in the first stage, which uses the horizontal and vertical position controllers. The buffer fullness is calculated and transformed before

being compared with thresholds to determine the long-term rate control parameters. The final Qp from short-term rate control adjustment is further adjusted based on the flatness information. The Qp is mapped to quantization level and used in the size prediction of entropy decoder, inverse quantization, and reconstruction.

5.1.4 Prediction, Inverse Quantization, and Reconstruction

The three or four components are predicted, inverse quantized, and reconstructed in parallel using replicated logic. The BP search uses the reconstructed pixels of the previous line from the line buffer and calculates 3-pixel sum of absolute differences (SAD) and the number of cycles passed since an edge occurred. In the next cycle, the 9-pixel SADs are calculated by adding the 3-pixel SAD of the current group and the previous two groups. The BP vector is determined and used in the next cycle. MMAP is partitioned into two cycles. The previous line's pixels are filtered and blended with the original pixels in the first cycle. The three samples of the component are predicted in parallel using MMAP in the next cycle. The decoder decides between BP and MMAP based on the BP vector, and decides whether to use MPP based on the MPP selection decision which is decoded in the entropy decoder.

5.1.5 Indexed Color History

The ICH in the decoder is implemented as shift registers that contain 32 registers, each of which stores one reconstructed pixel. The slice decoder updates the ICH registers once per group time, using the same method as the slice encoder. The slice decoder does not maintain the valid signals of the ICH entries, since the decoder does not search for best ICH entries. As long as the ICH is updated using the same manner as the encoder, only the valid ICH pixels are selected for an ICH-mode-coded group. The coding mode decision and ICH indices (which are set to zero if the group is P-mode-coded) are decoded from the entropy decoder. The ICH pixels for the current group are addressed by the three indices, which are used as the decoded pixels if ICH-mode is used.

5.1.6 Line Buffer

The line buffer uses the component-wise memory architecture and the same memory structure as the line buffer in the slice encoder, as detailed in Section 3.1.4. A significant difference

is that the memory in the slice decoder requires a write and a read in the same cycle; therefore, dual-port memory is required. Also, the pixels are read out from the memory in the fourth group earlier than the group in the same position of the next line.

5.2 Slice-Interleaved Decoder

The *slice-interleaved decoder* architecture [41] adds the support of decoding multiple slices per line to the slice decoder architecture using the minimum amount of additional hardware and frequency overhead.

5.2.1 Architecture

The slice-interleaved decoder supports multiple slices per line by sharing the combinational logic for calculations and uses a dedicated set of registers for each column of slice, which is the same as the resource sharing in the slice-interleaved encoder architecture in Section 3.2. The slice interleaved decoder processes one slice line of the slices in the same row from left to right, and repeats for all slice lines. Once one row of slices is decoded, it works on the next row of slices and repeats the process until all slices are decoded. When the decoder switches from one slice to another, the decoding states of the current slice are stored and resume when the decoder switches back to this slice in the next line.

The line buffer and rate buffer memory utilize the dynamic memory allocation scheme discussed in Section 3.2.1. At any point of time, the slice-interleaved decoder works on only one slice, and one read and one write is needed for both line buffer and rate buffer; therefore, they should use single-bank dual-port memory as the storage. The size of the line buffer is the total size of one picture line of pixels, regardless of the number of slices per line.

Since the ICH is invalidated at the end of a slice line when there is more than one slice per line, one instance of the ICH registers is used for all columns of slices. Dedicated funnel shifters are required for every column of slice.

5.2.2 Slice Decode Timing and Extra Buffering

Figure 5.2 shows the slice decode timing of two and four slices per line of the slice-interleaved decoder architecture. The initial decoding delay is assumed to be smaller than slice width in the

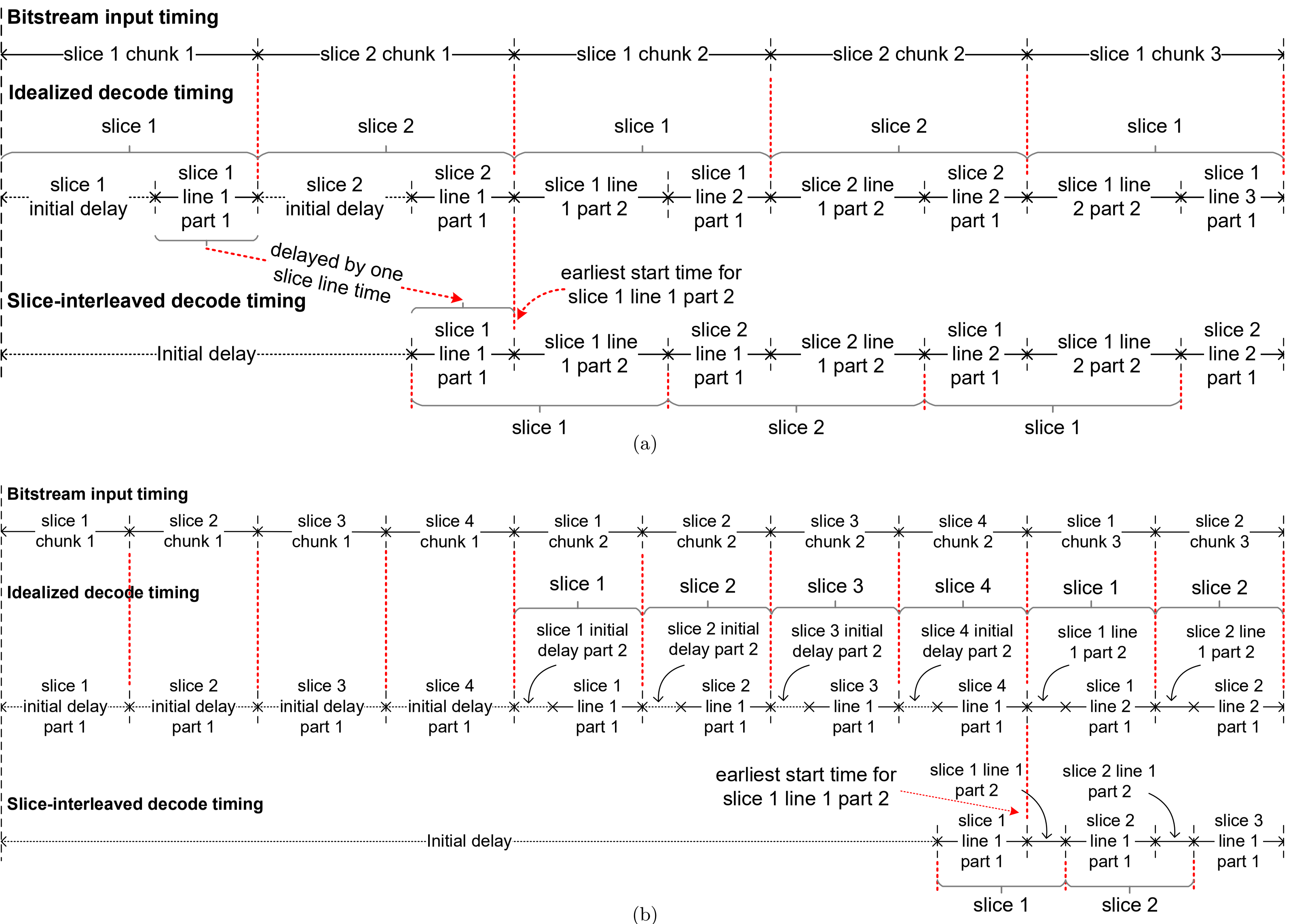


Figure 5.2: Slice-interleaved decode timing. (a) Two slices per line. (b) Four slices per line.

case of two slices per line, and larger than slice width in four slices per line. The idealized decode scheme always works on the same slice as the bitstream input and each slice is allocated an even portion of the processing time. In addition, the decoded pixels are not in raster-scan order in the idealized decode scheme. The slice-interleaved decode scheme also allocates one slice line pixel time for each slice; however it makes sure that the decoded pixels are in raster-scan order. To make sure that the rate buffer does not underflow under the slice-interleaved scheme, all pixels are decoded no earlier than they would be in the idealized decode timing. For example, slice 1 line 1 part 1 is delayed by one slice line pixel time to be contiguous with slice 1 line 1 part 2.

In two slices per line, the decoding of line 1 part 1 of both slices 1 and 2 are delayed compared to the idealized decode timing, which requires additional buffering in the rate buffer. The required amount of extra buffering in the rate buffer is:

$$\begin{aligned} rate_buffer_extra &= (slice_1_line_1_part_1 + slice_2_line_1_part_1) * bits_per_pixel \\ &= 2 * (slice_width - initial_dec_delay) * bits_per_pixel \end{aligned} \tag{5.1}$$

For four slices per line, the part 1 of the first line in all four slices are delayed in the decoding and require additional storage in the rate buffer. The amount of additional buffering is:

$$\begin{aligned} rate_buffer_extra &= 4 * slice_1_line_1_part_1 * bits_per_pixel \\ &= (8 * slice_width - 4 * initial_dec_delay) * bits_per_pixel \end{aligned} \tag{5.2}$$

In both cases, a conservative upper bound for the amount of extra buffering is one picture line worth of bitstream, i.e.,

$$rate_buffer_extra_upper_bound = picture_width * bits_per_pixel \tag{5.3}$$

For implementations that support particular configurations of slice width and initial decoding delay, the actual amount of extra buffering can be used to reduce memory size.

5.3 Parallel Slice Decoders

With increased video resolution and frame rates, the slice decoder may not be able to meet the throughput requirements. The *parallel slice decoder* architecture is designed to achieve higher throughput by configuring the pictures into multiple slices per line and decoding all slices of the same row in parallel.

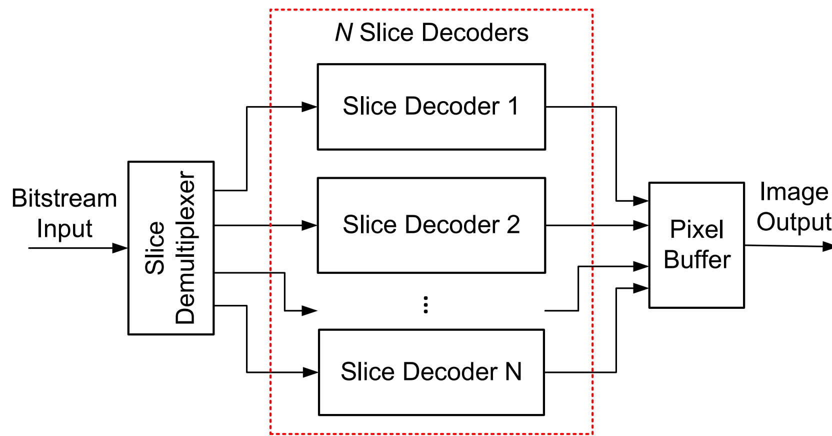


Figure 5.3: Parallel slice decoder architecture.

5.3.1 Architecture

Figure 5.3 shows the block diagram of the parallel slice decoder. It consists of multiple parallel instances of the slice decoder, each of which decodes one column of slice. A parallel slice decoder that consists of N slice decoders achieves a throughput up to $3 * N$ pixels per cycle when all slice decoders are active. The bitstream input is distributed into the rate buffer of each slice decoder by the slice demultiplexer. The pixel buffer rasterizes the decoded pixels of the slice decoders.

5.3.2 Slice Decode Timing and Extra Buffering

Figure 5.4 shows the slice timing of parallelly decoding two slices per line. Slices 1 and 2 are the left and right slices of the same row, respectively. The bitstream input rate is two groups' worth of data per cycle. The first chunk of bits (chunk 1) of slice 1 enters the decoder, followed by the first chunk of slice 2. Then the second chunk of slice 1 followed by second chunk of slice 2. This pattern repeats for all chunks of both slices. Then the future rows of slices repeat the pattern of the current row of slices until all bits have entered the decoder. The idealized decode timing processes each slice only in the time of the bitstream input and each chunk is divided into two parts. For example, during the time of slice 1 chunk 1, slice 1 part 1 is processed after an initial delay stage. The second part of slice 1 line 1 does not start until the time of slice 1 chunk 2. In the idealized decode timing, the decoder processes at a throughput of six pixels per cycle and the output pixels are not in raster-scan order.

The parallel slice decode timing is derived based on the idealized decode timing. Both

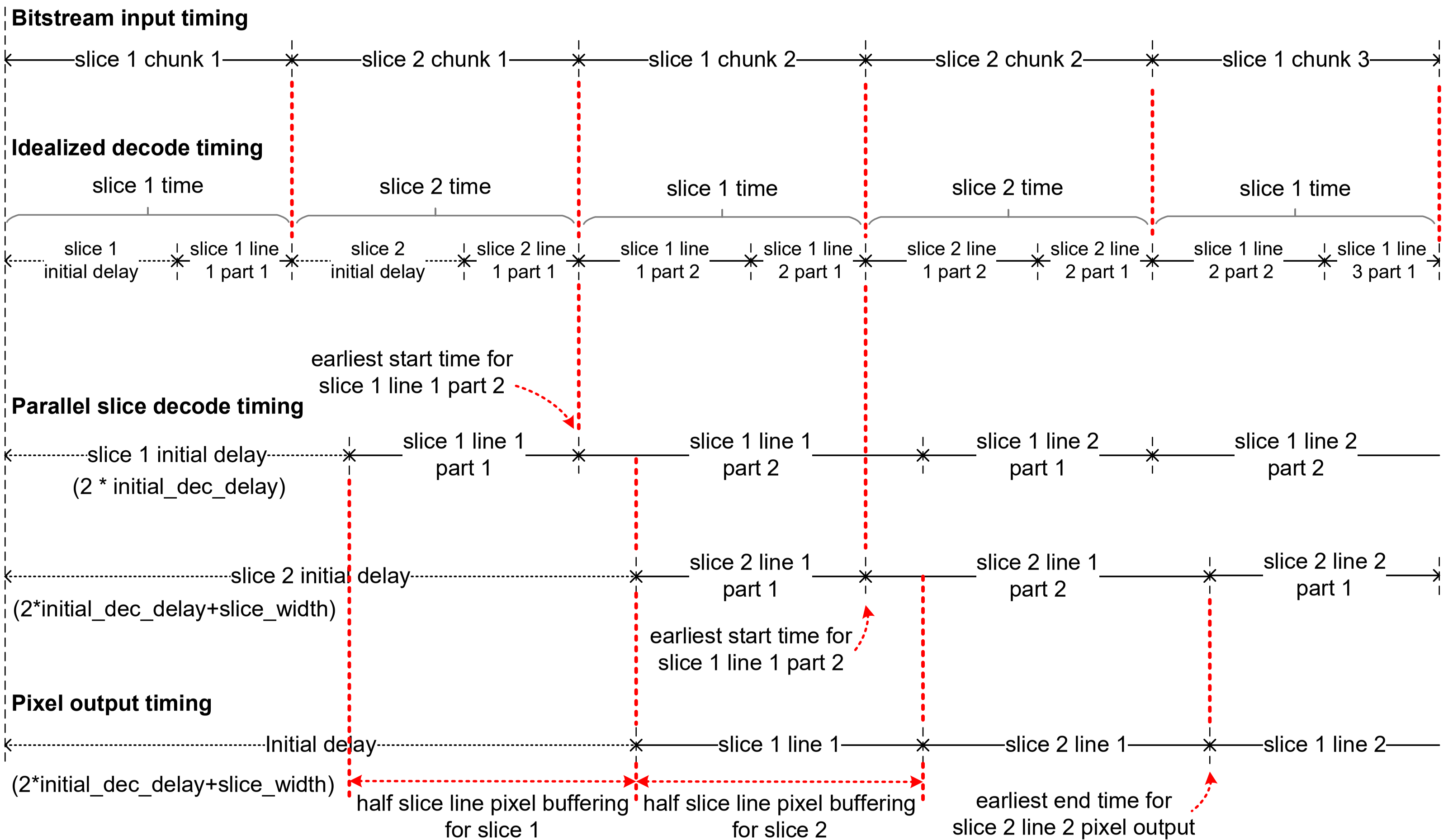


Figure 5.4: Parallel slice decode timing for two slices per line.

slices are decoded separately at the throughput of three pixels per cycle, which is half the speed of the bitstream input. The main constraint to ensure that the rate buffer does not underflow is that any pixel in the parallel decode timing is decoded no earlier than they would be in the idealized timing. For slice 1, line 1 part 2 starts at the same time as it does in the idealized timing and takes twice the time to finish. Slice 1 line 1 part 1 decoding ends at the start time of slice 1 line 1 part 2; therefore its starting time (i.e., the slice 1 initial delay) in terms of pixel time can be derived as:

$$\begin{aligned} slice_1_initial_delay &= 2 * slice_width - 2 * slice_1_line_1_part_1 \\ &= 2 * slice_width - 2 * (slice_width - initial_dec_delay) \\ &= 2 * initial_dec_delay \end{aligned} \tag{5.4}$$

Similarly, slice 2 line 1 part 2 starts at the same time as it would in the idealized timing, and the starting time of slice 2 line 1 part 1 (i.e., slice 2 initial delay) can be derived as:

$$\begin{aligned} slice_2_initial_delay &= 3 * slice_width - 2 * slice_2_line_1_part_1 \\ &= 3 * slice_width - 2 * (slice_width - initial_dec_delay) \\ &= slice_width + 2 * initial_dec_delay \end{aligned} \tag{5.5}$$

The same results hold if the initial decoding delay is larger than slice width. The pixel output timing makes sure the decoded pixels are in raster-scan order and match the rate of the bitstream input, i.e., six pixels per cycle. The constraint is that pixels are sent to output no earlier than they are decoded. To satisfy this constraint, the end time of slice 2 line 1 is no earlier than its end time in the parallel slice decode timing. Therefore, the initial delay for the pixel output timing is the same as the slice 2 initial delay in the parallel slice decode timing.

Compared to the idealized decode timing, the amount of extra buffering for the parallel slice decode timing for the rate buffer is equal to part 1 of first line in both slices amount of bitstream, i.e.,

$$\begin{aligned} rate_buffer_extra &= (slice_1_line_1_part_1 + slice_2_line_1_part_1) * bits_per_pixel \\ &= 2 * (slice_width - initial_dec_delay) * bits_per_pixel \end{aligned} \tag{5.6}$$

A conservative upper bound is one picture line worth of bitstream. For decoders that support specific slice width and initial delay, the accurate amount of extra buffering can be used. As the pixel output timing diagram shows, the first line of slice 1 is not sent to output until half a slice line has been decoded, which needs to be temporarily stored in the pixel buffer. The same amount of buffering is required for slice 2. In total, the pixel buffer needs to have a capacity of at least half a picture line of pixels.

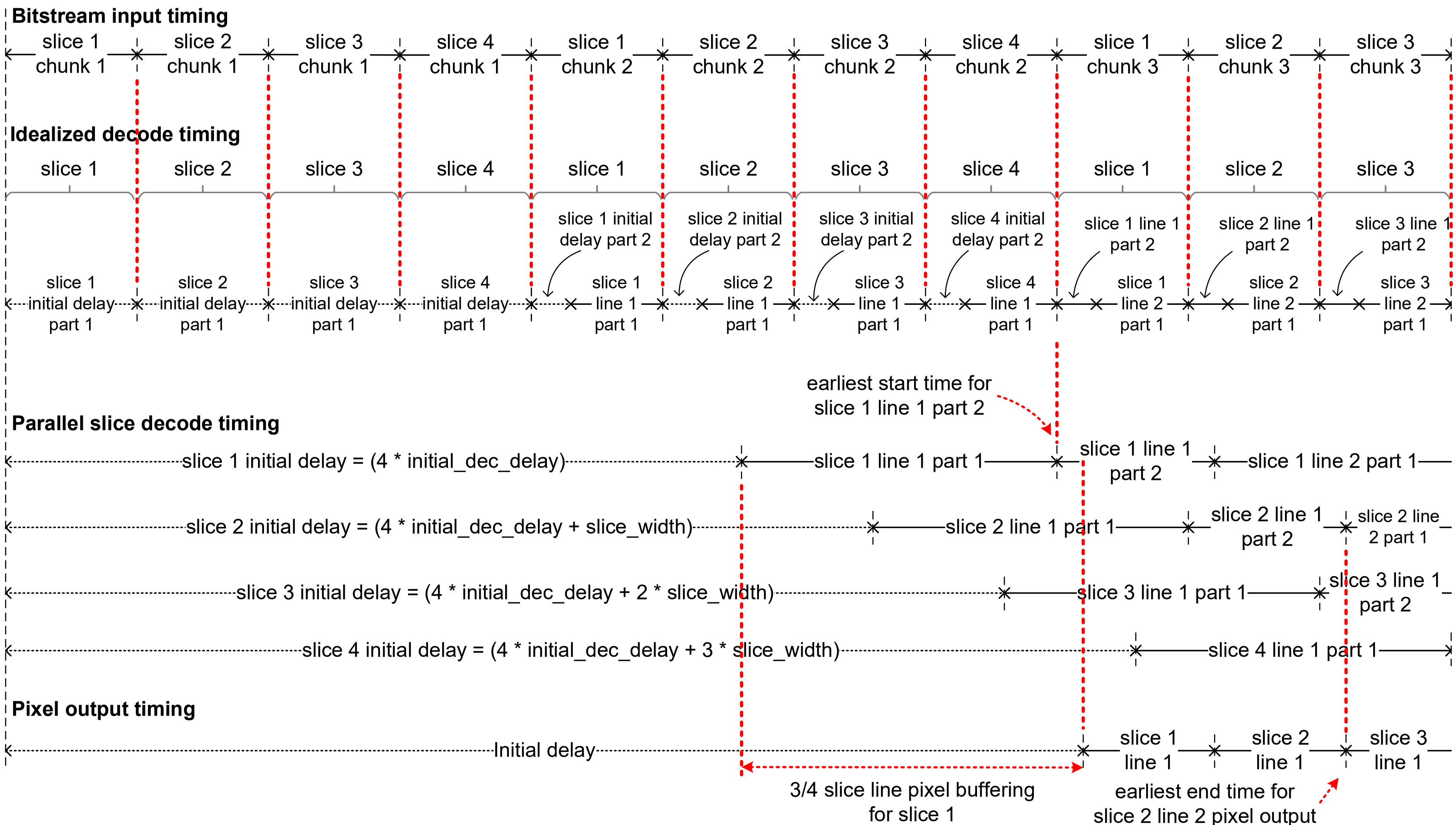


Figure 5.5: Parallel slice decode timing for four slices per line.

Figure 5.5 shows the slice timing of parallelly decoding four slices per line. The bitstream input rate is 12 pixels per cycle. In this analysis the initial decoding delay ($initial_dec_delay$) is assumed to be larger than slice width; therefore the initial delay period for each slice contains two parts. In the idealized decode timing, each slice line is divided into two parts. Each slice is allocated a quarter of the total time and it is only processed during the time that its bitstream enters the decoder. In the parallel slice decode timing, all four slices are decoded in parallel at the rate of three pixels per cycle, which is a quarter that of the idealized decode timing. The main timing constraint for the parallel decode timing is all pixels should be decoded no earlier than they would be in the idealized timing, so that rate buffer does not underflow. To satisfy this constraint, the second part of the first line of each slice should start at the same time or later than it would be in the idealized timing. The initial delay of each slice can be calculated accordingly. For example, the initial delay for slice 1 is computed as:

$$\begin{aligned} slice_1_initial_delay &= 8 * slice_width - 4 * slice_1_line_1_part_1 \\ &= 8 * slice_width - 4 * (2 * slice_width - initial_dec_delay) \\ &= 4 * initial_dec_delay \end{aligned} \tag{5.7}$$

Slices 2, 3, and 4 are delayed for another one, two, and three slice width pixel times compared to slice 1. The pixel output of all slices are in raster-scan order at the rate of 12 pixels per cycle, which matches the bitstream input rate. The pixels of each slice are not sent to output until the last quarter of each slice line time.

Compared to the idealized decode scheme, the parallel slice decode scheme requires additional buffering for the rate buffer. The minimum amount of additional buffering is the first part of the slice line for each slice; therefore, the total extra required rate buffer size is:

$$\begin{aligned} rate_buffer_extra &= 4 * slice_1_line_1_part_1 * bits_per_pixel \\ &= 4 * (2 * slice_width - initial_dec_delay) * bits_per_pixel \end{aligned} \tag{5.8}$$

A conservative upper bound is one picture line worth of bitstream. In addition, three quarters slice line of pixels should be buffered for each slice line, which totals to three quarters picture line amount of buffering for the pixel buffer.

5.4 Parallel-Interleaved Decoder

Both the slice-interleaved decoder and parallel slice decoder architectures are capable of decoding multiple slices per line; both architectures focus on either hardware cost (i.e., chip area) or

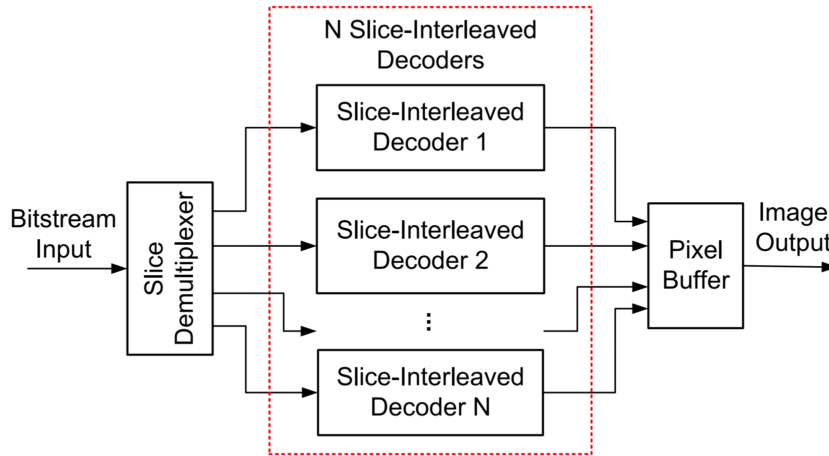


Figure 5.6: Parallel-interleaved decoder architecture.

performance (i.e., decoding throughput). The slice-interleaved decoder minimizes the additional hardware cost compared to the slice decoder at the cost of throughput overhead, whereas the parallel slice decoder maximizes throughput with linearly increased hardware cost. The *parallel-interleaved decoder* architecture takes advantage of both slice-interleaved decoder architecture and parallel slice decoder architecture and explores the trade off between throughput and hardware cost while supporting multiple slices per line.

5.4.1 Architecture

Figure 5.6 shows the high-level block diagram of the parallel-interleaved decoder which consists of N slice-interleaved decoders, and has a throughput up to $3 * N$ pixels per cycle when all slice-interleaved decoders are active. The maximum number of slices per line it supports is:

$$max_slices_per_line = \sum_{i=1}^{N} M(i) \tag{5.9}$$

where:

- $M(i)$ is the maximum slices per line supported by the ith slice-interleaved decoder.

When the parallel-interleaved decoder decodes pictures of N slices per line, every slice-interleaved decoder processes one column of slice. In this scenario, the parallel-interleaved decoder works as a parallel slice decoder. If the number of slices per line is greater than N, every slice-interleaved decoder processes a certain number of columns of slices.

The rate of the bitstream input matches the processing speed, i.e., the budgeted bitstream for N groups enters the slice demultiplexer each cycle. The slice demultiplexer splits the bitstream

of different columns of slices to the corresponding slice-interleaved decoder. In the decoder output side, the pixel buffer temporarily stores the decoded pixels of every slice-interleaved decoder and produces outputs in raster-scan order at a rate of $3 * N$ pixels per cycle.

5.4.2 Slice Decode Timing and Extra Buffering

Figure 5.7 shows the slice decode timing of a parallel-interleaved decoder that supports four slices per line. Slices 1 and 2 are processed using one slice-interleaved decoder; slices 3 and 4 are processed by another slice-interleaved decoder. Then one of slices 1 and 2 and one of slices 3 and 4 are processed in parallel. The bitstream input timing and idealized decode timing are the same as that of parallel-4-slice decode timing, except that the throughput is six pixels per cycle (i.e., two groups per cycle). The first two slices (slices 1 and 2) and the second two slices (slices 3 and 4) are parallelly decoded using slice-interleaved decoding scheme at the throughput of three pixels per cycle. Similar to the slice-interleaved and parallel slice decoder architectures, all pixels are decoded no earlier than they would be in the idealized decode timing, so that the rate buffer never underflows. The decoded pixels are sent to output in raster-scan order at a rate of six pixels per cycle, which matches the bitstream input rate after the initial delay period. The amount of initial delay is determined by the constraint that once the decoder starts sending pixels to output, the pixel stream should be continuous until the end of decoding. As shown in the pixel output timing, the earliest end time for slice 2 line 1 pixel output should be no earlier than the end time of slice 3 line 1. Therefore, the initial delay duration in terms of bitstream input pixel time is:

$$initial_delay = 6 * slice_width + 2 * initial_dec_delay \tag{5.10}$$

Compared to the idealized decode scheme, the parallel-interleaved decoder requires extra buffering in the rate buffer due to the additional delays. The worst case of extra buffering in terms of pixel times for the first two slices can be calculated by comparing the decoded slice lines until the end of slice 2 chunk 3.

$$\begin{aligned} extra_delay &= slice_1_line_2_part_1 + slice_2_line_1_part_2 + slice_2_line_2_part_1 \\ &= slice_1_line_2_part_1 + slice_2_line_1 \\ &= 3 * slice_width - initial_dec_delay \end{aligned} \tag{5.11}$$

Similar analysis can be performed on slices 3 and 4, which requires the same amount of extra delay.

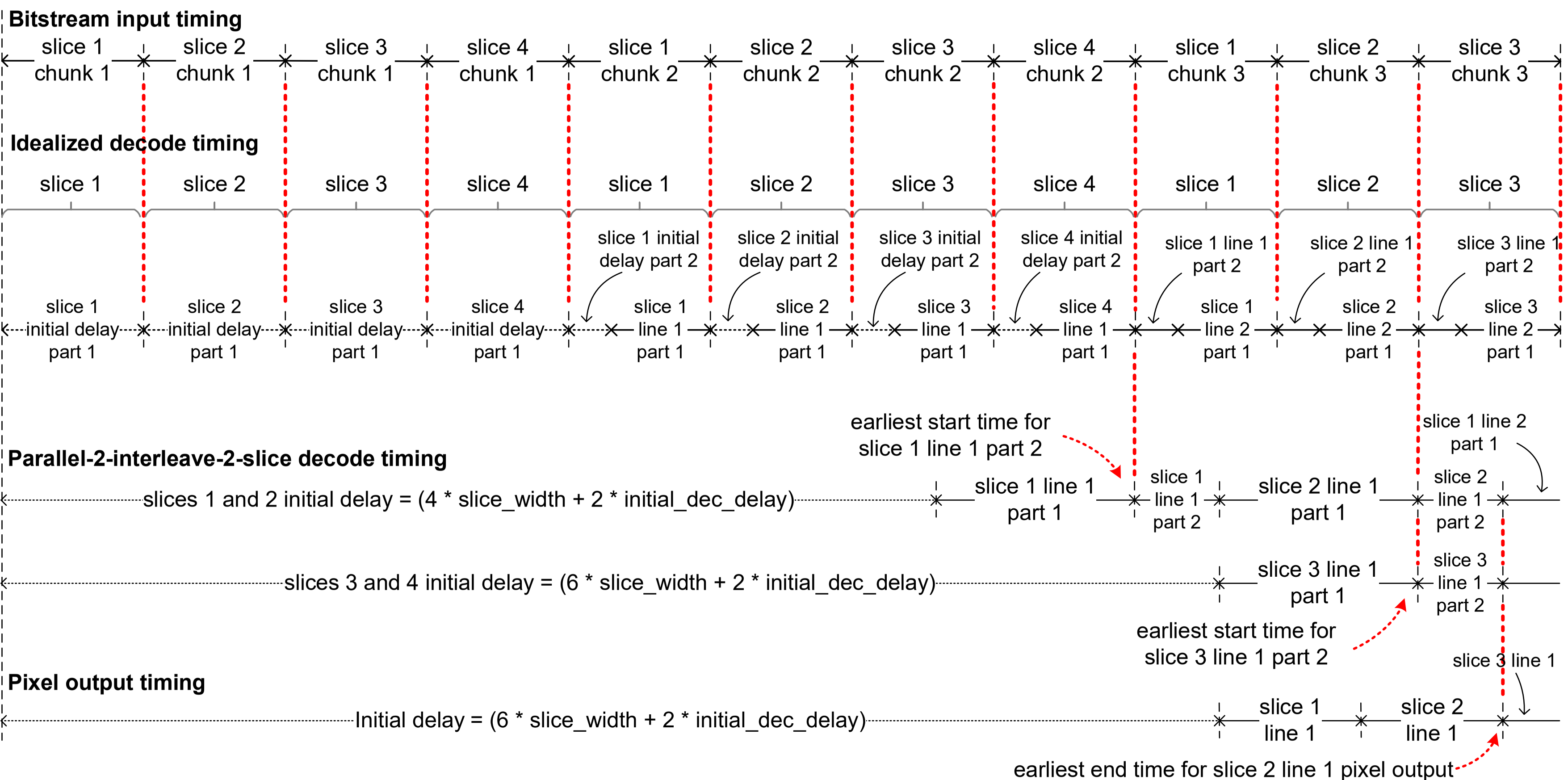


Figure 5.7: Parallel-interleaved slice decode timing for four slices per line.

Therefore, the total amount of extra buffering for the rate buffer is:

$$rate_buffer_extra = 2 * (3 * slice_width - initial_dec_delay) * bits_per_pixel \tag{5.12}$$

A non-tight upper bound is one picture line pixel time worth of bitstream, which is the same as the upper bound for slice-interleaved and parallel slice decoder architectures.

The pixel buffer is required to store one slice line of pixels for the first two slices and another slice line of pixels for the second two slices, which results in a total of half a picture line of pixels.

5.5 DSC Decoder Implementations in 28 nm CMOS

To evaluate the proposed decoder architectures, six DSC decoders are implemented: 1) a slice decoder; 2) two slice-interleaved decoders called *interleave-2-slice decoder* and *interleave-4-slice decoder*, which support up to two and four slices per line, respectively; 3) two parallel slice decoders called *parallel-2-slice decoder* and *parallel-4-slice decoder*, which support two and four slices per line, respectively; 4) a parallel-interleaved decoder called *par-2-inter-2-slice decoder* which processes two slices in parallel and supports two and four slices per line. All six decoders fully comply with DSC version 1.2a and are backward compatible with versions 1.2 and 1.1; they support uncompressed video component bit depth of 8 bits per component (bpc), compressed bit rate of 8 bits per pixel (bpp), and video resolutions up to 1080p (1920 $\times$ 1080).

5.5.1 Design Flow and Implementations

The decoders are implemented using a standard-cell-based automatic placement and routing (PnR) design flow. The architectures are modelled in register-transfer-level (RTL) using Verilog HDL. Each design is synthesized into a gate-level netlist using a 28 nm standard cell library by Synopsys Design Compiler, and placed and routed using Cadence Innovus. The C model which is provided by VESA is used as the golden reference for functional verification of the RTL and gate-level netlists after synthesis and placement and routing.

The same physical design flow is applied to all decoder implementations: 1) floorplanning, 2) power planning, 3) placement, 4) clock tree synthesis (CTS) and post-CTS optimization, 5) detailed routing and post-route optimization. I/O pins are evenly placed on the chip edges. The top two layers of metals are used for distributing power across the chip. Setup and hold timing

are optimized after CTS and detailed routing. The optimizations for clock frequency and core area are achieved through multiple iterations. First, a core utilization of 0.7 and aspect ratio of 1 are used for a initial floorplanning, which returns a core dimension. The first round of clock frequency optimization is done by fixing the core area and reducing clock period by 0.1 ns per PnR iteration until timing enclosure is no longer achievable. Core Area is optimized by fixing the clock period to the smallest value that achieved timing enclosure, and reducing each dimension of the core by five microns per iteration until timing no longer closes. A second round of clock frequency optimization is performed by fixing the core dimension to its smallest value and reducing clock period by 0.05 ns per iteration.

Figure 5.8 shows the final core layouts of the six decoders with their sizes scaled relative to slice decoder. The slice decoder is the smallest design and occupies only 0.39 mm^2, which is 2.6 times smaller than the parallel-4-slice decoder. Figure 5.9(a) compares the critical path delay after synthesis with a target clock period of 1.0 ns. The slice decoder results in the smallest delay of 1.11 ns. The delay of interleave-2-slice and interleave-4-slice decoders are 8% and 16% larger than the slice decoder, respectively. The delay of the parallel slice decoders and par-2-inter-2-slice decoder are 2%–5% larger than slice decoder. Figure 5.9(b) shows 38%–60% delay increase from synthesis to placement and routing. The core utilization in placement and routing is between 77.24% for slice decoder and 81.40% for interleave-4-slice decoder, as shown in Figure 5.9(c). The logic and memory area of all six decoders are plotted in Figure 5.9(d). The memory modules occupy 62%–73% of the total core area. The area can be further reduced by using SRAM cells rather than flip-flops for line buffer, rate buffer, and pixel buffer.

Figure 5.10 shows the layout of the slice decoder with all memories and logic blocks highlighted in different colors. Figure 5.11 shows the logic area breakdown of the slice decoder. Prediction, inverse quantization, and reconstruction is the most complicated block and occupies one third of the total logic area. The substream demultiplexer is the second largest block. The ICH of the decoder uses 13.5% of the logic area and is much smaller than its counterpart in the encoder since it does not need to find the best ICH entries and make coding mode decision. The logic area utilization breakdown of other decoders are similar to that of the slice decoder.

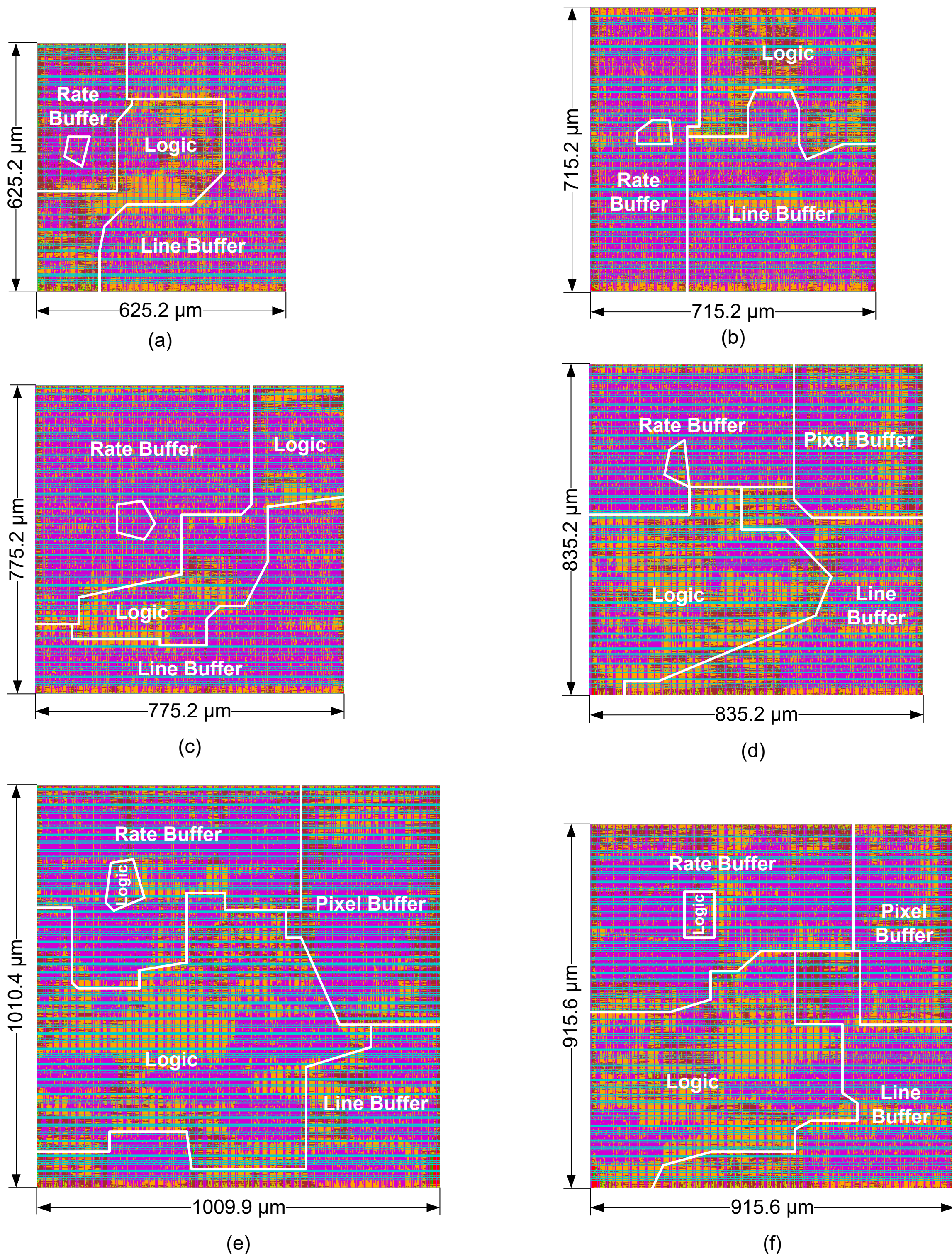


Figure 5.8: Core layouts of the six implemented DSC decoders. (a) Slice decoder. (b) Interleave-2-slice decoder. (c) Interleave-4-slice decoder. (d) Parallel-2-slice decoder. (e) Parallel-4-slice decoder. (f) Par-2-inter-2-slice decoder. All layout sizes are to scale.

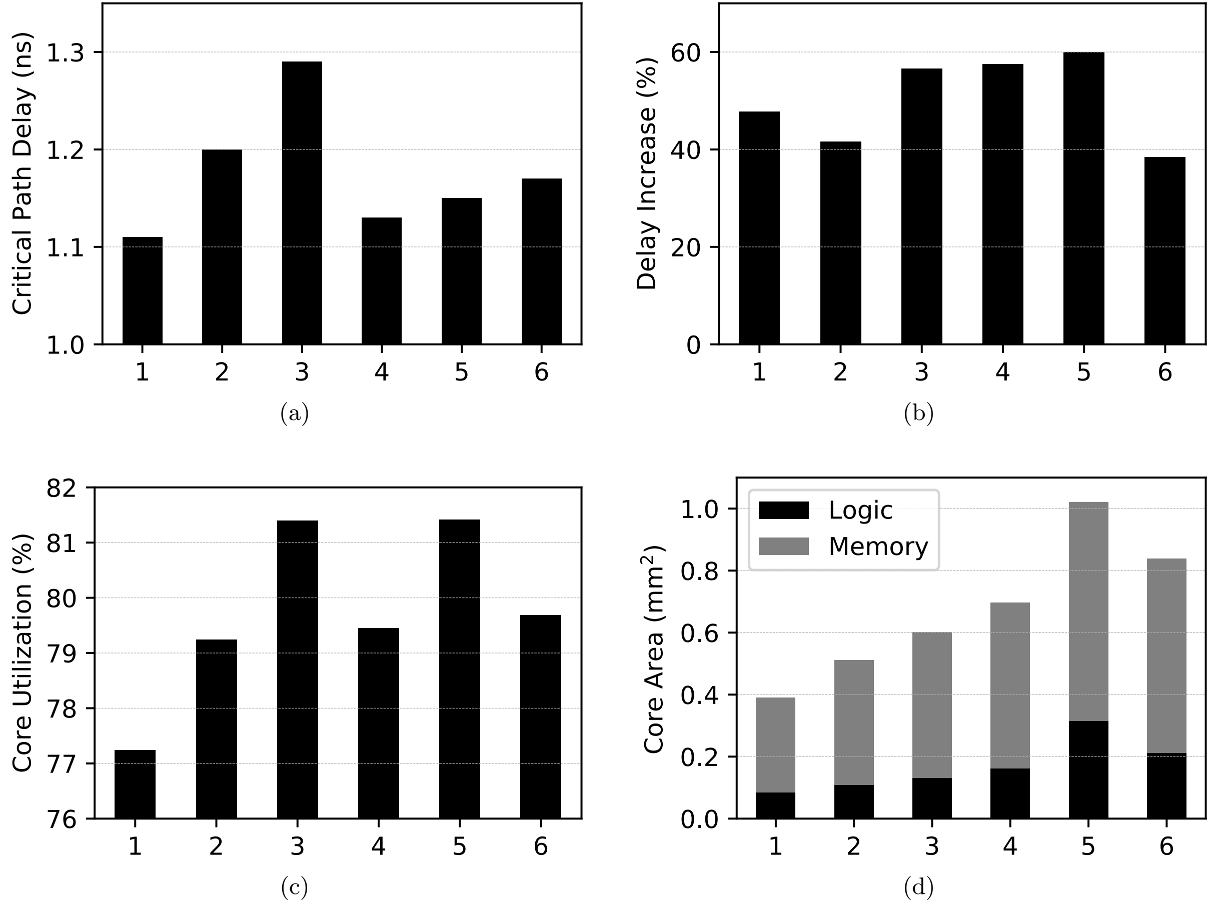


Figure 5.9: Comparison of the six implemented DSC decoders. (a) Post-synthesis critical path delay. (b) Increase from post-synthesis delay to final post-PnR delay. (c) Core utilization. (d) Core area. The decoders 1 to 6 are: slice decoder, interleave-2-slice decoder, interleave-4-slice decoder, parallel-2-slice decoder, parallel-4-slice decoder, and par-2-inter-2-slice decoder.

5.5.2 Results

Table 5.1 compares gate count, maximum frequency, throughput, power dissipation, and energy efficiency of the proposed decoders. The slice decoder contains 169.6 K logic gates in terms of minimum-sized NAND2 equivalent, which is 3.7 times smaller than the parallel-4-slice decoder. The interleave-2-slice and interleave-4-slice decoders consist of only 1.3 and 1.6 times the logic gate count of slice decoder. On the other hand, the parallel-2-slice and parallel-4-slice decoders contain 1.9 and 3.7 times the logic gate count of slice decoder, which almost increases linearly as the number of parallel processed slices. The slice-interleaved decoders occupy 33% and 58% fewer logic gates than

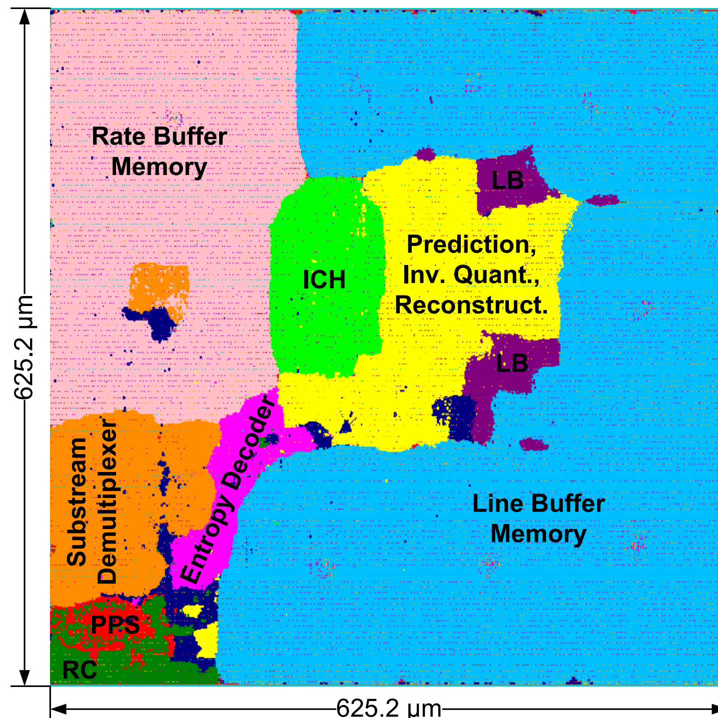


Figure 5.10: Annotated layout of the slice decoder.

the parallel slice decoders to support two and four slices per line, respectively, which demonstrates the superior area efficiency of the slice-interleaved architecture.

The decoders achieve 495.0–617.3 MHz maximum clock frequency at the nominal voltage of 1.0 V. Compared to the slice decoder, the maximum clock frequency reduces by 3.5% and 18.8% for the slice-interleaved architecture to support two and four slices per line, respectively. On the other hand, the parallel slice decoders achieve only 7.9% and 10.9% lower clock frequency than the slice decoder. In the proposed decoders, the maximum frequencies in the placement and routing are determined by hold time requirements rather than setup time; therefore, although the parallel-2-slice decoder has a longer critical path delay than the slice decoder, it achieves the highest clock frequency among all the decoders.

The maximum throughput is derived from the maximum clock frequency. In the slice decoder and slice-interleaved decoders, the throughput is three pixels per cycle for simple 4:4:4 mode and six pixels per cycle for native 4:2:2 and 4:2:0 modes, whereas parallel-2-slice and parallel-4-slice decoders achieve twice and four times the throughput per cycle. The par-2-inter-2-slice decoder achieves 6 and 12 pixels per cycle for simple 4:4:4 and native 4:2:2 and 4:2:0 modes, respectively. Compared to the slice decoder, the interleave-2-slice and interleave-4-slice decoders achieve 3.6% and

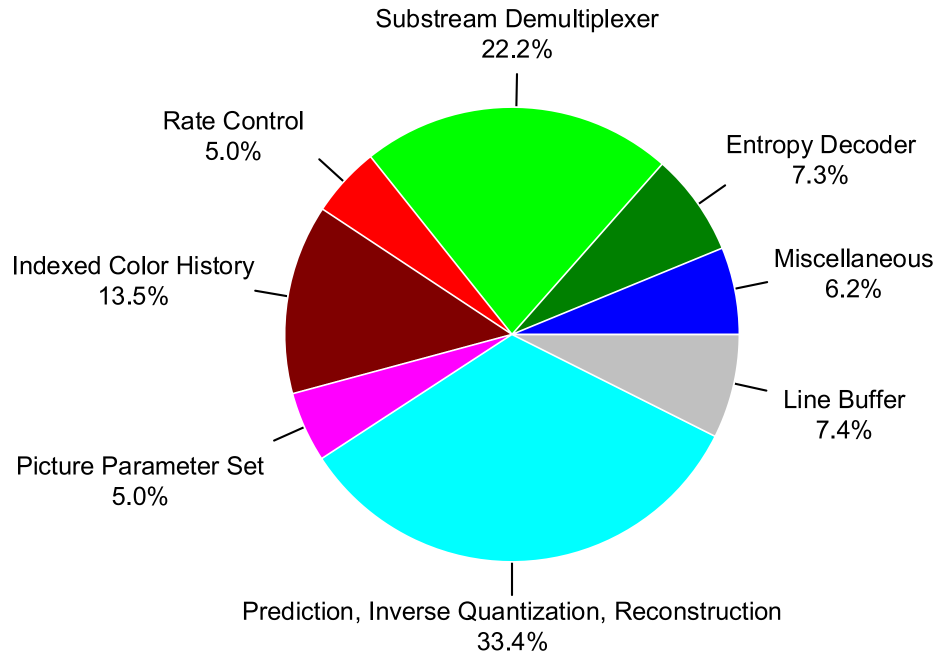


Figure 5.11: Logic area breakdown of the slice decoder. The logic area used for constructing previous line pixels is classified into prediction, inverse quantization, and reconstruction. The area of substream demultiplexer includes the area of funnel shifters, and the logic area for rate buffer. YCoCg-R to RGB conversion area is classified as miscellaneous.

3.8% lower throughput, whereas the parallel-2-slice and parallel-4-slice decoders achieve 1.84 and 3.56 times higher throughput. Overall, the throughput of 1.49–6.52 gigapixels per second (Gpixels/s) and 2.97–13.04 Gpixels/s is demonstrated for simple 4:4:4 and native 4:2:2/4:2:0 pixel modes, respectively.

Power dissipation is estimated by Synopsys PrimeTime. For each design in each pixel mode, a value change dump (VCD) dump file is created by simulating the netlist exported from placement and routing result with standard delay format (SDF) back annotation at the maximum clock frequency at 1.0 V. The VCD file records the the time of value changes in variables, which enables more accurate estimation of the dynamic power. PrimeTime uses the netlist, VCD file, and standard parasitic exchange format (SPEF) file to estimate the power. Four images (t_1280x768_Noise_128, t_Mandrill, t_1024x1024Cr, t_Desktop10) are used in the power estimation and the average power dissipation is reported in Table 5.1. For each decoder, the power dissipation in native 4:2:2 mode is 1.04–1.10 times higher than native 4:2:0 mode and 1.13–1.36 times higher than simple 4:4:4 mode.

The energy per pixel results are reported in Table 5.1. The parallel-4-slice decoder is most

energy efficient design and achieves 22.9 pJ, 24.4 pJ, and 43.2 pJ per 4:2:0, 4:2:2, and 4:4:4 pixel, respectively. In all decoders, the average energy required to decode a 4:2:2 pixel is 4%–10% more than that for a 4:2:0 pixel. On the other hand, although the power dissipation of simple 4:4:4 mode is lower than native 4:2:0 mode, a 4:4:4 pixel consumes 57%–89% more energy than a 4:2:0 pixel because the decoders achieve twice the throughput in native 4:2:0 mode than simple 4:4:4 mode. The energy efficiency decreases as the number of slices per line increases in the slice-interleaved decoders. The interleave-2-slice and interleave-4-slice dissipate 13%–18% and 23%–32% more energy per pixel than the slice decoder, respectively. On the other hand, the parallel slice decoders are more energy efficient as the number of parallelly processed slices increases. For example, parallel-4-slice decoder dissipates 80%, 79%, and 93% energy per 4:2:0, 4:2:2, and 4:4:4 pixel than the parallel-2-slice decoder. The energy efficiency for the par-2-inter-2-slice decoder is between the interleave-2-slice decoder and parallel-2-slice decoder.

5.5.3 Scalability

The proposed DSC decoder architectures are fully scalable for various component bit depths, video resolution, and number of slices per line. To add the support for larger component bit depths, i.e, 10, 12, 14, and 16 bpc, the datapath width needs to be increased accordingly. Since the line buffer stores one line of pixels, its size increases linearly with the increase of component bit depth. For 12 bpc and up, the mux word size is increased to 64 bits; therefore, the bit-width of rate buffer entries needs to be increased from 192 bits to 256 bits to hold four mux words, although the total size remains the same. In addition, the size of funnel shifters increases with the increased component bit depth.

To increase the maximum supported video resolution, only the size of rate buffer, line buffer, and pixel buffer need to be increased. Figure 5.12 shows the the memory size requirements for 1080p and 4K UHD video resolutions in all six decoders. From 1080p to 4K UHD, the size of line buffer and pixel buffer increases by twice, whereas the size of rate buffer increases by 1.4–1.5 times; the total memory size increases by 1.6–1.9 times.

The slice-interleaved, parallel slice, and parallel-interleaved decoders are fully scalable to support larger number of slices per line. However, increasing the number of slices per line results in larger rate buffer. The size of rate buffer increases by 2.3 and 3.3 times from one to two and

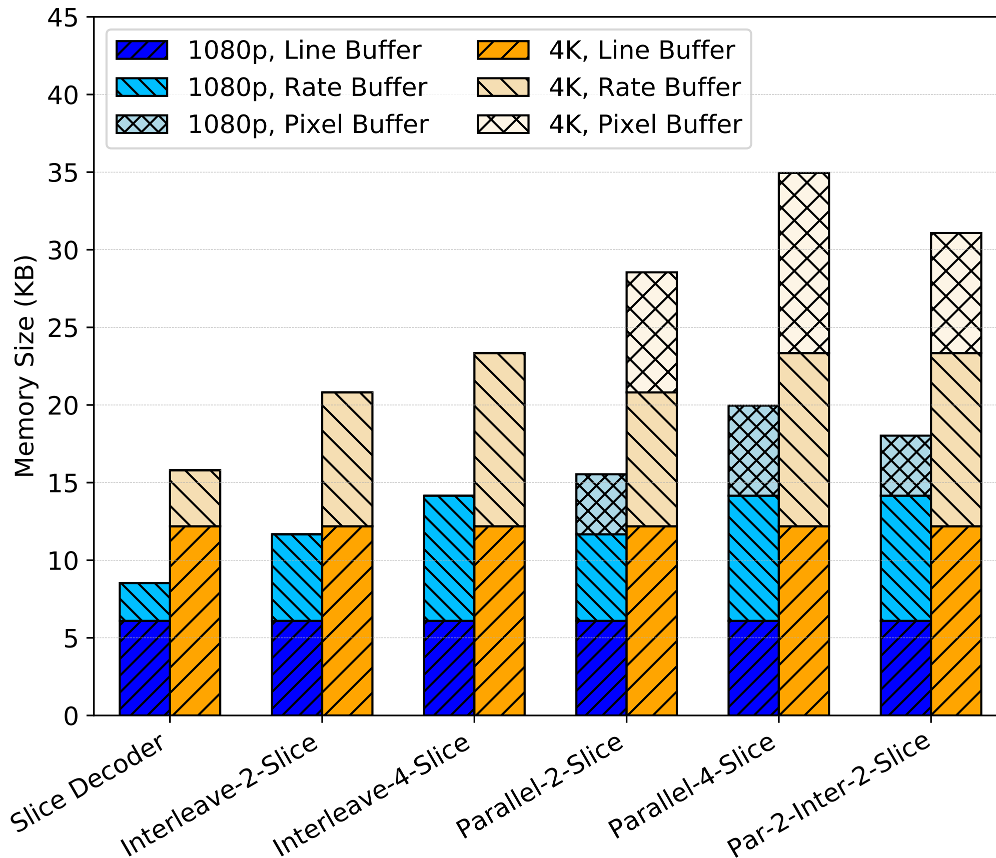


Figure 5.12: Memory size requirements in the proposed decoders for 1080p and 4K video resolution.

four slices per line to support 1080p. The pixel buffer also increases with the increased number of parallelly-processed slices. Therefore, it is desirable to limit the number of slices per line.

5.5.4 Comparison With Related Work

With technology scaling taken into consideration [42–45], the proposed slice decoder in this work achieves five times higher frequency and throughput than a published slice decoder [32], which runs at 82 MHz and achieves a throughput of 246 Mpixels/s after synthesis in 65 nm CMOS at 1.2 V.

5.6 Summary

This chapter presents four DSC decoder architectures and the design of six decoders based on the proposed architectures. The slice decoder, which supports one slice per line decoding, is the

smallest design consisting of 169.6 K logic gates. The parallel slice decoders decode multiple columns of slices in parallel using multiple slice decoders, which results in linear increase of throughput and area. The slice-interleaved decoders support multiple slices per line with only replicated registers for each column of slices, which results in 29% and 55% additional logic area than the slice decoder with only 3.5% and 19% lower throughput, respectively. The parallel-interleaved decoder architecture combines both parallel slice and slice-interleaved architectures and explores the trade off between throughput and area while supporting multiple slices per line.

Table 5.1: Comparison of the Proposed DSC Decoders

Decoder	Gate Count*	Max. Freq.	Throughput (Gpixels/s)		Power (mW)			Energy / Pixel (pJ/pixel)		
Design	(K)	(MHz)	4:2:0/4:2:2	4:4:4	4:2:0	4:2:2	4:4:4	4:2:0	4:2:2	4:4:4
Slice Decoder	169.6	609.8	3.66	1.83	106.6	111.0	87.3	29.1	30.3	47.7
Interleave-2-Slice	218.6	588.2	3.53	1.76	121.2	126.3	94.9	34.3	35.8	53.8
Interleave-4-Slice	262.5	495.0	2.97	1.49	107.5	118.5	87.0	36.2	39.9	58.6
Parallel-2-Slice	324.3	561.8	6.74	3.37	193.3	208.7	155.9	28.7	31.0	46.3
Parallel-4-Slice	627.0	543.5	13.04	6.52	298.4	318.4	281.7	22.9	24.4	43.2
Par-2-Inter-2-Slice	421.4	617.3	7.41	3.70	244.6	262.6	196.1	33.0	35.5	52.9

* Minimum-sized NAND2 equivalent.

Chapter 6

Display Stream Compression Decoders for Fine-Grained Many-Core Processor Arrays

This chapter presents the many-core software design of three Display Stream Compression (DSC) decoders on fine-grained many-core processor arrays. The proposed decoders exploit task-level, component-level, and slice-level parallelisms to achieve real-time decoding performance in 1080p video resolution.

6.1 Introduction

The application-specific integrated circuit (ASIC) implementations of DSC codecs in Chapters 4 and 5 have demonstrated low hardware cost, high performance, and high energy efficiency. On the other hand, software video codecs allow for more flexibility and scalability. Generally, software computation platforms can be classified into three categories: 1) general purpose processors, such as central processing unit (CPU), 2) graphics processing unit (GPU), and 3) fine-grained multiple instruction multiple data (MIMD) many-core processor array. The single-core and multi-core general purpose processors allow for coarse-grained parallelism, such as thread-level. GPUs [46–52] consist of massive number of parallel cores with single instruction multiple data (SIMD) architecture which enables data-level parallelism by processing different data using the exact

same instruction. The fine-grained MIMD many-core computation platform allows for task-level parallelism and processes different data in parallel with different instructions. A number of such processor arrays have been developed, such as the 64-core TILE64 [53], 72-core TILE-Gx72 [54], 80-core TeraFLOPS [55], 336-core Am2045 [56], 1024-core Epiphany-V [57], and the asynchronous array of simple processors (AsAP), which include a 36-core AsAP [58,59], a 167-core AsAP2 [60,61], and a 1000-core KiloCore [62],

Fine-grained MIMD many-core computation platforms have demonstrated better energy efficiency than general purpose processors and GPUs in many applications, such as sparse matrix-vector multiplication [63,64], matrix inversion [65], machine learning [66,67], Canonical Huffman decoding [68], H.264/AVC video coding [69–71], Advanced Encryption Standard (AES) cipher engines [72], pattern matching [73], and sorting [74].

6.2 The Targeted Many-Core Processor Arrays

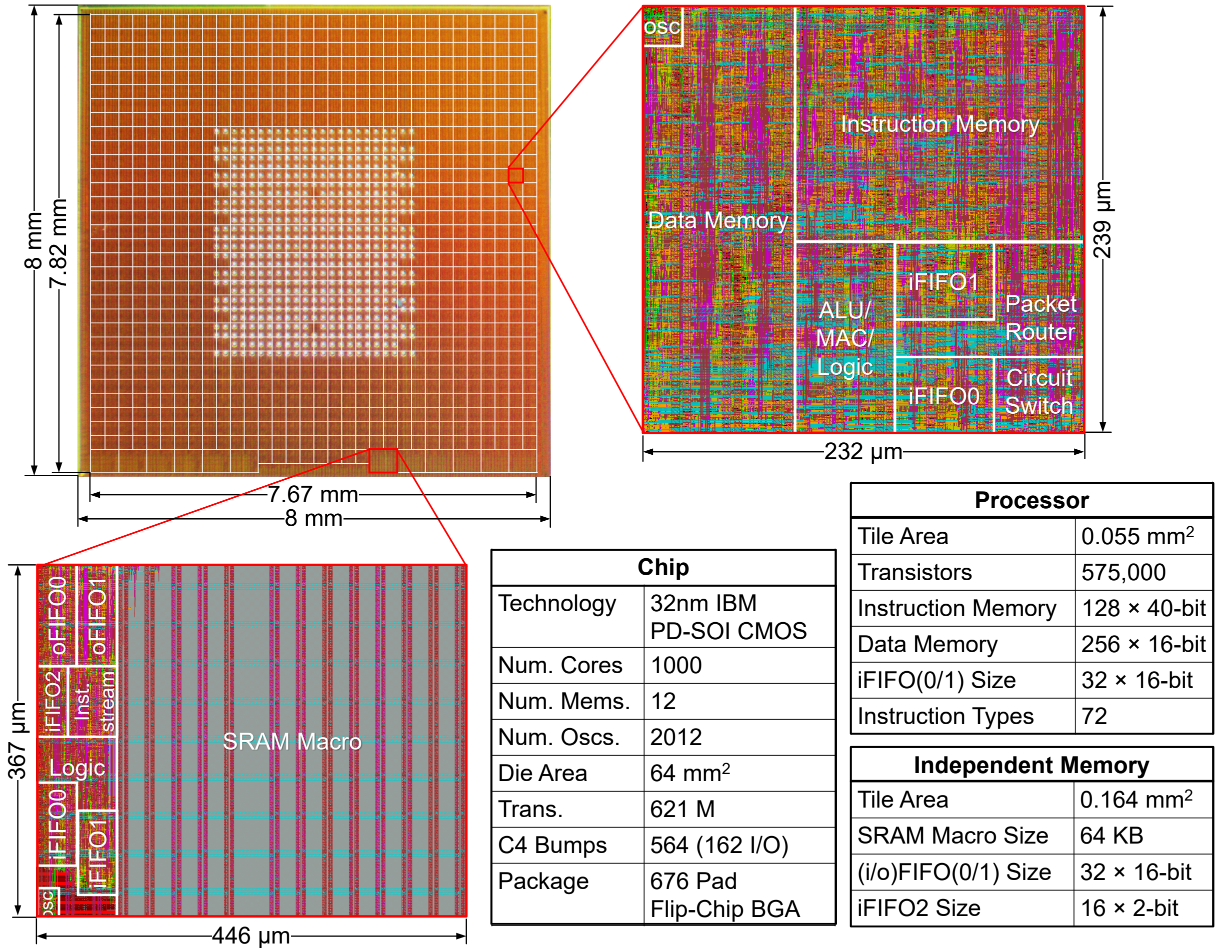
6.2.1 The KiloCore Chip

This work [76] targets fine-grained multiple instruction multiple data (MIMD) many-core arrays of independent, programmable processors. The KiloCore chip [62,77,78] is an example of the targeted many-core processor array and it is used as the computation platform in this work. It consists of 1,000 RISC-style processors and 12 64-KB independent memory modules connected via a 2-D mesh network that supports communication between adjacent and distant processors. Each processor contains 128×40-bit words of instruction memory, 256×16-bit words of data memory, and two 32×16-bit input dual-clock FIFOs [79]. Each processor and memory module occupies 0.055 mm^2 and 0.164 mm^2 in 32 nm CMOS technology, respectively. Figure 6.1 shows the die photo of the KiloCore array and the annotated layouts of a single processor tile and a single independent memory module.

6.2.2 Programming Methodology

The programming on KiloCore is achieved by four main steps [80].

1. Applications are partitioned into small tasks through *serial partitioning* and *parallel partitioning*. Serial partitioning converts the application into a sequence of tasks forming a computation



Chip	
Technology	32nm IBM PD-SOI CMOS
Num. Cores	1000
Num. Mems.	12
Num. Oscs.	2012
Die Area	64 mm²
Trans.	621 M
C4 Bumps	564 (162 I/O)
Package	676 Pad Flip-Chip BGA

Processor	
Tile Area	0.055 mm²
Transistors	575,000
Instruction Memory	128 × 40-bit
Data Memory	256 × 16-bit
iFIFO(0/1) Size	32 × 16-bit
Instruction Types	72

Independent Memory	
Tile Area	0.164 mm²
SRAM Macro Size	64 KB
(i/o)FIFO(0/1) Size	32 × 16-bit
iFIFO2 Size	16 × 2-bit

Figure 6.1: Die photo of the KiloCore array, and annotated layout plots of a single processor tile and a single independent memory tile [75].

pipeline. Parallel partitioning increases throughput by replicating critical path tasks, an example of which is loop unrolling by creating multiple instances of the loop body so that they can be mapped to separate processor tiles later.

2. Program code is written for every unique task in C++ or assembly language. In order to fit into the processor array, the program need to satisfy the resource constrains. A cycle-accurate C++ simulator [81] is used to verify the functionality.

3. The inputs and outputs of tasks are connected together using a configuration file.

4. An automated mapping tool [82] maps the tasks to the many-core processor array. The mapping process performs optimizations including considerations such as inter-processor communication patterns to reduce routing congestion and energy usage.

6.3 Slice Decoder

The first many-core software DSC decoder is a *slice decoder*, which is capable of decoding pictures configured into one column of slices. Task-level parallelism is exploited to partition the decoder into small tasks, the goal of which is to achieve optimized throughput and area. All data transfers in the slice decoder are done as words of 16 bits, which is the native datapath size of the KiloCore chip.

The throughput optimization is done in two steps. First, the three or four pixel components are processed in parallel and mapped to different sets of processors. This step is applied to the funnel shifters, entropy decoders, ICH pixels, and the prediction, inverse quantization, and reconstruction. The processing of the fourth component is enabled only in native 4:2:2 mode; in other modes, the processors for the fourth component stay idle to reduce energy dissipation. Second, the latency of the feedback loops which are caused by inter-group data dependencies, i.e., the decoded information from the previous group is used in the current group, are reduced as much as possible. There are four major data dependencies that this design takes into account: 1) the funnel shifters generate syntax elements for entropy decoders, which decode the syntax element and send back the number of bits decoded for updating of funnel shifter; 2) entropy decoding of current group uses the ICH-mode decision, MPP selection, and size of quantized residuals from the previous group; 3) the rate control algorithm relies on the number of coded bits, which is decoded by the entropy decoder, to calculate quantization parameter (Qp), which is mapped to quantization level ($qlevel$) and used in the entropy decoder; 4) the prediction loop, i.e., all three predictors require the reconstructed pixel values of the previous group to predict values for the current group. The total latency of each loop limits the decoder throughput and the goal is to reduce it as much as possible. The kiloCore processor array does not use a conventional shared-memory architecture, so it is required to explicitly transmit data between processors. The inter-processor communication incurs latency overhead; therefore, it is essential to minimize the number of processors involved in the loops, while the amount of instruction and data memory used falls within the capacity of the processors. In addition, instructions are scheduled such that the execution time of the instructions in the loop is minimized.

Assuming the unused processors and memory modules on the chip can be used for other applications, the area of the decoder is considered as the total area of the processors and memory modules used in the design. Therefore, less number of processors and memory modules used in

the design yields smaller area. Reducing the processors used in the loops results in smaller design area. Moreover, since the critical path of the design is in the data-dependent feedback loops, more work can be fit into the processors in non-critical paths (which reduces the throughput of these processors) as long as they do not become the bottleneck of the design. For example, the YCoCg-R to RGB pixel format conversion task is combined with the task of merging reconstructed pixels in the ICH block and executed on one processor.

6.3.1 Rate Buffer and Substream Demultiplexer

Figure 6.2 depicts the dataflow of the rate buffer, substream demultiplexer, and entropy decoder. The budgeted amount of bits for a group is written to the rate buffer memory as 16-bit words. After the initial delay stage, the substream demultiplexer starts reading mux words from the rate buffer. This design supports 8 and 10 bpc and thus a mux word contains 48 bits, which is split into three 16-bit words. The substream demultiplexer always contains three (simple 4:4:4 and native 4:2:0 modes) or four (native 4:2:2 mode) mux words, so that it has enough mux words ready to be sent to the funnel shifters right after receiving the requests for mux words. Therefore, the time that funnel shifters spend waiting for mux words is minimized. Then the substream demultiplexer reads from the rate buffer the same number of mux words for the next group as it sent out for the current group.

The funnel shifter is partitioned into two stages and mapped to two processors in order to meet the instruction memory limit of a single KiloCore processor. The first stage maintains the fullness (i.e., the number of remaining bits) of the funnel shifter and requests mux words from the demultiplexer if it falls below the maximum size of syntax element. Once the mux word is received, it is shifted left based on the current fullness. Whenever a new mux word enters the funnel shifter, the shifter fullness is increased by 48, and after the funnel shifter receives the the actual size of the syntax element (se_size), the fullness of the funnel shifter is updated again. The second stage generates the syntax element and updates the funnel shifter storage. The maximum syntax element size is 36 bits for 8 bpc and 44 bits for 10 bpc, whereas the minimum funnel shifter buffer size is 83 and 91 bits for 8 bpc and 10 bpc, respectively. A syntax element is split into three 16-bit words (see se_h, se_m, and se_l in Figure 6.2) and the lowest 16-bit word contains unused bits, whereas a funnel shifter buffer is implemented with six 16-bit words. The funnel shifter is first updated

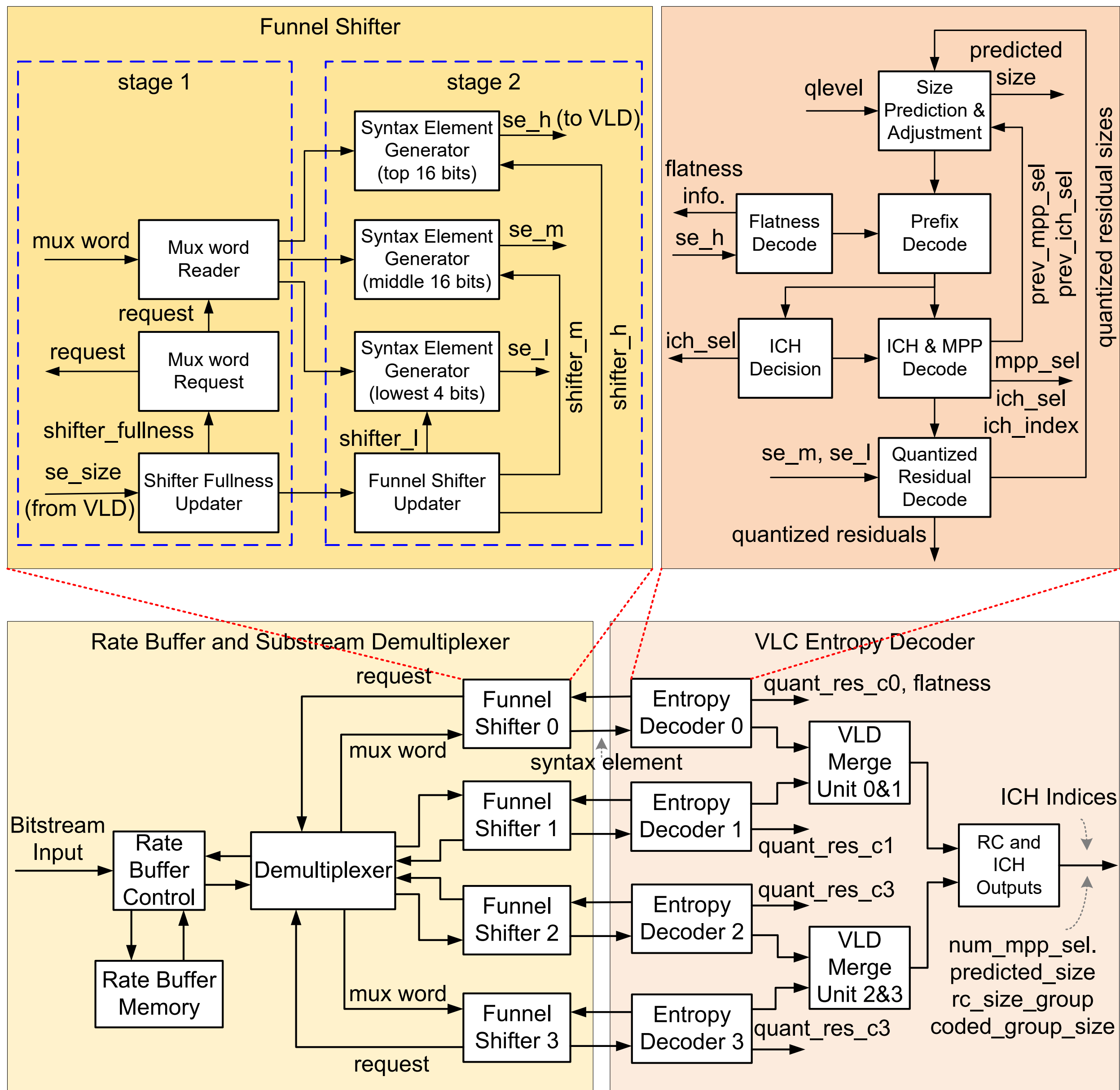


Figure 6.2: Dataflow diagram of the rate buffer, substream demultiplexer, and entropy decoder in the slice decoder. The details of a funnel shifter and entropy decoder of the first unit are elaborated.

with the shifted mux word by logic OR operations. Then, the top 16 bits of the syntax element are generated by a syntax element generator and sent to the entropy decoder, which is sufficient for the entropy decoder to calculate *se_size*. While waiting for *se_size* to be returned, the lower two words are generated and sent to the entropy decoder. After receiving *se_size*, the funnel shifter is updated again by shifting out the decoded bits.

Since the demultiplexer is mapped to one processor and one of its two input ports is

connected to the rate buffer control processor, it only has one spare input port which is used for the mux word requests. Three processors are used for merging the mux word requests from the four funnel shifters before sending the requests to the substream demultiplexer. In total, 13 processors and 1 memory module are used for rate buffer and substream demultiplexer.

6.3.2 Entropy Decoder

The right half of Figure 6.2 shows the diagram of the VLC entropy decoder. Four parallel entropy decoders process the syntax elements of different components in parallel. An entropy decoder consists of: residual size prediction and adjustment, prefix decode, ICH and MPP decode, and quantized residual decode and size calculation in predictive mode. The first component also contains flatness decode and ICH mode decision. The residual size prediction and adjustment depends on decoded information from the previous group: size of quantized residuals, MPP selection, and ICH selection. Therefore, the decoding of current group cannot start until the previous group is finished. To minimize the latency of this data dependency loop, the entropy decoder of each component is mapped into one processor. The only exception is the first component, where a processor is dedicated for flatness decoding since one processor does not have enough instruction memory. Right after the number of bits used to code the current group (*se_size*) is known, it is sent back to the funnel shifter.

This module is implemented with nine processors, which includes one processor used for distributing the input data, as well as three processors for merging the outputs of each components and generating the outputs for rate control (predicted size, number of MPP selection in the group (*num_mpp_sel*), total number of bits used (*coded_group_size*), the predicted number of bits to be used (*rc_size_group*)), and ICH indices.

6.3.3 Rate Control

Rate control dynamically generates a Qp for entropy decoding and prediction of the next group. Figure 6.3 shows that the rate control module is mapped to nine processors, including one processor for the buffer level tracker, three processors for linear transformation, one processor for long-term RC range selection, three processors for short-term RC Qp adjustment (short-term RC), and one processor for flatness Qp adjustment (flatness). A key consideration in partitioning this

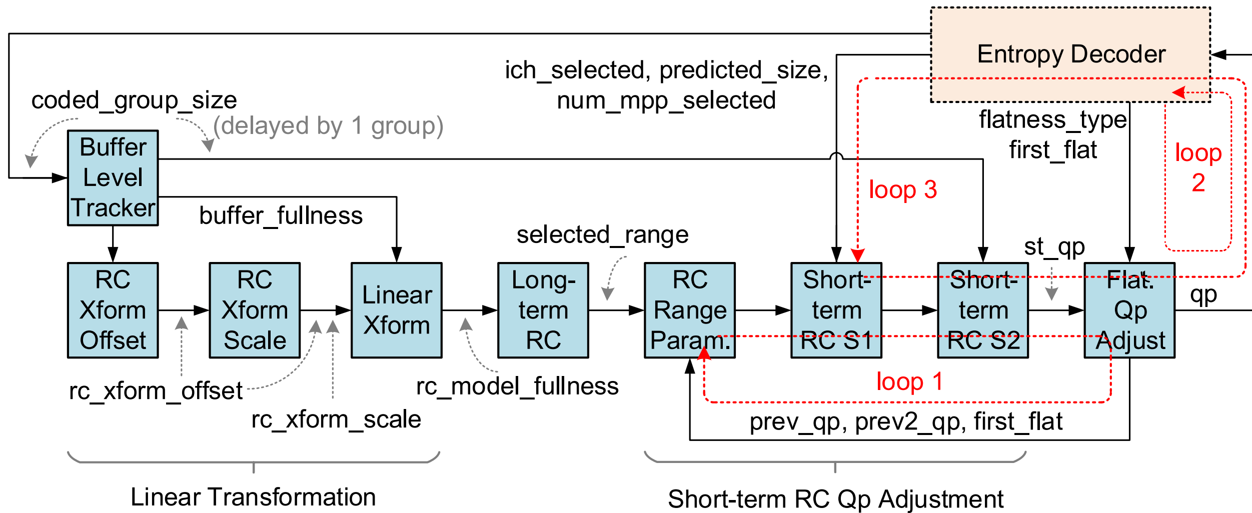


Figure 6.3: Dataflow diagram of the rate control module. The entropy decoder module in the right top corner is not part of the rate control but it is added to help demonstrate the signal connections and three loops (red) with the rate control module.

module is the data dependent feedback loops highlighted in Figure 6.3. The first feedback loop is between short-term RC and flatness. The quantization parameter of the current group, as well as the Qp for next group, which is calculated by short-term RC, are adjusted by the flatness module and used in the short-term RC of the next group. The second loop is between flatness and entropy decoder, where the flatness information decoded by the entropy decoder is used for adjusting the Qps, which is used in the entropy decoder for the next group. The third loop involves short-term RC, flatness, and entropy decoder. The proposed task partition optimizes the latency of these loops from rate control.

6.3.4 Line Buffer

In this design, an 64-KB independent memory module is used for line buffer storage. Figure 6.4 shows the line buffer storage organization for different pixel modes. The 64-KB memory module contains 32,768 entries of 16-bit words. The first 16,380 entries are used for luma storage, whereas entries 16,384–32,763 are used for chroma components. In simple 4:4:4 mode, three luma samples and six chroma samples are written to the memory in every group; six luma and six chroma samples are stored per group in 4:2:2 mode; six luma and three chroma are written to line buffer in one group in native 4:2:0 mode. The line buffer read starts in the third last group of first slice line

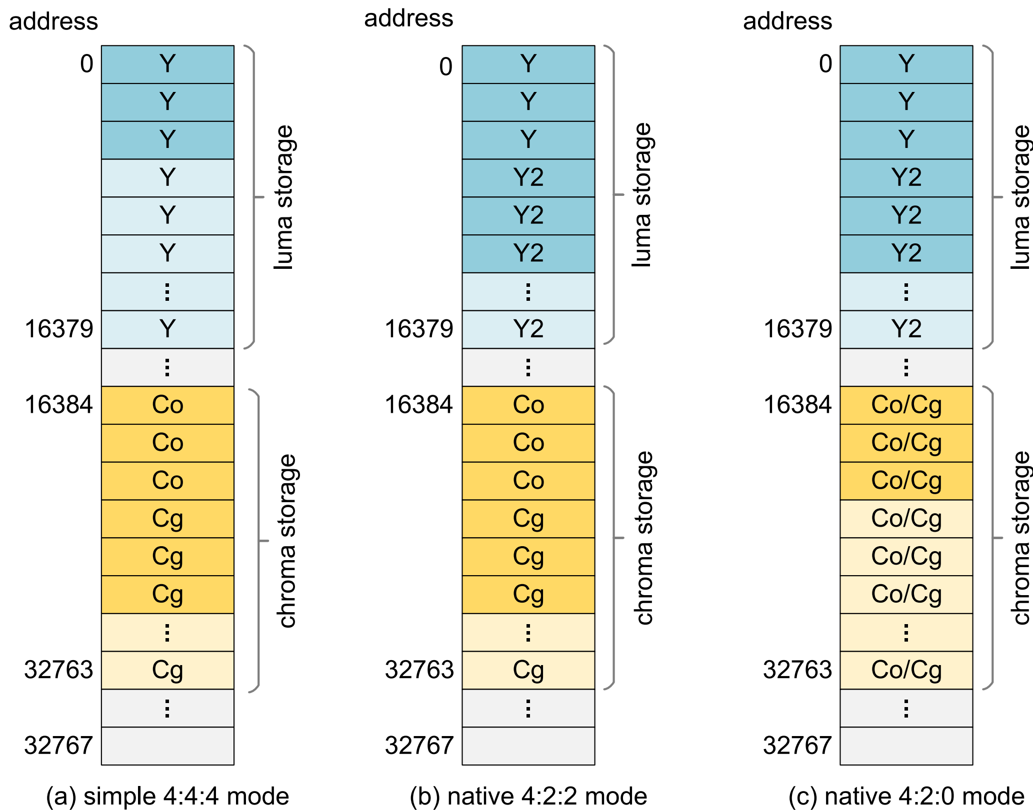


Figure 6.4: Line buffer storage for different pixel components.

(second slice line for chroma data of native 4:2:0 mode) and the same number of samples are read as they are written in every group time.

The dataflow of the line buffer is shown in the bottom of Figure 6.5. The buffer write and read operations are partitioned into separate tasks and mapped to different processors, so that writes and reads can occur independently. The read data are distributed to four processors, each of which stores and sends the data of one component to BP search, prediction, and ICH. Constructing the seven ICH pixels pointing to the previous line is mapped to three processors. In total, nine processors and one memory module are used for the line buffer.

6.3.5 Prediction, Inverse Quantization, and Reconstruction

Figure 6.5 shows the flow for prediction, inverse quantization, reconstruction, and block prediction search (BP search). All three predictors require reconstructed pixels of the previous group, forming a feedback loop. The prediction, inverse quantization, and reconstruction of each component are processed independently with a different set of three processors, which results in

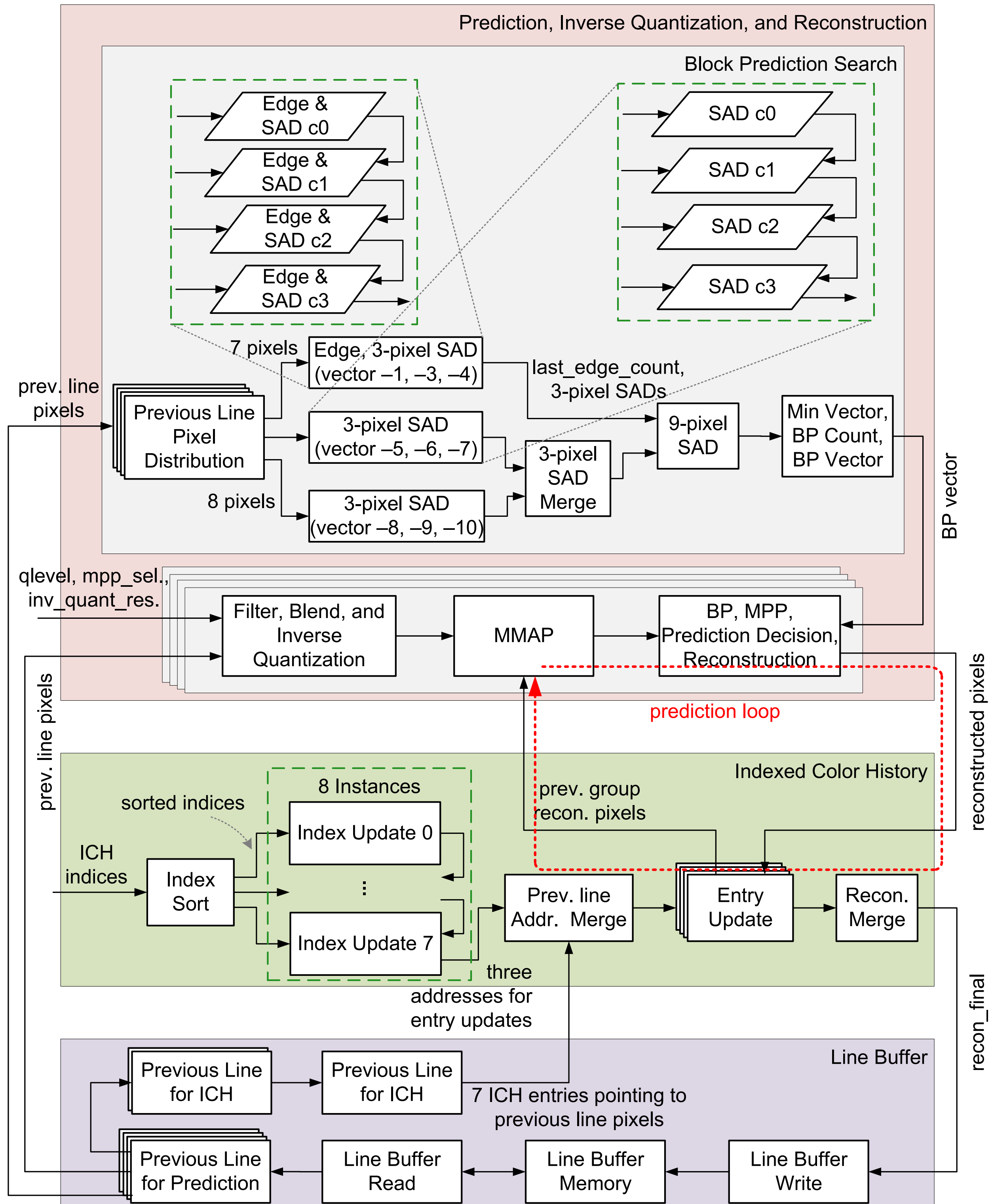


Figure 6.5: Dataflow diagram of prediction, inverse quantization, reconstruction, indexed color history, and line buffer in the slice decoder.

a total of 12 processors. The first processor performs inverse quantization, which uses quantized residuals decoded by the entropy decoders, and the filter and blend operations that use previous line reconstructed pixels for MMAP. The second processor performs MMAP using the blended values and reconstructed pixel values of the previous group. The third processor performs MPP, BP, predictor selection, and reconstruction. As such, only the second and third processor are involved in the prediction loop.

The BP search module is partitioned and mapped to 19 processors. Four processors store and distribute the previous line's reconstructed pixels, where each processor handles one component. In every group, three reconstructed pixels enter these four processors and together with the stored 10 pixels from previous groups, they are used for calculating the SADs for the current group. Twelve processors calculate 3-pixel SADs for the nine vectors, in which one processor calculates the 3-pixel SADs of three vectors for one component independently. The component-wise 3-pixel SADs are added together to form pixel-wise 3-pixel SADs. Specifically, the 3-pixel SADs of the first component are added to the 3-pixel SADs of the same vectors in the second component, the sum of which is added to the 3-pixel SADs of the same vectors in the third component. Finally, the pixel-wise SADs are calculated in the processors which are allocated for the fourth component. To reuse the calculation results, the edge detection is performed in the set of processors that calculates 3-pixel SADs for vectors -1, -3, and -4. The 3-pixel SADs of the previous two groups are reused and added with the SADs of current group to form 9-pixel SADs. The last processor finds the vector with the minimum SADs and determine the BP vector. The current task partitioning of calculating the SADs is the optimal way that uses the smallest number of processors without becoming the throughput bottleneck of the entire decoder.

6.3.6 Indexed Color History

The main challenge of implementing the ICH module on many-core processor arrays is updating the 32 ICH records efficiently in every group. In this work [83], three design approaches are proposed and evaluated: 1) the shift entry approach, 2) the separate entry and index update approach, and 3) the parallel index update approach.

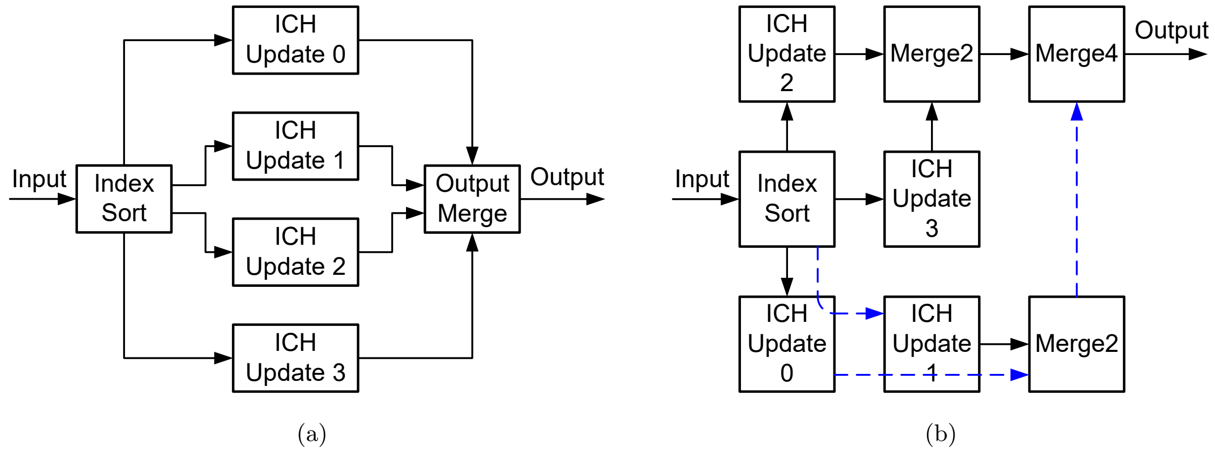


Figure 6.6: The shift entries approach for ICH. (a) Dataflow diagram. (b) Processor array mapping.

Design Approach I: Shift Entries

Figure 6.6(a) shows the dataflow diagram of the *shift entries* approach. It updates the ICH records by serially shifting the ICH entries. This approach consists of three tasks: 1) sort the three ICH indices for the current group, which is used to determine the shift amount of each entry; 2) serially shift the first 25 entries (32 entries for the first line of slices) using the approach discussed in Figure 2.11 and determine the final decoded pixels of the current group based on coding mode. The ICH entries of the three or four components are updated in parallel; 3) merge the the decoded pixel values of the three or four components into a single output stream. Figure 6.6(b) shows an 8-processor mapping of the *Shift-Entry* design which is implemented based on this algorithm. Each task is mapped to one processor, except that merging outputs uses three processors, since each processor has only two input ports and thus can merge only two inputs.

To find the throughput bottleneck of this ICH design, simulations are performed on each processor running alone, which can achieve maximum performance since no I/O stalls are caused by other processors. The processors that update the ICH entries take 561 cycles on average to process one group, while the processors that sort indices, merge two components, and merge all components require 90, 6, and 12 cycles, respectively. Therefore, updating the ICH entries is the bottleneck that limits performance. By running the Shift-Entry design on the KiloCore platform at 1.75 GHz, it achieves 9.1 fps, 9.0 fps, and 4.4 fps in 4:2:0, 4:2:2, and 4:4:4 modes, which is too slow for real-time applications.

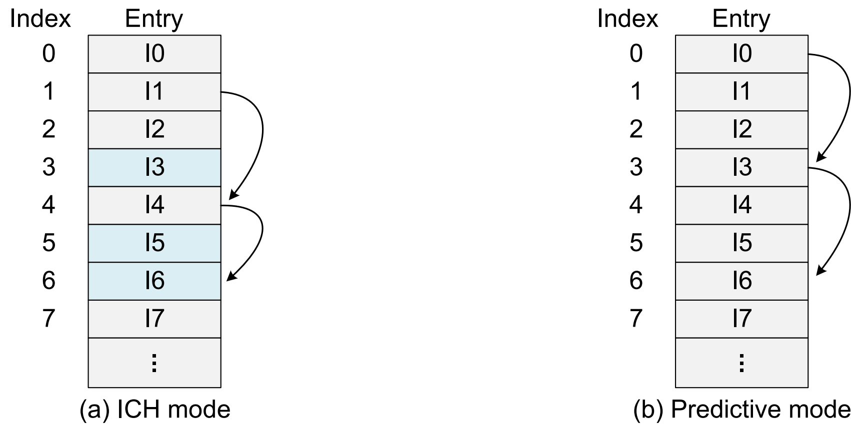
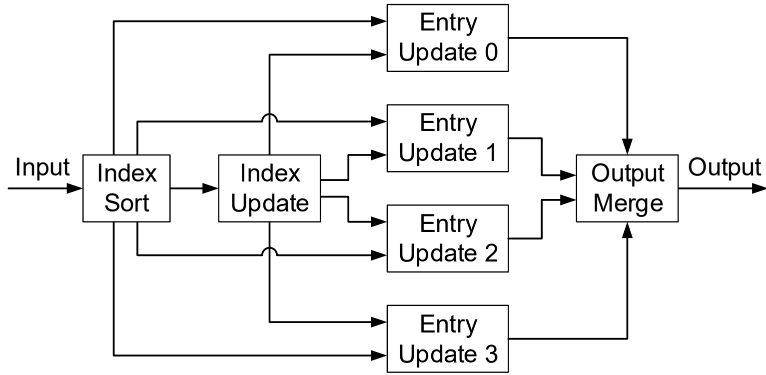


Figure 6.7: Data dependency of the shift entry approach.

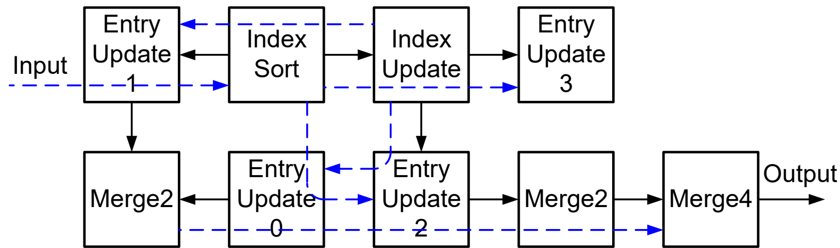
The data dependency prohibits parallelism on the ICH table update task and thus further improvement on throughput is not possible. Figure 6.7 shows the data dependency of shifting the ICH entries. In this example, the ICH mode selects indices 3, 5, and 6. When updating the ICH records, entry 1 is shifted down by three positions and its value is written to entry 4, whereas entry 4 is shifted down by two positions and it is written to entry 6. The value of entry 4 cannot be updated with the value of entry 1 until its original value has been written to entry 6; otherwise, the original value of entry 4 will be overwritten and both entry 4 and 6 will be updated with the value of entry 1. In the case of predictive mode, there are similar data dependencies. For example, entry 3 cannot be updated with the value of entry 0 until its value has been updated to entry 6.

Design Approach II: Separate Entry and Index Update

Motivated by the fact that in every group no more than three new pixels enter the ICH table, while the majority of ICH entries remain in the table with only their indices changed, the *separate index and entry update* approach is proposed which partitions the ICH table update task into index update and entry update tasks. An index table is maintained for ICH indices and an entry table stores ICH entries, where the index of each ICH entry is stored in the same location as the index table. The index table is serially updated by looping through the 32 indices, while the entry table is updated by writing zero to three new pixel values into the appropriate locations, the addresses of which are calculated by the index update task. The dataflow diagram of this approach is shown in Figure 6.8(a).



(a)



(b)

Figure 6.8: The separate index and entry update approach. (a) Dataflow. (b) Processor array mapping.

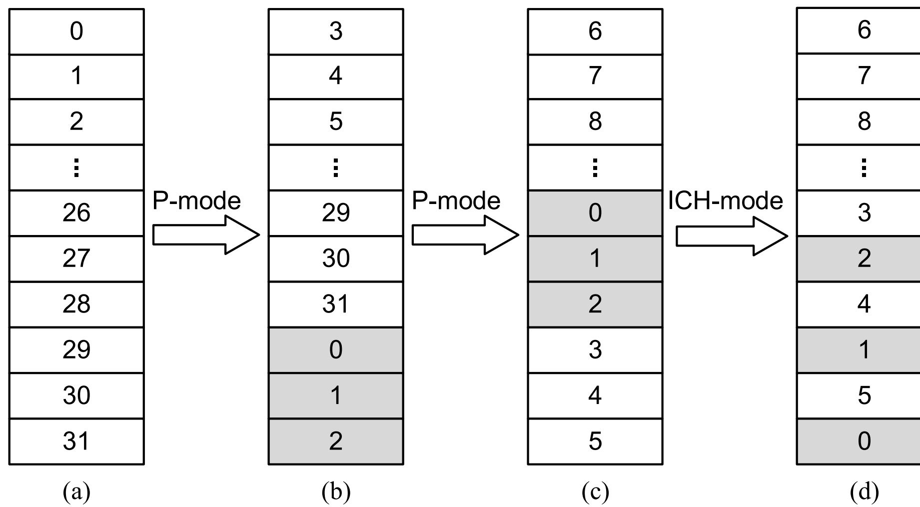


Figure 6.9: Example of index table update, the first location is on the top. (a) Initial state. (b) P-mode is selected. (c) P-mode is selected again. (d) ICH-mode is selected with indices (1, 3, 5).

Figure 6.9 shows an example of updating the index table. The ith word is initialized to i. In this example, the first two groups are P-mode-coded, thus the index values of 29, 30, and 31 are

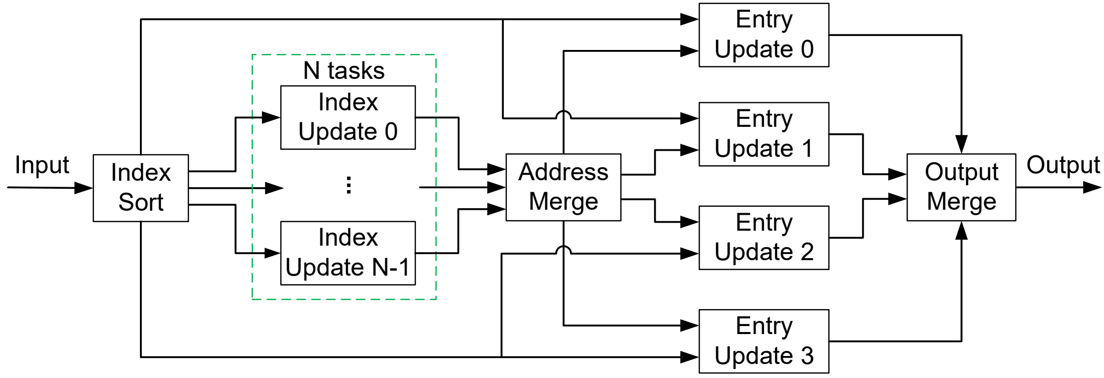


Figure 6.10: Dataflow diagram of the parallel index update approach.

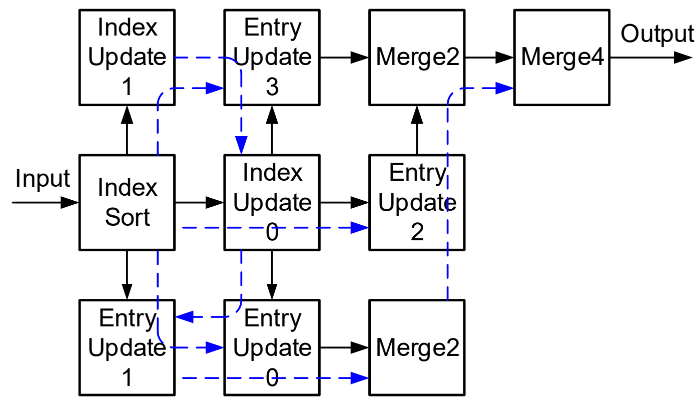
updated to 0, 1, and 2, respectively, while the remaining 29 indices are increased by three, as shown in Figure 6.9(b–c). The third group uses ICH-mode coding with indices (1, 3, 5); therefore, the ICH indices 5, 3, and 1 are updated to 0, 1, and 2, respectively. Index 0 is increased to 3, since it is smaller than all three selected indices; indices 2 and 4 are smaller than two and one of the selected indices, thus they are increased by two and one, respectively; all other indices remain unchanged.

The 9-processor *Split-Index* design is implemented using this approach, as shown in Figure 6.8(b). The simulation on processors running alone shows that the index update processor takes 1,154 cycles to process one group, while 68 cycles are needed for the entry update processor. Therefore, to increase throughput, it is essential to speed up the index update task.

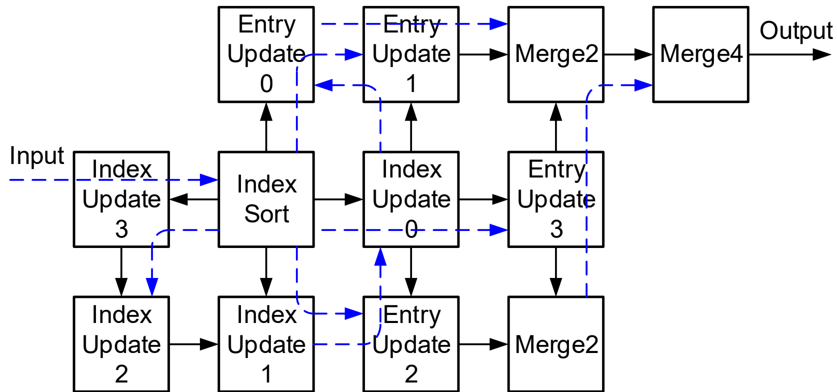
Design Approach III: Parallel Index Update

Unlike shifting ICH table entries, there is no data dependency in the update process between ICH indices. Therefore, the index table update task can be partitioned into multiple parallel subtasks, the outputs of which are merged later. We use N, a number evenly divisible by 32, to denote the number of index update subtasks. Every subtask maintains a table of $32/N$ indices, resulting in N times speed up compared to the non-parallel index update task. Since one subtask does not contain all indices, the three ICH outputs can contain zero to three valid addresses. In addition, for every ICH address, there is one and only one valid output in the N subtasks. By setting the invalid outputs to zero, the ICH addresses of the subtasks are merged together using bitwise OR operations.

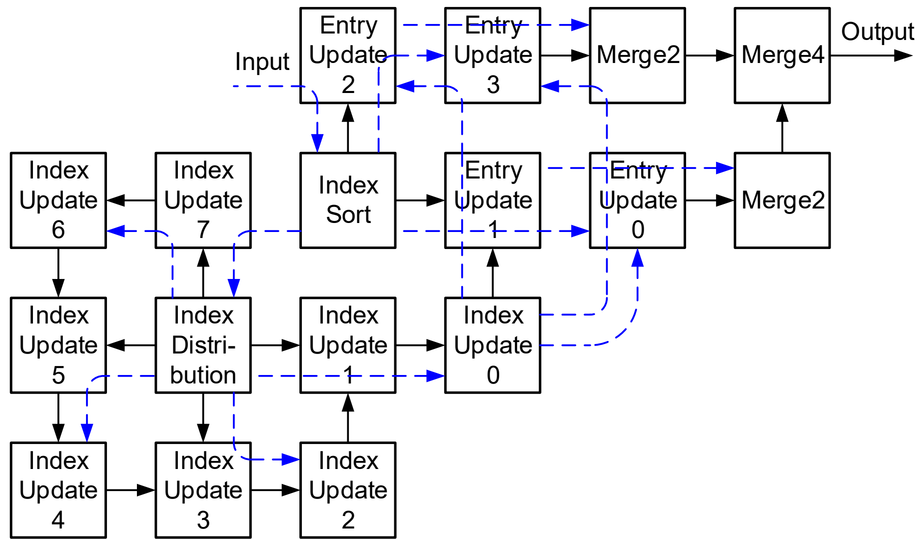
Figure 6.10 shows the dataflow of this approach. For evaluation purpose, five designs



(a)



(b)



(c)

Figure 6.11: Processor mapping of the parallel index update ICH designs. (a) Parallel-Index-2. (b) Parallel-Index-4, and (c) Parallel-Index-8.

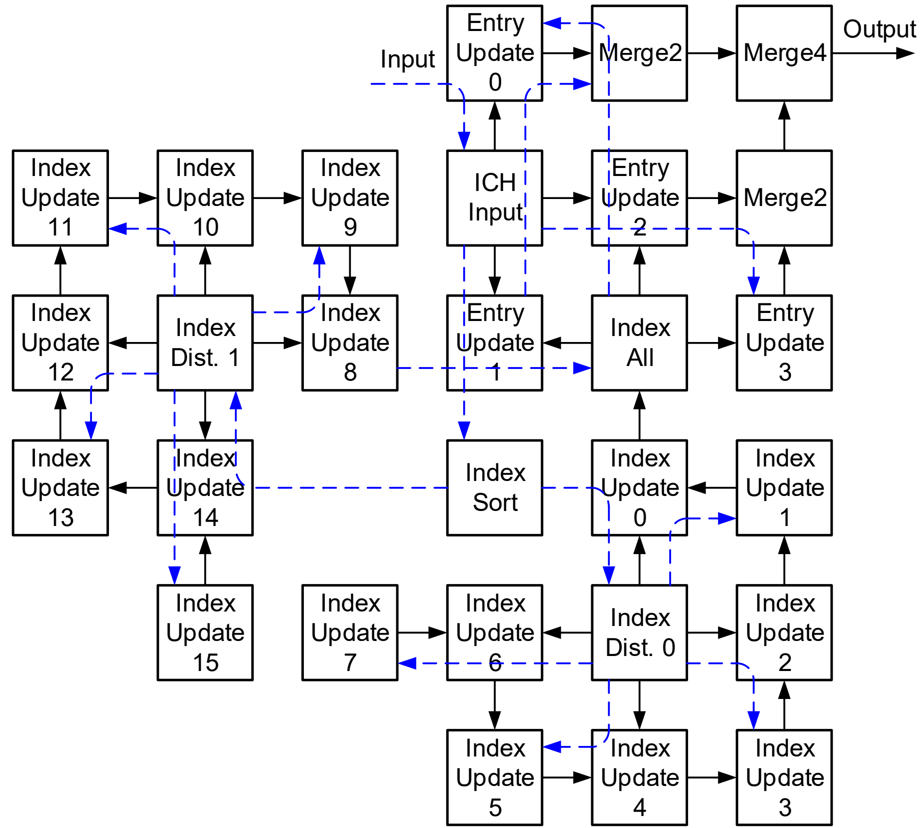


Figure 6.12: Processor mapping of the Parallel-Index-16 ICH design.

are implemented: *Parallel-Index-2*, *Parallel-Index-4*, *Parallel-Index-8*, *Parallel-Index-16*, *Parallel-Index-32*, which correspond to N = 2, 4, 8, 16, and 32, respectively. Each index update subtask is mapped to a different processor. For $N \leq 8$, the ICH addresses are merged in one of the index update processors, which saves an extra merging processor. Parallel-Index-2 and Parallel-Index-4 are mapped to 10 and 12 processors, respectively. Figure 6.11 shows the processor mapping of Parallel-Index-2, Parallel-Index-4, and Parallel-Index-8. Since the index sort task of Parallel-Index-8 needs 12 output connections (four for entry update and eight for index update) but every processor has eight output ports, it is mapped to two processors, where one processor distributes data to the index update processors, resulting in a total of 17 processors. For Parallel-Index-16, four processors are used for the index sort task, where one processor distributes data to entry update processors, one processor sorts the indices, and two processors distribute data to the index update processors. The results of every eight index update processors are merged first, which are then merged using an additional processor. As a result, in total 28 processors are required. Parallel-Index-32 is mapped to 52 processors, out of which four processors distribute data to the 32 index update processors and

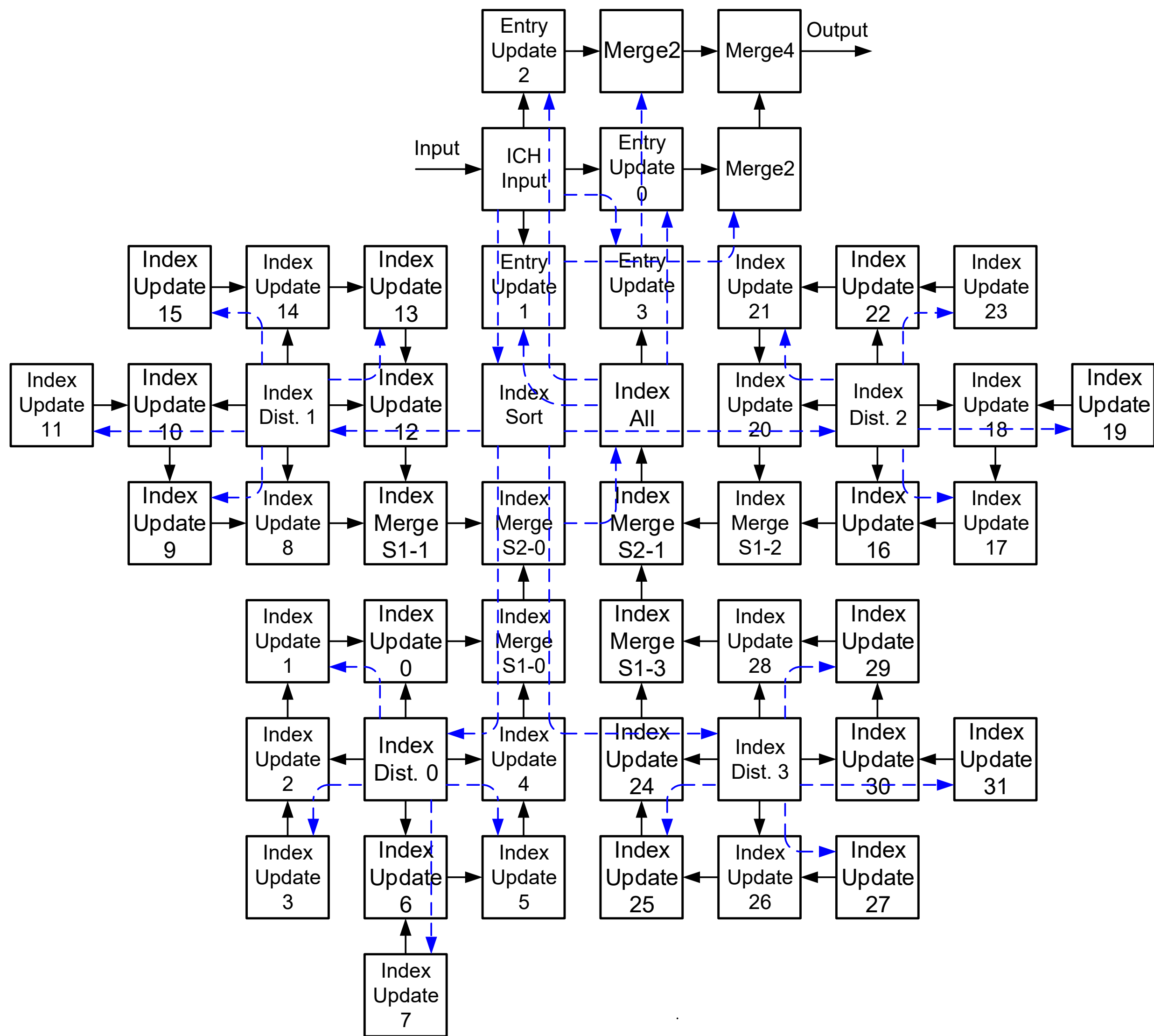


Figure 6.13: Processor mapping of the Parallel-Index-32 ICH design.

seven processors merge the results.

Implementation Results

The seven proposed ICH engine designs are evaluated in a cycle-accurate C++ simulator of the KiloCore chip [80]. In the simulation, the kiloCore processors are set to run at 1.75 GHz with a supply voltage of 1.1 V. Results are obtained from simulating two 1080p frames—a noise image and an image of mandrills. Two reference designs *i7-Shift-Entry* and *i7-Split-Index* implemented with similar methods are evaluated on an i7-7700HQ processor using the same test images.

Figure 6.14 shows the average throughput and the frame rates. The throughput of 4:2:0

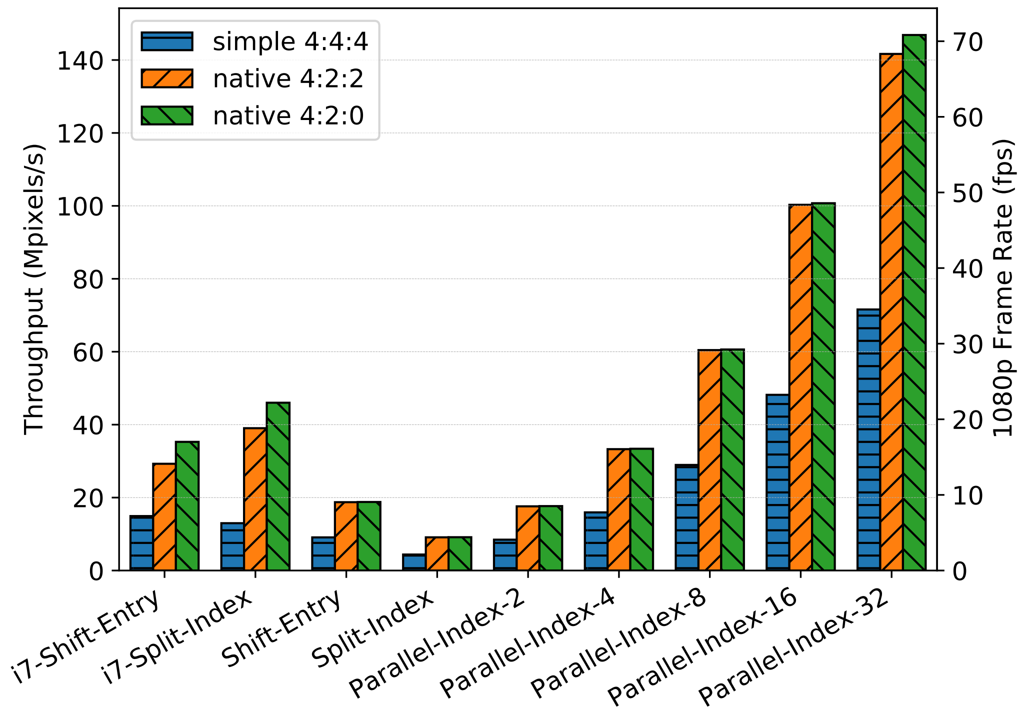


Figure 6.14: Throughput and frame rates of the many-core and i7 ICH designs.

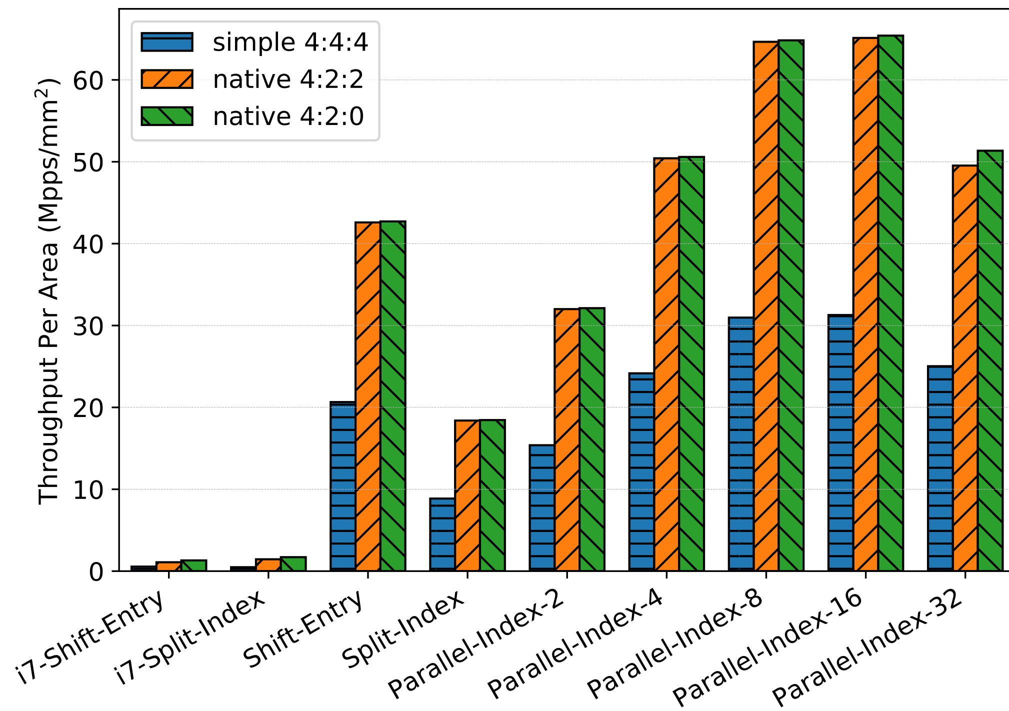


Figure 6.15: Throughput per unit chip area of the many-core and i7 ICH designs.

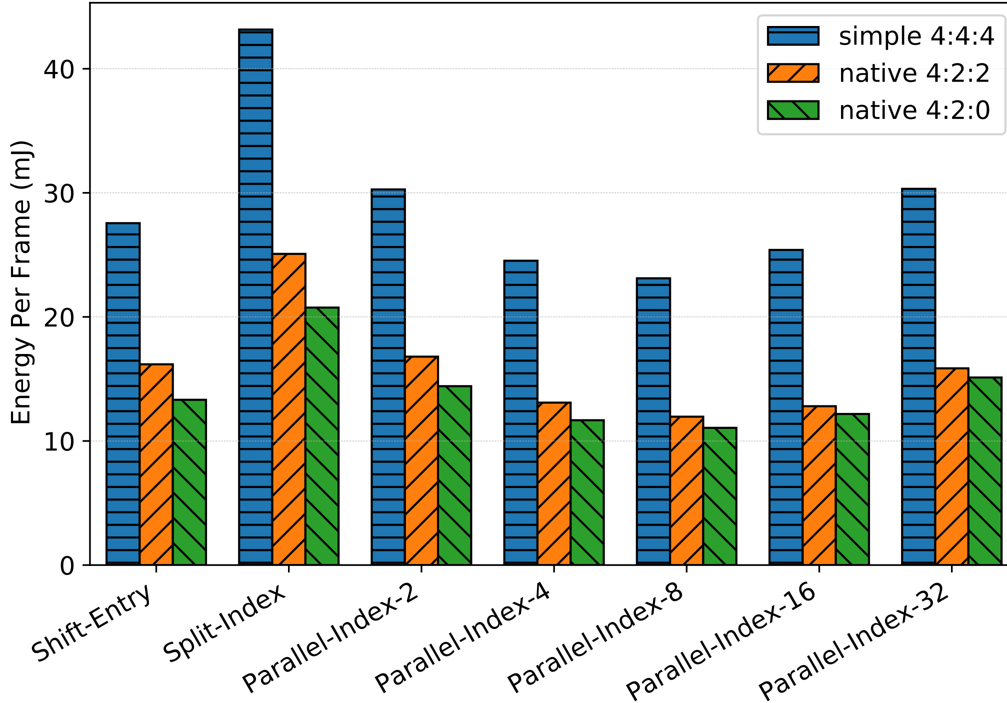


Figure 6.16: Energy per 1080p frame of the seven many-core ICH designs. The i7 reference designs are not shown because they are far off scale.

and 4:2:2 modes are around twice of 4:4:4 mode in the same design, since a container pixel is processed in the same way as a 4:4:4 pixel. Increasing the number of parallel index update processors results in a nearly linear increase in throughput. Parallel-Index-32 has the highest throughput of 146.9 megapixels per second (Mpixels/s) in 4:2:0 mode, 141.7 Mpixels/s in 4:2:2 mode, and 71.6 Mpixels/s in 4:4:4 mode, which are equivalent to 75 fps, 74 fps, and 38 fps at 1080p, and results in a throughput 3.2 times, 3.6 times, and 4.8 times higher than the i7 reference designs.

To fairly compare performance, the throughput is normalized by the total chip area used, as shown in Figure 6.15. For the i7 reference designs, only the area of one core is counted, which we estimate as 8 mm^2 based on a publicly-available die photo. Our highest normalized throughput of 64.9 Mpps/mm^2, 64.6 Mpps/mm^2, and 31.0 Mpps/mm^2 in 4:2:0, 4:2:2, and 4:4:4 modes are achieved in Parallel-Index-16, and are 38 times, 45 times, and 56 times higher throughput per unit chip area than the i7 reference designs.

Figure 6.16 shows the energy consumption to process one 1080p frame. The power of i7 designs are estimated using Intel Power Gadget 3.5 [84]. After scaling to 32 nm using data from

Holt [43], i7-Shift-Entry consumes 1,259 mJ, 1,481 mJ, and 2,929 mJ per frame, and i7-Split-Index consumes 923 mJ, 1,187 mJ, and 3,271 mJ per frame in 4:2:0, 4:2:2, and 4:4:4 modes, respectively. Since 4:4:4 frames contain twice the number of groups compared to 4:2:0 and 4:2:2 frames, 4:4:4 mode consumes around twice energy per frame compared to 4:2:0 and 4:2:2 modes. Parallel-Index-8 is the most energy efficient, featuring 11 mJ, 12 mJ and 23 mJ in 4:2:0, 4:2:2, and 4:4:4 modes. This is 84 times, 99 times, and 127 times lower energy per frame compared to the most energy efficient i7 design.

Now that we have determined the most energy-efficient implementation of the ICH module, the proposed slice decoder uses the parallel index update approach for the ICH module and uses eight processors to update the ICH indices. Further parallelizing the index update process does not improve the throughput of the decoder but increases design area. Figure 6.5 shows the dataflow of the ICH module in the slice decoder. To reduce the latency of the prediction loop (highlighted in Figure 6.5), the reconstructed pixel values are directly fed into the entry update processors; another processor merges the addresses from the index updates and seven ICH entries pointing to the previous line. Three processors merge the final reconstructed pixels which are written into the line buffer memory. In total, the ICH module is mapped to 17 processors.

6.3.7 Mapping Slice Decoder to a Many-Core Processor Array

Figure 6.17 shows an example of mapping (i.e., assign the logical tasks and data flows to physical processors and communication links on a chip) the slice decoder to 88 processors and 2 independent memory modules of the KiloCore chip. Figure 6.18 shows the area and energy dissipation breakdown of processor and memory resources utilized by the slice decoder. The area of processors and independent memory modules used in the decoder is considered in this analysis. The prediction module is the largest block and accounts for one third of the total area in the decoder, due to the high computation requirement in BP search and component-level parallelism. The combined area of prediction and ICH is over 50% of the entire decoder. Native 4:2:2 mode is considered in the energy dissipation analysis, since the fourth pixel component is used only in this mode. Energy dissipation is correlated with the chip area used; therefore, energy breakdown is similar to the area breakdown.

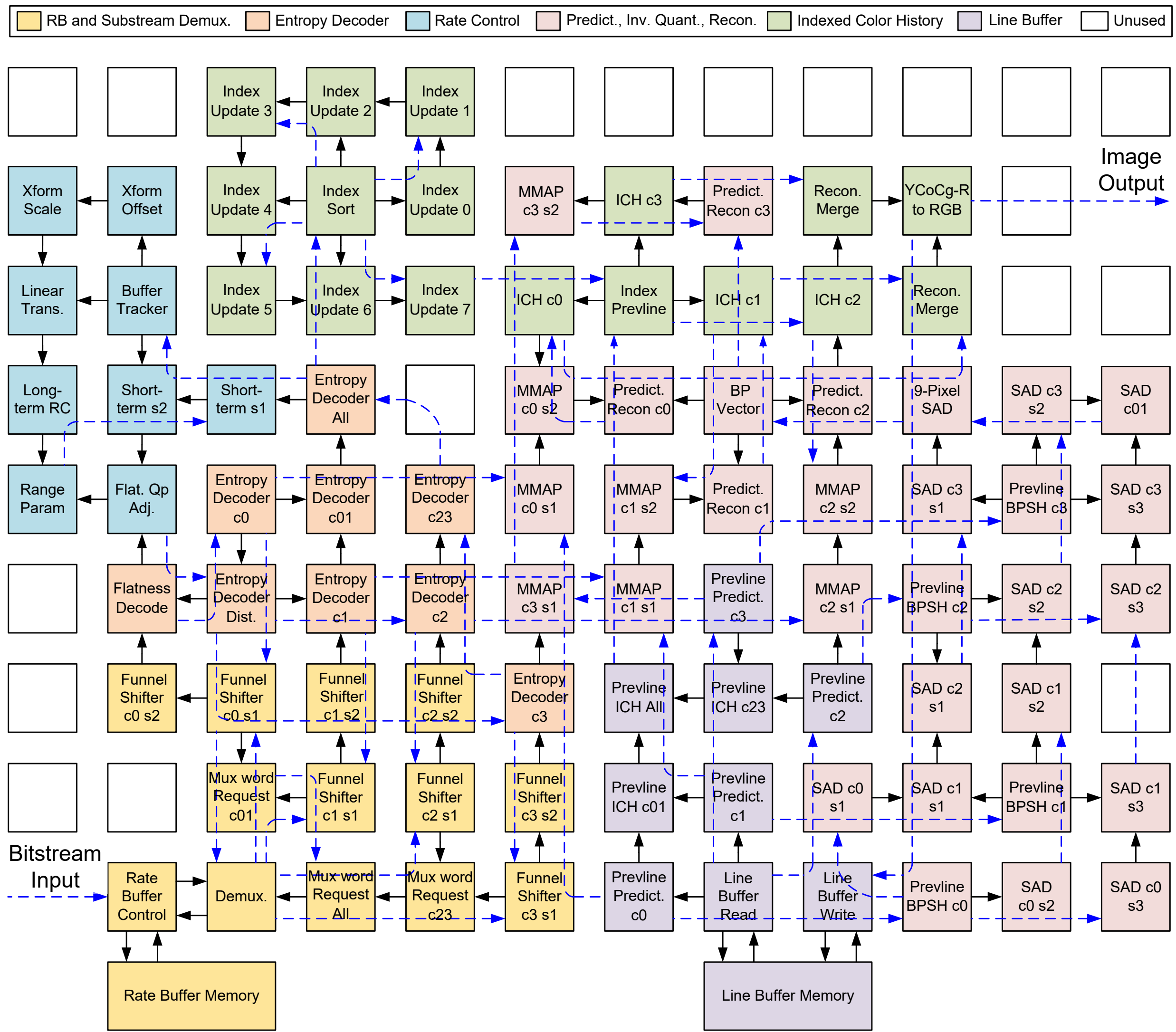


Figure 6.17: Processor array mapping of the slice decoder by an automatic mapping tool. For clarity, unused inter-processor communication links are omitted. Processors corresponding to pixel components contain "c0", "c1", "c2", or "c3" in their names; "c01" and "c23" denote processors performing merging operations. Tasks mapped to multiple processors are named with a suffix of "s1", "s2", or "s3" to denote the stage of computation.

6.4 Parallel Slice Decoders

6.4.1 High-Level Dataflow

The *parallel slice decoders* achieve higher performance than the slice decoder by utilizing slice-level parallelism, i.e., they process multiple columns of slices in parallel. Every column of slices is decoded with a separate *modified slice decoder*—the same as the slice decoder except without a

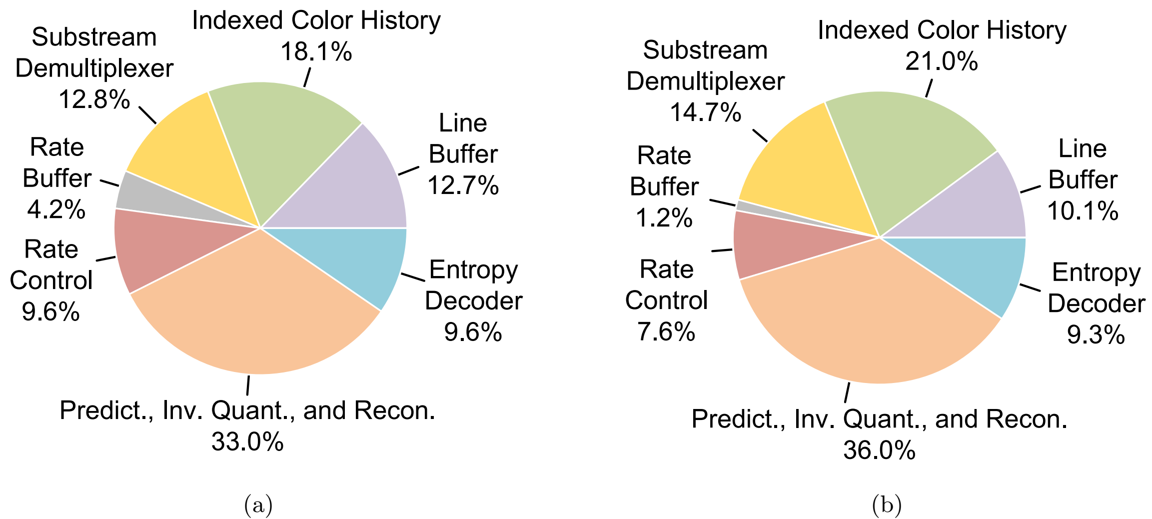


Figure 6.18: Many-core software slice decoder breakdown. (a) Area. (b) Energy dissipation.

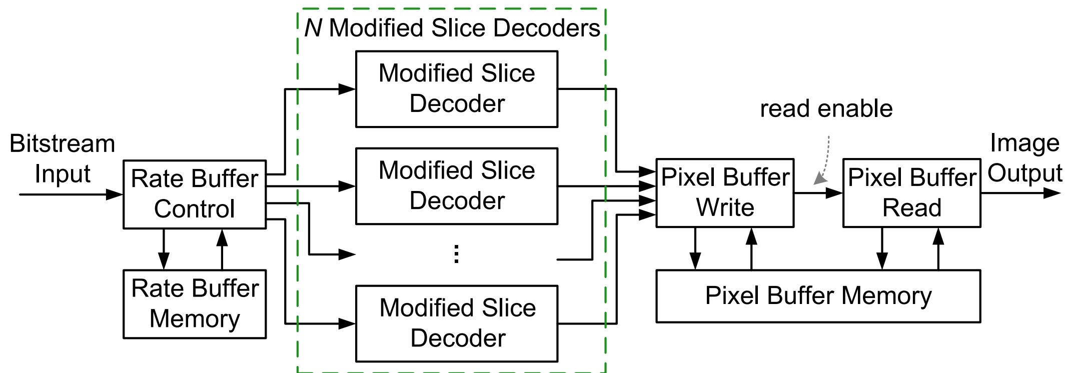


Figure 6.19: Dataflow diagram of the parallel slice decoder.

rate buffer. Instead, one rate buffer memory is shared by all columns of slices. Figure 6.19 shows a high-level generic dataflow of the parallel slice decoders. At the same voltage, when the number of modified slice decoder is increased, throughput scales up linearly, while energy efficiency remains almost the same.

In the parallel slice decoder, it is essential to make sure all modified slice decoders are synchronized. In every group time, the decoder reads the budgeted amount of data from the input bitstream and writes it into the rate buffer memory, where every column of slices is allocated a dedicated portion of memory locations, as shown in Figure 6.20. The rate buffer write side uses a pixel counter to record the position of the current group within the slice. Rate buffer read for all columns of slices starts after the rightmost column of slices has passed the initial decoding delay

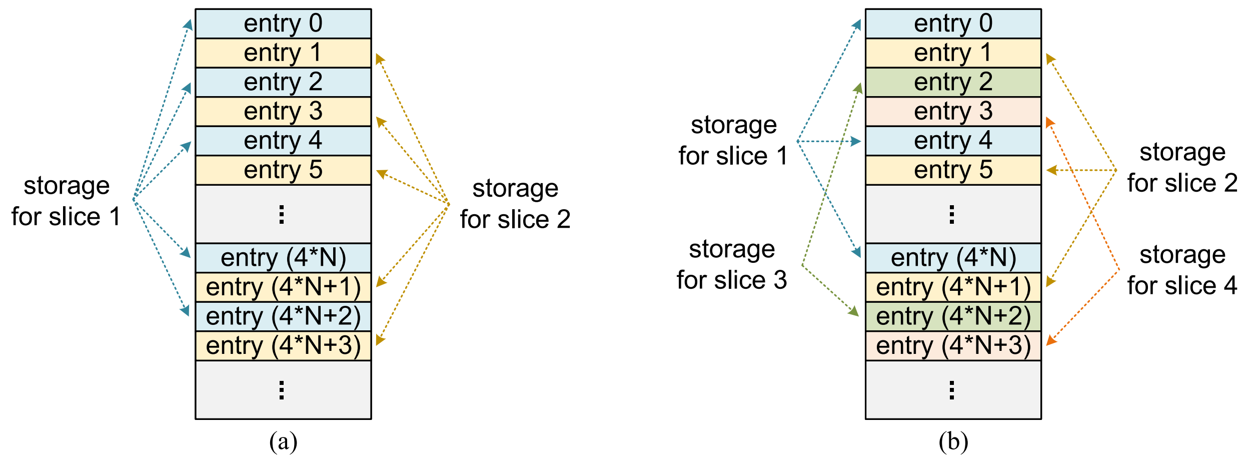


Figure 6.20: Rate buffer and pixel buffer memory storage organization for parallel slice decoders. (a) Two slices per line. (b) Four slices per line.

stage. On the buffer read side, since the number of bits to code each group can vary significantly, only the requested number of mux words are sent to each modified slice decoder. This strategy helps make sure that all the modified slice decoders are in the same pace.

Another challenge is to rasterize the decoded pixels. Since the modified slice decoders are synchronized, and the output pixels of each slice are in raster-scan order within the slice, our solution is to implement a pixel buffer: write the output pixels of all modified slice decoders into a independent memory and read them out in raster-scan order whenever one picture line of pixels has been written. The purpose of the pixel buffer is to temporarily store the pixels of the modified slice decoders when they are not used as the decoder output, so that the modified slice decoders do not stall on output writes. To facilitate concurrent pixel buffer writes and reads, separate processors are used for write and read control. When more than two modified slice decoders are used, some extra processors are used to merge the decoded pixels.

The bitstream data and decoded pixels of different columns of slices are stored in different portions of the memory in an interleaved manner. Figure 6.20 illustrates the storage organization for the rate buffer and pixel buffer in the proposed parallel slice decoders. For two slices per line, the even-position entries (i.e., entry 0, 2, 4, ...) store the data of the first column of slices, whereas the odd-position entries (i.e., entry 1, 3, 5, ...) store the data of the second column of slices. In case of four slices per line, the entry $4 * N$, $4 * N + 1$, $4 * N + 2$, $4 * N + 3$, where N is equal to zero or a positive integer, are used for the first, second, third, and fourth column of slices. For

N number of slices per line, the buffer read and write addresses for each column of slices always use a stepsize of N. Compared to allocating contiguous memory block of addresses for each slice, the proposed interleaved memory allocation scheme does not need to manually check whether the address has reached the end of the allocated memory blocks. Instead, by simply increasing the address, it automatically goes back to the start of the memory locations once the end of the memory is reached.

6.4.2 Parallel-2-Slice Decoder

Figure 6.21 shows the dataflow and KiloCore processor array mapping diagram of the parallel-2-slice decoder, which uses two modified slice decoders to process two slices in parallel. It is implemented with 178 processors and 4 independent memory modules, which occupy a total area of 10.44 mm^2.

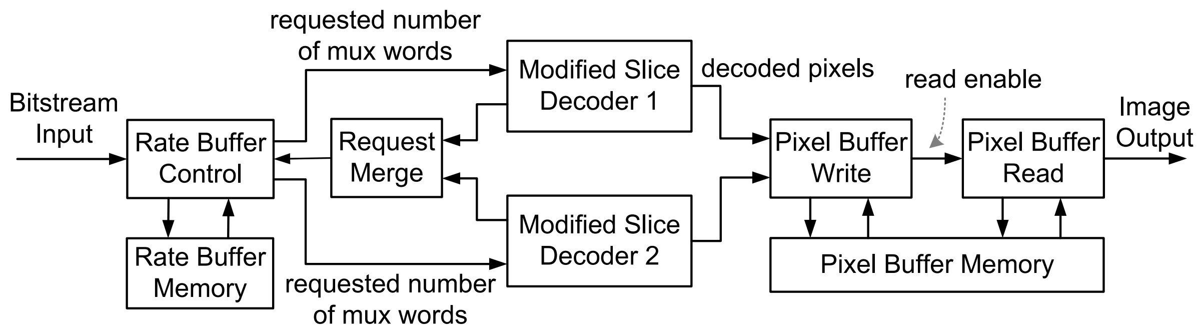
6.4.3 Parallel-4-Slice Decoder

Figure 6.22 shows the dataflow and mapping on the KiloCore processor array of the parallel-4-slice decoder using an automatic mapping tool. This design is implemented on the KiloCore processor array utilizing 359 processors and 6 independent memory modules, which occupy a total area of 20.73 mm^2. Note that the KiloCore has 1,000 processors and the decoder is mapped to the bottom half of the array. Each modified slice decoder is mapped to 87 processors and 1 independent memory module; both the rate buffer and pixel buffer uses an independent memory module and their read and write control are mapped to 4 processors; 2 processors distribute the mux words to the four modified slice decoders to avoid routing congestion; 3 processors merge the mux word requests; 2 processors merge the decoded pixels from the four modified slice decoders.

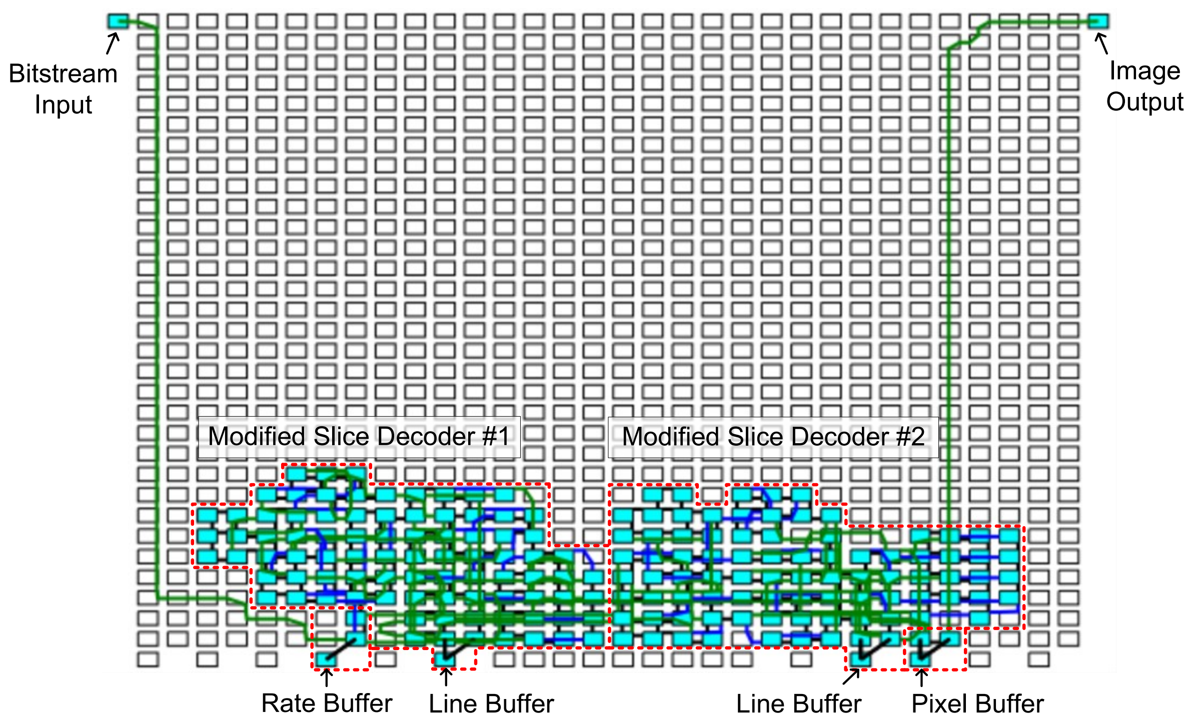
6.5 Simulation Results and Analysis

The proposed many-core software slice decoder and parallel slice decoders are implemented and evaluated on a cycle-accurate C++ simulator of the KiloCore chip [80]. They fully comply with DSC version 1.2a, and support 8 and 10 bits per component (bpc) in constant bit rate (CBR) mode.

To add support for 12, 14, and 16 bpc, the rate buffer, substream demultiplexer, and entropy decoder need to be updated. For 12 bpc or higher, a mux word needs to be split into four

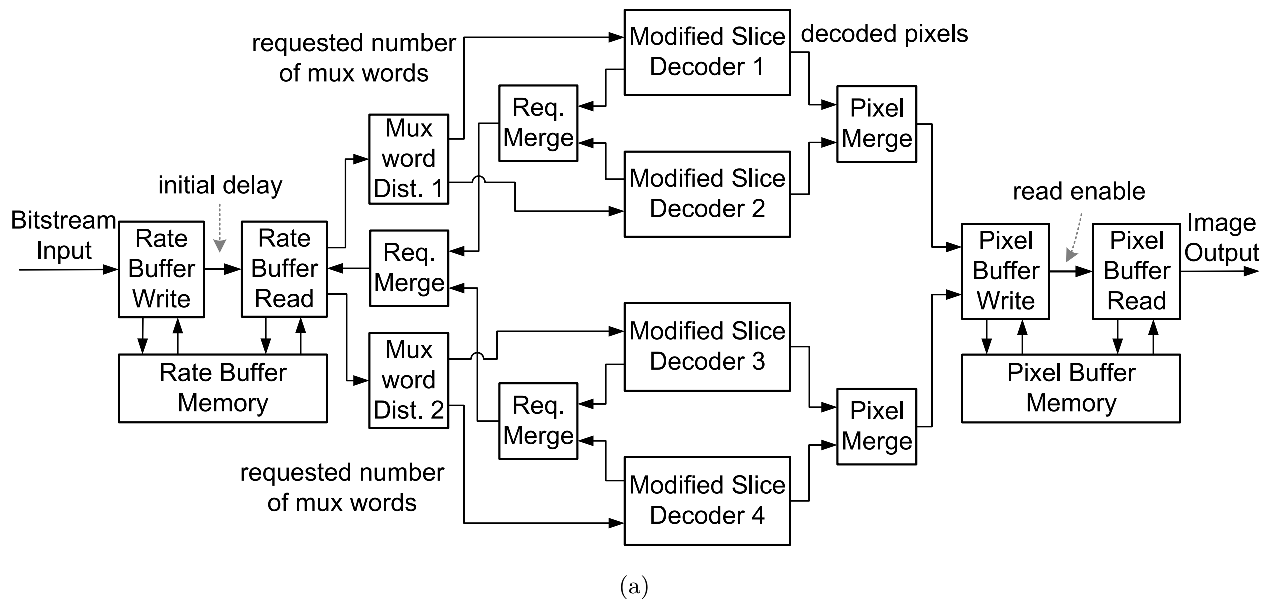


(a)

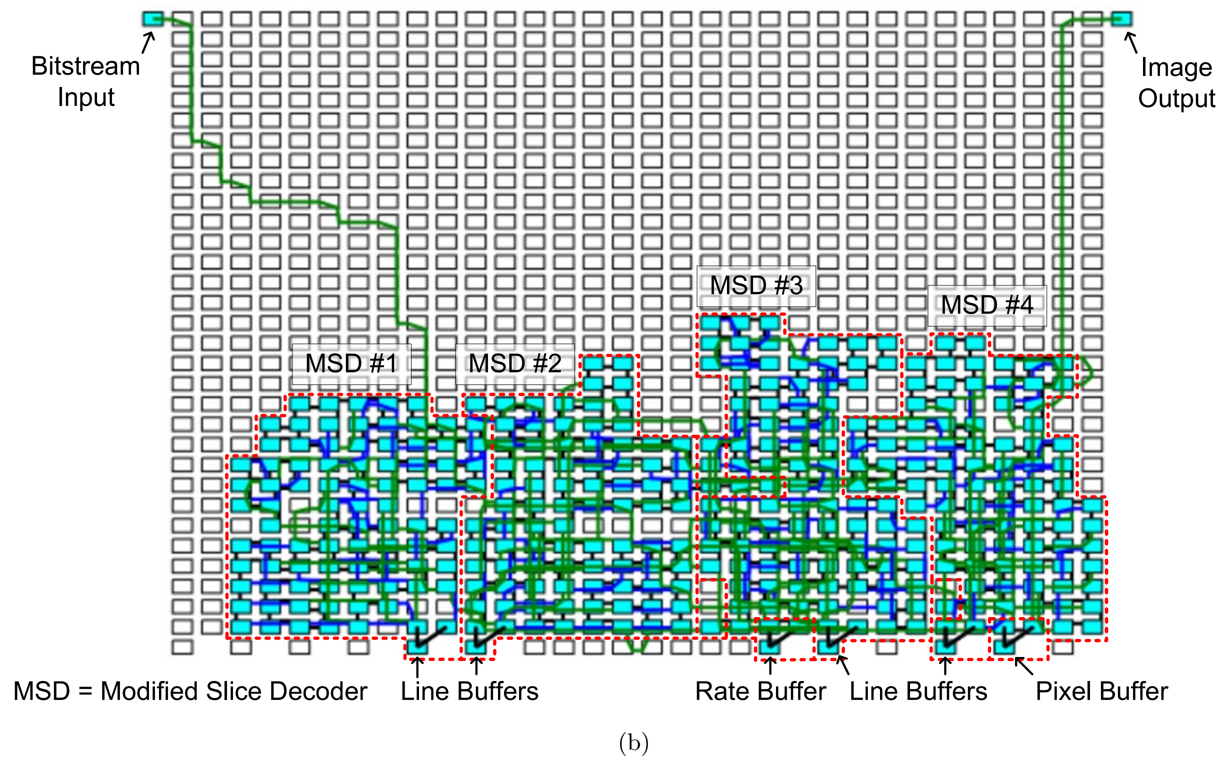


(b)

Figure 6.21: Parallel-2-slice decoder. (a) Dataflow diagram. (b) Many-core processor array mapping. The two links to the input and output ports (located at the top-left and top-right corners of the array, respectively) are long because of the relative placement of the I/O ports and independent memory modules. Black, blue, and green lines represent inter-processor communication links, according to the link length.



(a)



(b)

Figure 6.22: Parallel-4-slice decoder. (a) Dataflow diagram. (b) Many-core processor array mapping. The two links to the input and output ports (located at the top-left and top-right corners of the array, respectively) are long because of the relative placement of the I/O ports and independent memory modules. Black, blue, and green lines represent inter-processor communication links, according to the link length.

Table 6.1: Simulation Voltages and Frequencies of KiloCore Processors and Memories

Voltage (V)	0.76	0.80	0.85	0.90	0.95	1.00	1.05	1.10
Processor (MHz)	678	831	1027	1217	1395	1548	1669	1751
Memory (MHz)	656	831	1047	1250	1431	1577	1686	1739

16-bit words, and a syntax element needs to be stored as four 16-bit words, both of which require one more 16-bit word than 8 and 10 bpc. Besides, the funnel shifters will require seven words for 12 bpc and 8 words for 14 and 16 bpc, whereas six words are needed for 10 bpc. Therefore, more operations are required to read/write and transmit the mux words and syntax elements and update the funnel shifters, which is likely to degrade the performance.

To add support for variable bit rate (VBR) mode, some logic needs to be added to the rate buffer to manually determine the chunk boundaries, since each chunk can contain different number of bits. The buffer level tracker in the rate control module needs to clamp the buffer fullness to zero if underflow happens.

6.5.1 Experimental Setup

The decoders are simulated in native 4:2:0, native 4:2:2, and simple 4:4:4 with RGB output format, respectively, all of which are under 8 bpc and 8 bpp in CBR mode. In addition, block prediction is enabled and the line buffer component bit depth is set to 8 bpc. Full error precision is used in the ICH and P-mode decisions. Slice height is 108 lines, and the slice width is 1920 for slice decoder, 960 for parallel-2-slice decoder, and 480 for parallel-4-slice decoder. Default rate control parameters (see Annex E in the DSC standard [22]) are used. Three 1080p (1920×1080) images (t_1024x1024Cr, t_1280x768_Noise_128, and t_Mandrill) are tested in the simulation and the average results are reported in this section.

The simulations are conducted setting the KiloCore processors and independent memory modules to the maximum frequencies allowed at each supply voltage from 1.10 V down to 0.80 V with a stepsize of 50 mV, as well as the minimum operating voltage of 0.76 V for memories. Table 6.1 summarizes the voltages and the corresponding frequencies of the processors and independent memories that are used in the simulation.

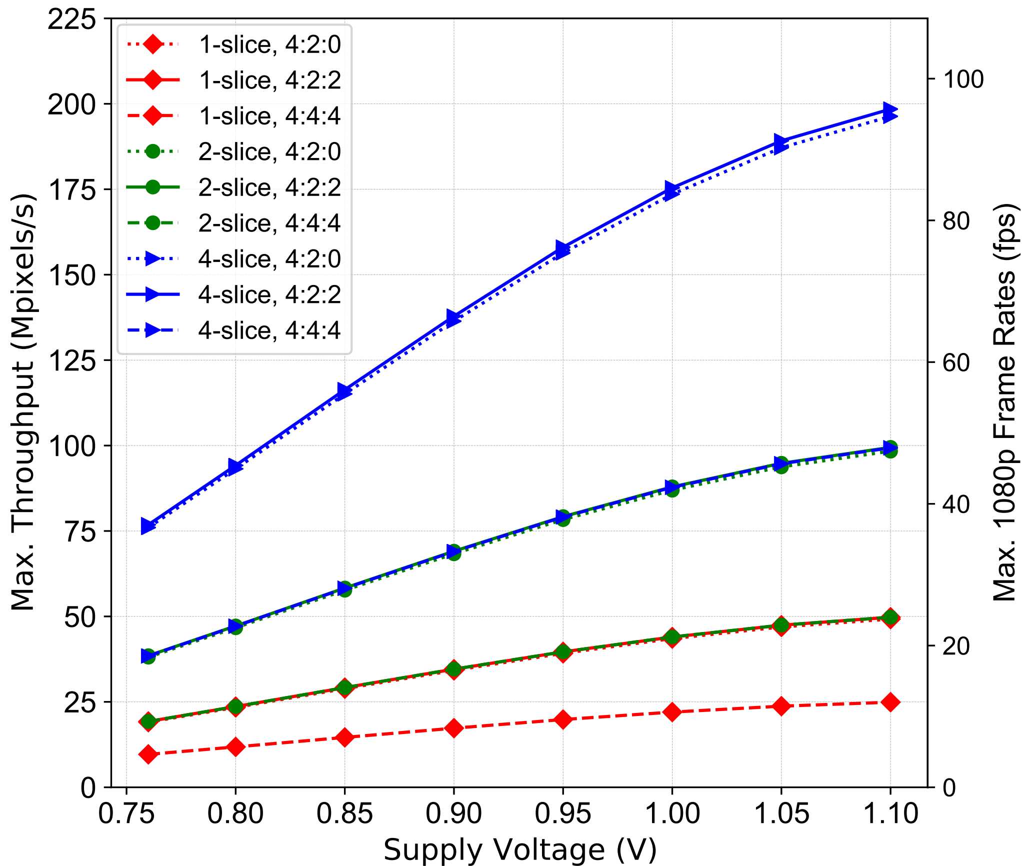


Figure 6.23: Throughput versus voltage of the many-core software DSC decoders.

6.5.2 Throughput

Figure 6.23 plots the throughput of the many-core software DSC decoders. The parallel-2-slice decoder and parallel-4-slice decoder achieve 2 and 4 times the throughput of slice decoder in the same pixel mode at the same voltage, respectively—this shows a linear increase of throughput with the increased number of parallelly processed slices. For each proposed decoder, native 4:2:0 and 4:2:2 modes have almost the same throughput which is twice the throughput of simple 4:4:4 mode due to the fact that two neighboring pixels are processed as one container pixel in native 4:2:2/4:2:0 modes. From 0.76 V to 1.1 V, the throughput increases by 2.58 times. At 1.1 V and 1.75 GHz, the parallel-4-slice decoder achieves 196.34 Mpixels/s, 198.46 Mpixels/s, and 99.33 Mpixels/s in native 4:2:0, 4:2:2, and 4:4:4 modes, which is equivalent to 94.7 frames per second (fps), 95.7 fps, and 47.9 fps in 1080p, respectively.

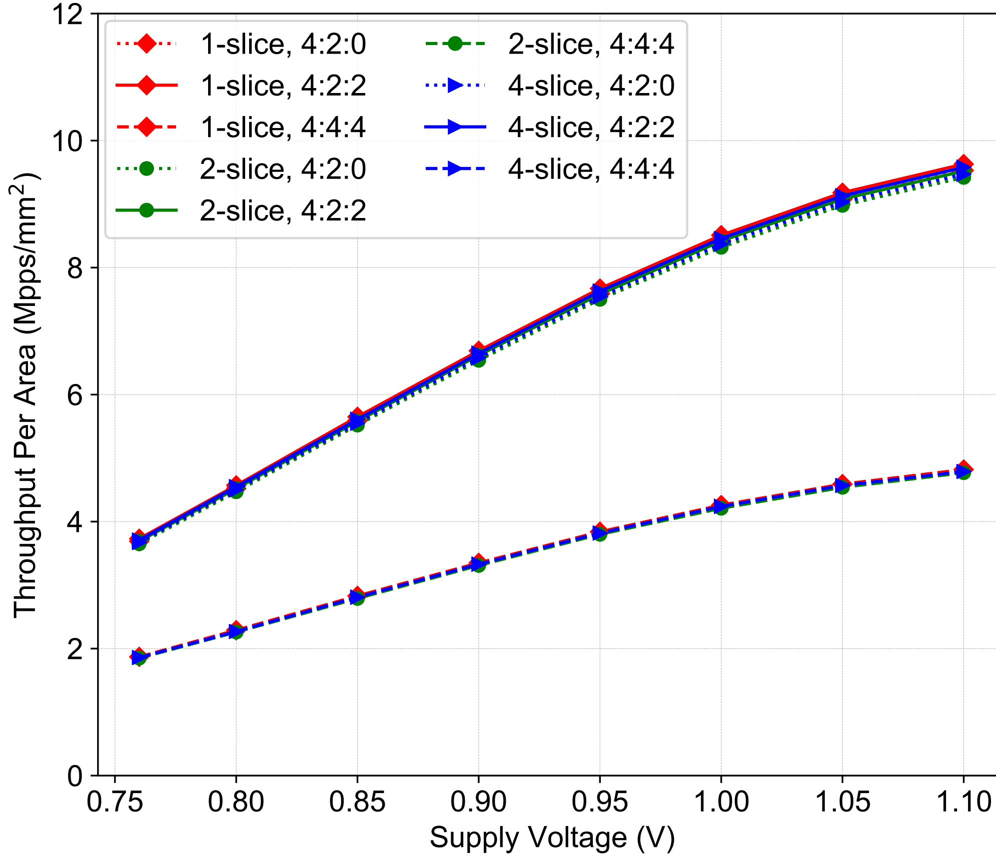


Figure 6.24: Throughput per area versus voltage of the many-core software DSC decoders.

6.5.3 Area Efficiency

Assuming the unused processor cores and memory modules in the array can be used for other applications, the area of each decoder is considered as the total area of the processors and memory modules used. Therefore, the chip area used by the slice decoder, parallel-2-slice decoder, and parallel-4-slice decoder are 5.17 mm^2, 10.44 mm^2, and 20.73 mm^2, respectively. Figure 6.24 plots the throughput per chip area, which is calculated by dividing the throughput by the chip area used by each decoder. At the same voltage and pixel mode, the throughput per area of the slice decoder is 1% larger than the parallel slice decoders. The throughput per area in native 4:2:2 is 1.01 and 2 times larger than native 4:2:0 and 4:4:4 modes, respectively. The highest area efficiency is achieved in the slice decoder at 1.10 V, which is 9.53 Mpps/mm^2, 9.63 Mpps/mm^2, and 4.82 Mpps/mm^2 for pixel modes of native 4:2:0, native 4:2:2, and simple 4:4:4, respectively.

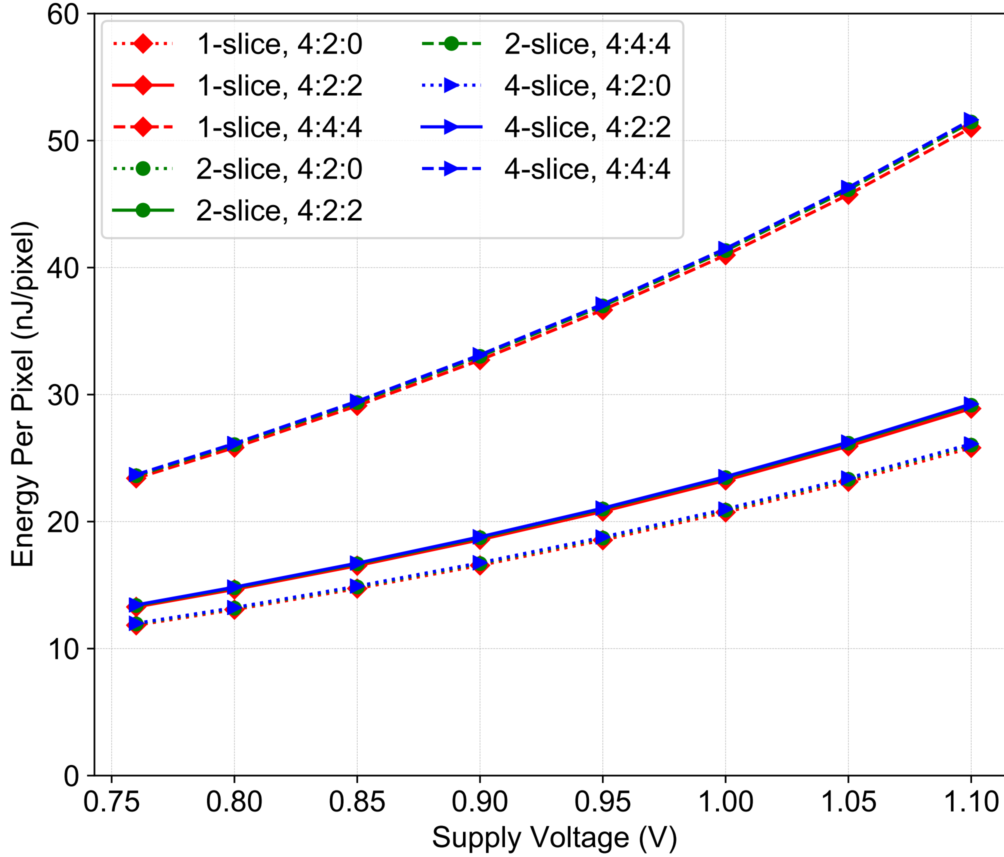


Figure 6.25: Energy per pixel versus voltage of the many-core software DSC decoders.

6.5.4 Energy Efficiency

Figure 6.25 plots the average energy dissipation to decode one pixel at various supply voltages. In the same pixel mode, the energy per pixel of the three decoders are very close and with the increased number of modified slice decoders, the energy efficiency decreases very slightly—the parallel-2-slice decoder dissipates 0.8%–0.9% more energy per pixel than the slice decoder, whereas the parallel-4-slice decoder dissipates 0.3%–0.4% more energy per pixel than the parallel-2-slice decoder. At the same voltage, native 4:2:2 mode consumes 12% more energy per pixel than 4:2:0 since native 4:2:2 mode has one more component than native 4:2:0 mode, whereas 4:4:4 mode dissipates 1.98 times the energy per pixel than native 4:2:0 mode. Energy dissipation decreases by 2.18 times when reducing the supply voltage from 1.10 V to 0.76 V. The minimum energy per pixel is achieved in slice decoder at 0.76 V, featuring 11.84 nJ, 13.28 nJ, and 23.41 nJ per 4:2:0, 4:2:2, and 4:4:4 pixel.

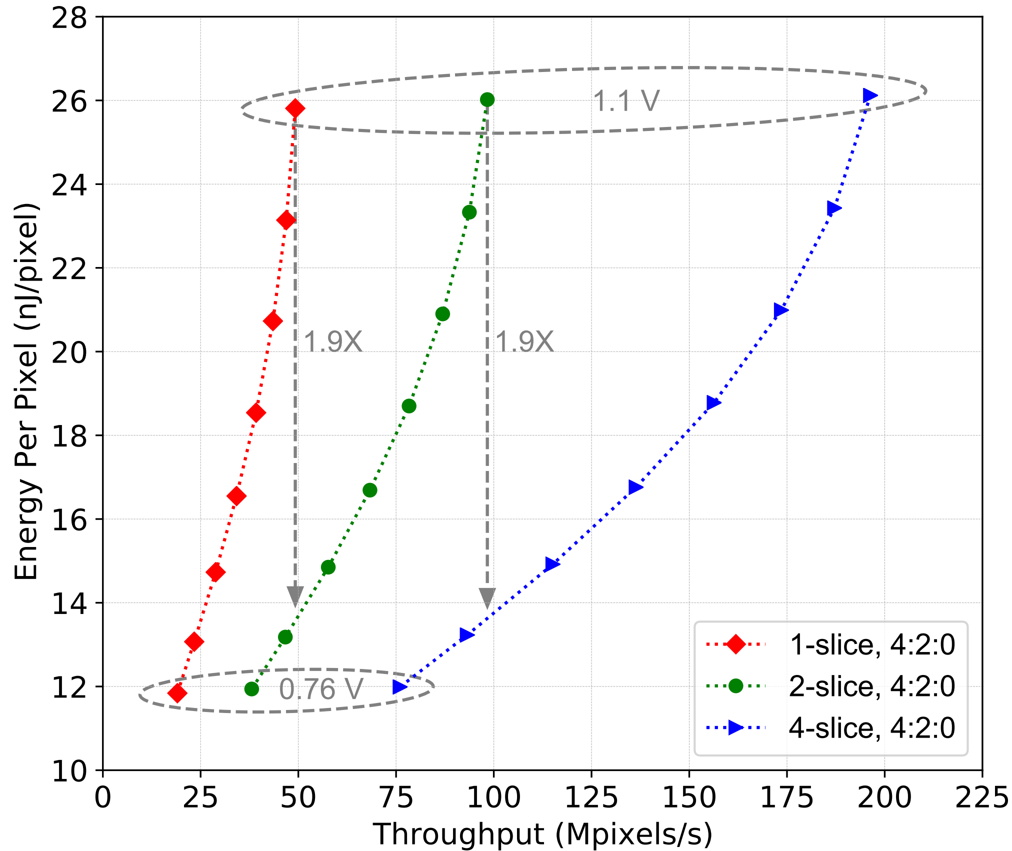


Figure 6.26: Energy per pixel versus throughput of the many-core software DSC decoders in native 4:2:0 mode at various voltages.

Figure 6.26 plots the energy per pixel versus throughput at various voltages in native 4:2:0 mode for the three proposed decoders. To meet a given throughput target, the parallel slice decoders can achieve better energy efficiency via voltage scaling. For example, at 0.81 V, the parallel-2-slice decoder and parallel-4-slice decoder achieve the same throughput as slice decoder and parallel-2-slice decoder at 1.1 V, respectively. In both case, the energy per 4:2:0 pixel is reduced by 1.9 times.

6.5.5 Scalability for Higher Resolutions and Performance

The size of line buffer, pixel buffer, and rate buffer are related to the width of picture; therefore, larger buffers are needed for higher resolutions. The proposed DSC decoders can support higher resolutions such as 4K and 8K without any design modifications, since all three buffers use the 64-KB memory modules, which are sufficient to support 8K. The throughput (Mpixels/s) and

energy efficiency for 4K and 8K are expected to be similar to that for 1080p.

The parallel slice decoders are fully scalable to achieve higher throughput by increasing the number of parallel modified slice decoders. More processors will be required to distribute mux words, merge the requests for mux words, and the decoded pixels.

6.5.6 Comparison With Other Software Video Decoders

Table 6.2 compares the proposed DSC decoders with the DSC C model provided by VESA, and H.264/AVC, HEVC, and VVC decoders.

Compared to the DSC C model provided by VESA that runs on one core of an Intel i7-7700HQ processor, the proposed parallel slice decoders achieve up to 159 times higher throughput, 769 times lower energy per pixel, and 192 times higher throughput per chip area—this shows the performance and energy efficiency benefits of the many-core design approach over general purpose processors.

The H.264/AVC intra-frame decoder by Zhu *et al.* [85] uses a hardware accelerator and single instruction multiple data (SIMD) instructions, which results in increased throughput, reduced area and energy per pixel. Nevertheless, our parallel slice decoder achieves 2.6 times higher throughput than [85]. Moreover, our designs achieve 3.4 times higher throughput and 20 times lower energy per pixel than a HEVC decoder [17] on an Intel i7-3720QM processor. Despite lower throughput, our designs achieve up to 7.6 times and 7.8 times lower energy per pixel than a HEVC decoder [86] on a 6-core Intel i7 E5-1650 processor and a Versatile Video Coding (VVC) decoder [87] on an Intel i9-9980HK processor, respectively.

6.6 Summary

This chapter presents the software design of DSC decoders for fine-grained many-core processor arrays. The KiloCore chip is used as the targeted computation platform. The slice decoder exploits task-level parallelism of the DSC decoding algorithm and maps tasks to the small processors. Throughput and area are optimized by reducing the latency of feedback loops caused by data dependencies. Area is further reduced by fitting more work to processors in non-critical paths and thus less number of processors are needed. The parallel slice decoders leverages slice-level parallelism and results show that 47.9–95.6 1080p frames per second is achieved by processing 4

slices in parallel, which is sufficient for most real-time applications. Higher throughput is achievable by processing more slices in parallel. The proposed decoders support 4K and 8K video resolution without any design changes.

Table 6.2: Comparison of Software Video Decoder Implementations

Design	Standard	Platform	Tech. (nm)	Freq. (GHz)	Area (mm^2)	Throughput‡ (Mpixels/s) 4:2:0	4:2:2	4:4:4	Throughput/Area‡ (Mpps/mm^2) 4:2:0	4:2:2	4:4:4	Energy/Pixel‡ (nJ/pixel) 4:2:0	4:2:2	4:4:4
ASICON'13 [85]	H.264	24-Core Proc.	65	0.80	7.29*	76.95			10.56			6.34		
TCSVT'12 [17]	HEVC	i7-3720QM	22	2.60	NA	58.38			NA			503.4§		
TCSVT'16 [86]	HEVC	i7 E5-1650	32	3.40	NA	329.7			NA			197.2§		
ICIP'20 [87]	VVC	i9-9980HK	14	2.40	NA	215.8			NA			201.1§		
VESA C Model [22]	DSC	i7-7700HQ	14	2.81	26.94†	1.67	1.25	0.83	0.06	0.05	0.03	17,412‖	22,230‖	34,273‖
Slice Decoder	DSC	KiloCore	32	1.75	5.17	49.21	49.68	24.90	**9.52**	**9.62**	**4.82**	**25.8**	**28.9**	**51.0**
Parallel-2-Slice Decoder	DSC	KiloCore	32	1.75	10.44	98.35	99.47	49.78	9.42	9.52	4.77	26.0	29.2	51.5
Parallel-4-Slice Decoder	DSC	KiloCore	32	1.75	20.73	**196.27**	**198.27**	**99.30**	9.47	9.57	4.79	26.1	29.3	51.6

* Area is scaled to 32 nm using data from Holt [43].

† The core area is estimated using a publicly available die photo and scaled to to 32 nm using data from Holt [43].

‡ Data are scaled to 32 nm using data from Holt [43].

§ Energy data are unavailable; assuming device power is half of thermal design power (TDP) [88].

‖ Energy data are estimated using Intel Power Gadget 3.5 [84].

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This dissertation discusses the hardware design and many-core software implementations of Display Stream Compression (DSC) codecs. The main contributions of this dissertation are as follows.

The area-efficient, high-performance encoder architectures are presented. The slice encoder architecture supports one slice per line and achieves an average throughput of one pixel per cycle. The slice-interleaved encoder architecture shares combinational logic and replicates only registers in the slice encoder to support multiple slices per line in which only one slice is processed at a time, resulting in the same throughput per cycle as the slice encoder. The time-interleaved architecture is capable of processing up to three slices in parallel with shared computation resources and optionally processes more slices serially using the slice-interleaved encoding scheme.

A DSC encoder chip that supports 4K video resolution is designed to evaluate the time-interleaved encoding architecture and is fabricated in TSMC 28 nm CMOS technology. The encoder supports a throughput of one pixel per cycle in one slice per line, and two pixels per cycle in two and four slices per line. It achieves up to 1448 Mpixels/s at 1.15 V, which means 174 frames per second in 4K resolution. The minimum energy per pixel of 84 pJ is demonstrated at 0.8 V. The time-interleaved encoding at 0.91 V achieves the same throughput as non-interleaved encoding at the nominal voltage, while reducing energy per pixel by 1.87–1.94 times. Furthermore, 1.38–1.54 times energy reduction is demonstrated by doubling the number of parallel encoder instances.

The hardware architectures of DSC decoders are presented, which includes: 1) the slice decoder architecture that decodes one slice per line at a throughput of three pixels per cycle, 2) the slice-interleaved decoder architecture that shares combinational logic to process multiple slices per line serially, 3) the parallel slice decoder architecture that uses dedicated slice decoders to process different columns of slices, and 4) the parallel-interleaved decoder architecture that explores the trade off between area and throughput while maintaining the capability for multiple slices per line. Six decoders based on the proposed architectures are implemented in a 28 nm standard cell library and laid out using an automatic placement and routing tool. The designs utilize 169.6–627 K logic gates and achieve maximum throughput of 1.49–13.04 Gpixels/s while dissipating 22.9–58.6 pJ/pixel at 1.0 V.

The task-level parallelism of the DSC decoding algorithm is exploited on a fine-grained many-core processor array. The many-core software implementation of the slice decoder utilizes 88 processors and 2 independent memory modules. A minimum energy per 4:2:0 pixel of 11.8 nJ is demonstrated at the supply voltage of 0.76 V. Slice-level parallelism is applied to achieve higher performance by processing multiple slices in the same row in parallel using a modified version of the slice decoder for each slice. The many-core software decoder that processes four slices in parallel is mapped to 359 processors and 6 memory modules. At 1.1 V, it is capable of decoding 1080p video at 47.9–95.6 frames per second while dissipating 26.1–51.6 nJ per pixel, which is up to 159 times higher throughput and 769 times lower energy per pixel than a DSC decoder on an Intel i7-7700HQ processor. This demonstrates the benefits of the many-core design approach over general purpose processors.

The ASIC implementations of DSC decoders achieve significant better performance and energy efficiency than the many-core software DSC decoders. With technology scaling considered [43], the ASIC decoders achieve 71–72 times, 66 times, and 64 times higher throughput while dissipating 779–939 times, 796–977 times, and 1001–1049 times lower energy per pixel for the slice decoder, parallel-2-slice decoder, and parallel-4-slice decoder, respectively.

7.2 Future Work

There are several topics on the DSC algorithm and codec design that can be investigated in future research.

7.2.1 DSC Algorithm Optimization

The data dependency on inverse quantized residuals of previous pixels within the current group for MMAP is one of the factors that limits the encoder throughput. This data dependency can be possibly removed by using the inverse quantized residuals of the previous group with consideration on the change of quantization level. As such, the three pixels in the group can be predicted in parallel and the hardware implementation of the encoder can process one group per cycle. The effect of this change to the compression quality needs to be evaluated.

Another factor to consider is increasing the group size. The DSC decoders can achieve a throughput of one group per cycle; therefore, a larger group size may lead to higher throughput. However modifying the group size will incur changes to most of the modules in the DSC algorithm.

7.2.2 DSC Codec on a Single Chip

Some systems may need to encode and decode video data in different times. Therefore, a DSC codec on the same die with shared resources between the encoding and decoding modes results in significantly lower hardware cost than using separate encoder and decoder chips. The codec can use a control signal to switch between encoding and decoding mode. The line buffer and rate buffer can be used in both encoding and decoding modes. The input buffer in the encoding mode can be used to buffer the output pixels in decoding mode.

7.2.3 DSC Encoders for Fine-Grained Many-Core Processor Arrays

This dissertation has demonstrated that fine-grained many-core processor arrays can achieve high energy efficiency and performance for DSC decoders. It is interesting to also design DSC encoders on such many-core processor arrays. The many-core design of DSC encoders is expected to be more complicated than the decoders. The many-core software encoders may reuse some modules from the many-core decoder, such as the rate control, line buffer, ICH update, prediction, and reconstruction. On the other hand, there are several design challenges that need to be addressed for throughput optimizations, such as: 1) the data dependency in MMAP between pixels of the same group; 2) efficient search for the best ICH entries and coding mode decision; 3) flatness determination, 4) slice-level parallelism for higher performance.

Glossary

ASIC *Application-Specific Integrated Circuit.* An integrated circuit (IC) chip customized for a particular use, rather than intended for general-purpose use.

CBR *Constant Bit Rate Mode.* A rate control scheme in the DSC standard in which the compressed bit rate is equal to a specified value.

CMOS *Complementary Metal–Oxide–Semiconductor.* CMOS is the most commonly used technology in the design and fabrication of integrated circuits.

CRC *Cyclic Redundancy Check.* An error-detecting code that can be used to detect accidental changes to raw data and correct errors in the data.

CTS *Clock Tree Synthesis.* A process in the physical design flow which makes sure that the clock gets distributed evenly to all sequential elements in a design.

DDR *Double Data Rate.* A operation mode that transfers data on both the rising edge and falling edges of the clock signal.

DP *DisplayPort.* A digital display interface standardized by VESA. It is primarily used to connect a video source to a display device. It can also carry audio, USB, and other forms of data.

DRC *Design Rule Check.* DRC check verifies whether the circuit layout satisfies the design rules, i.e., geometric constraints.

DSC *Display Stream Compression.* A VESA-developed low-latency compression algorithm to overcome the limitations posed by sending high-resolution video over physical media of limited bandwidth.

DSI *Display Serial Interface.* A specification by the MIPI Alliance for reducing the cost of display controllers in mobile devices.

DSU-VLC *Delta Size Unit-Variable Length Coding.* An entropy encoding method used in DSC to encode quantized residuals.

eDP *Embedded DisplayPort.* A display panel interface standard based on the DisplayPort standard. It is used for portable and embedded devices and defines the signaling interface between graphics cards and integrated displays.

FIFO *First-In First-Out.* A buffer structure in which the data enter first are read out first.

FPGA *Field Programmable Gate Array.* A integrated circuit that is reprogrammable after fabrication by users to achieve specific functionality.

Full HD *Full High Definition.* A display resolution of 1920×1080 pixels; it is also known as 1080p.

GDSII A database format for the physical layout of integrated circuits.

Group In DSC, a group is three consecutive pixels in the same slice line.

H.264/AVC *Advanced Video Coding (AVC).* A video coding standard developed jointly by ITU-T VCEG and ISO/IEC MPEG. It is also known as MPEG-4 Part 10.

HDMI *High-Definition Multimedia Interface.* An interface for transmitting uncompressed video data and compressed or uncompressed digital audio data.

HDR *High Dynamic Range.* A technology that can represent more colors, brighter highlights and darker shadows in videos and images.

HEVC *High Efficiency Video Coding.* A video coding standard developed jointly by ITU-T VCEG and ISO/IEC MPEG. HEVC is also known as H.265 and MPEG-H Part 2, and it is the successor to H.264/AVC.

ICH *Indexed Color History.* ICH is part of DSC and stores recently used pixels.

ICH-mode *ICH Coding Mode.* A coding mode in the DSC algorithm that codes each pixel using a 5-bit index.

ITU *International Telecommunication Union.* A specialized agency of the United Nations responsible for all matters related to information and communication technologies.

ITU-T *ITU Telecommunication Standardization Sector.* ITU-T develops standards for the global infrastructure of information and communication technologies.

JPEG *Joint Photographic Experts Group.* A committee that created and maintains the JPEG, JPEG 2000, and JPEG XR digital image standards.

KiloCore A fabricated chip containing 1,000 independent processors and 12 independent memory modules.

LFSR *Linear-Feedback Shift Register.* A shift register using a linear function (commonly exclusive-or operation) of its previous state as the input bit.

LSB *Least Significant Bit.* The rightmost bit in a binary number.

LVS *Layout Versus Schematic.* LVS check verifies whether the layout matches the gate-level netlist after placement and routing.

MIMD *Multiple Instruction Multiple Data.* A parallel computer architecture where different instructions with different data can be executed at the same time.

MIPI *Mobile Industry Processor Interface.* MIPI Aliance is a collaborative global organization that develops interface specifications for mobile and mobile-influenced products.

MPEG *Moving Picture Experts Group.* A group of the ISO/IEC that developed standards for coding digital audio, video, 3D graphics and genomic data.

MSB *Most Significant Bit.* The leftmost bit in a binary number.

Mux word A syntax element is the bits used to code one component of a group.

OLED *Organic Light-Emitting Diode (LED).* A LED in which the emissive electroluminescent layer is a film of organic compound that emits light in response to an electric current.

PCB *Printed Circuit Board.* A building block of electronic devices that is used to mechanically support and electrically connect electronic components.

PCIe *Peripheral Component Interconnect Express.* A high-speed serial computer expansion bus standard primarily used as an interface for motherboard components.

P-mode *Predictive Coding Mode.* A coding mode of DSC that uses prediction, quantization, and reconstruction to process pixels and code the quantized residuals in the bitstream.

Picture In the DSC standard, a picture is defined as a frame for progressive format videos or a field for interlaced format videos.

PPS *Picture Parameter Set.* PPS has a size of 128 bytes and stores the configuration parameters for DSC encoders and decoders, both of which should use identical PPS.

RGB A color model which includes three primary color components: red, green, and blue.

RISC *Reduced Instruction Set Computer.* A type of microprocessor architecture utilizing a small, highly-optimized set of instructions.

RTL *Register-Transfer Level.* A design abstraction that models the circuits in terms of registers and combination logic.

SIMD *Single Instruction Multiple Data.* A parallel computer architecture where different data can be executed in parallel using the same instruction.

Slice In DSC, slices are independently decodeable, identically sized rectangular regions of a picture. A picture can be configured and partitioned to one or more rows and columns of slices.

Slices per line The number of columns of slices per picture.

SPEF *Standard Parasitic Exchange Format.* An IEEE standard for representing parasitic data of wires in a chip. SPEF data are extracted after placement and routing.

SRAM *Static Random-Access Memory.* A type of volatile random-access memory which stores data only when power is supplied.

Supergroup A set of four consecutive groups in a slice in the DSC standard.

Syntax element A syntax element is the bits used to code one component of a group.

UHD *Ultra-High-Definition.* Digital video formats with aspect ratio of 16:9, including 4K UHD (3840×2160) and 8K UHD (7680×4320).

VBR *Variable Bit Rate Mode.* A rate control scheme in the DSC standard which allows the compressed bit rate to be lower than the specified value.

VCD *Value Change Dump.* An file that contains header information, as well as definition and value changes of variables in a design.

VCEG *Video Coding Experts Group.* A working group of the ITU-T which developed the "H.26x" video coding standards.

VDC-M *VESA Display Compression-M.* A video coding standard developed by VESA targeting low-latency, visually lossless video compression over display links. VDC-M can achieve higher compression ratio than DSC with higher implementation cost.

VESA *Video Electronics Standards Association.* A technical standards organization for computer display standards.

VLC *Variable Length Coding.* A coding scheme that uses a variable number of bits to code the source information.

Vt *Threshold Voltage.* The minimum gate to source voltage to create a conducting path between the source and drain terminals of a field-effect transistor.

VVC *Versatile Video Coding.* A video coding standard also known as H.266 and MPEG-I Part 3; It is the successor to HEVC.

WCG *Wide Color Gamut.* A much wider palette of colors that can be displayed on the screen.

YCbCr A color space that includes a luma component (Y), a blue-difference chroma component (Cb), and a red-difference chroma component (Cr).

Bibliography

[1] *Video Codec for Audiovisual Services at px64 kbit/s.* ITU-T Rec. H.261, version 1: Nov. 1990, version 2: Mar. 1993.

[2] *Video Coding for Low Bit Rate Communication.* ITU-T Rec. H.263, Nov. 1995 (and subsequent editions).

[3] *Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s—Part 2: Video.* ISO/IEC 11172-2 (MPEG-1), ISO/IEC JTC 1, 1993.

[4] *Coding of Audio-Visual Objects—Part 2: Visual.* ISO/IEC 14496-2 (MPEG-4 Visual version 1), ISO/IEC JTC 1, Apr. 1999 (and subsequent editions).

[5] ITU-T and ISO/IEC JTC 1, *Generic Coding of Moving Pictures and Associated Audio Information—Part 2: Video.* ITU-T Rec. H.262 and ISO/IEC 13818-2 (MPEG-2 Video), ITU-T and ISO/IEC JTC 1, Nov. 1994.

[6] ——, *Advanced Video Coding for generic audiovisual services.* Rec. ITU-T H.264 and ISO/IEC 14496-10 (AVC), May 2003 (and subsequent editions).

[7] ——, *High Efficiency Video Coding.* Rec. ITU-T H.265 and ISO/IEC 23008-2 (HEVC), Apr. 2013 (and subsequent editions).

[8] ——, *Versatile Video Coding.* Rec. ITU-T H.266 and ISO/IEC 23090-3 (VVC), Jul. 2020 (and subsequent editions).

[9] J. Bankoski, P. Wilkins, and Y. Xu, "Technical overview of VP8, an open source video codec for the web," in *Proc. IEEE International Conference on Multimedia and Expo (ICME)*, Jul. 2011, pp. 1–6.

[10] D. Mukherjee, J. Bankoski, A. Grange, J. Han, J. Koleszar, P. Wilkins, Y. Xu, and R. Bultje, "The latest open-source video codec VP9 - an overview and preliminary results," in *Proc. Picture Coding Symposium (PCS)*, Dec. 2013, pp. 390–393.

[11] Y. Chen, D. Murherjee, J. Han, A. Grange, Y. Xu, Z. Liu, S. Parker, C. Chen, H. Su, U. Joshi, C. Chiang, Y. Wang, P. Wilkins, J. Bankoski, L. Trudeau, N. Egge, J. Valin, T. Davies, S. Midtskogen, A. Norkin, and P. de Rivaz, "An overview of core coding tools in the AV1 video codec," in *Proc. Picture Coding Symposium (PCS)*, Jun. 2018, pp. 41–45.

[12] A. Descampe, T. Richter, T. Ebrahimi, S. Foessel, J. Keinert, T. Bruylants, P. Pellegrin, C. Buysschaert, and G. Rouvroy, "JPEG XS–a new standard for visually lossless low-latency lightweight image coding," *Proceedings of the IEEE*, vol. 109, no. 9, pp. 1559–1577, Sep. 2021.

[13] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the H.264/AVC standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 9, pp. 1103–1120, Sep. 2007.

[14] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, Jul. 2003.

[15] G. J. Sullivan, J. Ohm, W. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649–1668, Dec. 2012.

[16] J. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand, "Comparison of the coding efficiency of video coding standards—including high efficiency video coding (HEVC)," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1669–1684, Dec. 2012.

[17] F. Bossen, B. Bross, K. Suhring, and D. Flynn, "HEVC complexity and implementation analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1685–1696, Dec. 2012.

[18] B. Bross, Y. Wang, Y. Ye, S. Liu, J. Chen, G. J. Sullivan, and J. Ohm, "Overview of the versatile video coding (VVC) standard and its applications," *IEEE Transactions on Circuits and Systems for Video Technology*, 2021.

[19] J. Han, B. Li, D. Mukherjee, C. Chiang, A. Grange, C. Chen, H. Su, S. Parker, S. Deng, U. Joshi, Y. Chen, Y. Wang, P. Wilkins, Y. Xu, and J. Bankoski, "A technical overview of AV1," *Proceedings of the IEEE*, vol. 109, no. 9, pp. 1435–1462, Sep. 2021.

[20] "JPEG XS, a new standard for visually lossless low-latency lightweight image coding system," White Paper, Joint Photographic Experts Group (JPEG), 2019. [Online]. Available: www.jpeg.org

[21] F. G. Walls and A. S. MacInnis, "VESA display stream compression: An overview," in *SID Symposium Digest of Technical Papers*, vol. 45, no. 1, 2014, pp. 360–363.

[22] Video Electronics Standards Association, *VESA Display Stream Compression (DSC) Standard Version 1.2a*, Jan. 2017. [Online]. Available: http://vesa.org

[23] R. S. Allison, L. M. Wilcox, W. Wang, D. M. Hoffman, Y. Hou, J. Goel, L. Deas, and D. Stolitzka, "Large scale subjective evaluation of display stream compression," in *SID Symposium Digest of Technical Papers*, vol. 48, no. 1, 2017, pp. 1101–1104.

[24] A. Sudhama, M. D. Cutone, Y. Hou, J. Goel, D. Stolitzka, N. Jacobson, R. S. Allison, and L. M. Wilcox, "Visually lossless compression of high dynamic range images: A large-scale evaluation," in *SID Symposium Digest of Technical Papers*, vol. 49, no. 1, 2018, pp. 1151–1154.

[25] F. G. Walls and A. S. MacInnis, "VESA display stream compression for television and cinema applications," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 4, pp. 460–470, Dec. 2016.

[26] J. Xu, R. Joshi, and R. A. Cohen, "Overview of the emerging HEVC screen content coding extension," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 26, no. 1, pp. 50–62, Jan. 2016.

[27] W. Peng, F. G. Walls, R. A. Cohen, J. Xu, J. Ostermann, A. MacInnis, and T. Lin, "Overview of screen content video coding: Technologies, standards, and beyond," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 4, pp. 393–408, Dec. 2016.

[28] X. Xu and S. Liu, "Overview of screen content coding in recently developed video coding standards," *IEEE Transactions on Circuits and Systems for Video Technology*, 2021.

[29] "Cisco annual internet report (2018–2023)," White Paper, Cisco, Mar. 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[30] Video Electronics Standards Association. (2021) VESA display compression codecs. [Online]. Available: https://vesa.org/vesa-display-compression-codecs/

[31] Hardent Inc. VESA DSC IP cores. [Online]. Available: https://www.hardent.com/ip-products-vesa-dsc/

[32] S. W. Kim, S. Park, J. Jun, and Y. Han, "Design and implementation of display stream compression decoder with line buffer optimization," *IEEE Transactions on Consumer Electronics*, vol. 65, no. 3, pp. 322–328, Aug. 2019.

[33] H. S. Malvar, G. J. Sullivan, and S. Srinivasan, "Lifting-based reversible color transformations for image compression," in *Applications of Digital Image Processing XXXI*, vol. 7073, 2008, pp. 707 307.

[34] S. Wu, S. Gutgutia, M. Alioto, and B. Baas, "Display stream compression encoder architectures for real-time 4K and 8K video encoding," in *Proc. IEEE Asilomar Conference on Signals, Systems, and Computers (ACSSC)*, Oct. 2018, pp. 251–255.

[35] S. Wu, K. De Silva, S. Gutgutia, B. Baas, and M. Alioto, "A 1448-Mpixel/s, 84-pJ/pixel display stream compression encoder in 28 nm for 4K video resolution," in *Proc. IEEE Asian Solid-State Circuits Conference (A-SSCC)*, Nov. 2021, In press.

[36] G. Frank, "Pulse code communication," Mar. 17 1953, US Patent 2,632,058.

[37] S. W. Golomb, *Shift Register Sequences: Secure and Limited-Access Code Generators, Efficiency Code Generators, Prescribed Property Generators, Mathematical Models*, third revised ed. World Scientific, 2017.

[38] D. Zhou, G. He, W. Fei, Z. Chen, J. Zhou, and S. Goto, "A 4320p 60fps H.264/AVC intra-frame encoder chip with 1.41Gbins/s CABAC," in *Proc. IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, Jun. 2012, pp. 154–155.

[39] S. Tsai, C. Li, H. Chen, P. Tsung, K. Chen, and L. Chen, "A 1062Mpixels/s 8192×4320p high efficiency video coding (H.265) encoder chip," in *Proc. IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, Jun. 2013, pp. C188–C189.

[40] C. Ju, T. Liu, K. Lee, Y. Chang, H. Chou, C. Wang, T. Wu, H. Lin, Y. Huang, C. Cheng, T. Lin, C. Chen, Y. Lin, M. Chiu, W. Li, S. Wang, Y. Lai, P. Chao, C. Chien, M. Hu, P. Wang, Y. Huang, S. Chuang, L. Chen, H. Lin, M. Wu, and C. Chen, "A 0.5 nJ/pixel 4 K H.265/HEVC codec LSI for multi-format smartphone applications," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 56–67, Jan. 2016.

[41] S. Wu and B. M. Baas, "A low-cost slice interleaving DSC decoder architecture for real-time 8K video decoding," in *Proc. IEEE International Midwest Symposium on Circuits and Systems (MWCAS)*, Aug. 2018, pp. 364–367.

[42] A. Stillmaker, Z. Xiao, and B. Baas, "Toward more accurate scaling estimates of CMOS circuits from 180 nm to 22 nm," VLSI Computation Lab, ECE Department, University of California, Davis, Tech. Rep. ECE-VCL-2011-4, Dec. 2011, http://vcl.ece.ucdavis.edu/pubs/2011.12.techreport.techscale/.

[43] W. M. Holt, "Moore's law: A path going forward," in *Proc. IEEE International Solid-State Circuits Conference - Digest of Technical Papers (ISSCC)*, Feb. 2016, pp. 8–13.

[44] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm," *Integration, the VLSI Journal*, vol. 58, pp. 74–81, Jun. 2017.

[45] S. Sarangi and B. Baas, "DeepScaleTool: A tool for the accurate estimation of technology scaling in the deep-submicron era," in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2021, pp. 1–5.

[46] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar./Apr. 2008.

[47] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, Mar./Apr. 2011.

[48] "NVIDIA GeForce GTX 680," White Paper, NVIDIA, 2012. [Online]. Available: https://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

[49] "NVIDIA GeForce GTX 980," White Paper, NVIDIA, 2014. [Online]. Available: https://www.microway.com/download/whitepaper/NVIDIA_Maxwell_GM204_Architecture_Whitepaper.pdf

[50] "NVIDIA Tesla P100," White Paper, NVIDIA, 2016. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[51] "NVIDIA Tesla V100 GPU architecture," White Paper, NVIDIA, 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[52] "NVIDIA Ampere GA102 GPU architecture," White Paper, NVIDIA, 2020. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[53] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64 - processor: A 64-core SoC with mesh interconnect," in *Proc. IEEE International Solid-State Circuits Conference - Digest of Technical Papers (ISSCC)*, Feb. 2008, pp. 88–598.

[54] EZchip Semiconductor. TILE-Gx72 Processor Product Brief. [Online]. Available: https://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx72.pdf

[55] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, "An 80-tile sub-100-W TeraFLOPS processor in 65-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, Jan. 2008.

[56] M. Butts, "Synchronization through communication in a massively parallel processor array," *IEEE Micro*, vol. 27, no. 5, pp. 32–40, Sep./Oct. 2007.

[57] A. Olofsson, "Epiphany-V: A 1024 processor 64-bit RISC system-on-chip," *arXiv preprint arXiv:1610.01832*, 2016.

[58] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. M. Baas, "An asynchronous array of simple processors for DSP applications," in *Proc. IEEE International Solid-State Circuits Conference - Digest of Technical Papers (ISSCC)*, Feb. 2006, pp. 1696–1705.

[59] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, D. Truong, T. Mohsenin, and B. Baas, "AsAP: An asynchronous array of simple processors," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 3, pp. 695–705, Mar. 2008.

[60] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas, "A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling," in *Proc. IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, Jun. 2008, pp. 22–23.

[61] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas, "A 167-processor computational platform in 65 nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 4, pp. 1130–1144, Apr. 2009.

[62] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "KiloCore: A 32-nm 1000-processor computational array," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 891–902, Apr. 2017.

[63] J. Pimentel, "Methods for reducing floating-point computation overhead," Ph.D. dissertation, University of California, Davis, CA, USA, Aug. 2017. [Online]. Available: http://vcl.ece.ucdavis.edu/pubs/theses/2017-2.pimentel/

[64] P. Shi, "Sparse matrix multiplication on a many-core platform," Master's thesis, University of California, Davis, CA, USA, Dec. 2018. [Online]. Available: http://vcl.ece.ucdavis.edu/pubs/theses/2018-1.pshi/

[65] Z. Zhao, "Matrix inversion on a many-core platform," Master's thesis, University of California, Davis, CA, USA, Mar. 2021. [Online]. Available: http://vcl.ece.ucdavis.edu/pubs/theses/2021-1.zzhao/

[66] F. Borges, "AlexNet deep neural network on a many core platform," Master's thesis, University of California, Davis, CA, USA, Sep. 2019. [Online]. Available: http://vcl.ece.ucdavis.edu/pubs/theses/2019-1.fborges/

[67] A. Hlaing, "Long short-term memory on a many-core platform," Master's thesis, University of California, Davis, CA, USA, Mar. 2020. [Online]. Available: http://vcl.ece.ucdavis.edu/pubs/theses/2020-2.ahlaing/

[68] S. Sarangi and B. Baas, "Canonical huffman decoder on fine-grain many-core processor arrays," in *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2021, pp. 512–517.

[69] Z. Xiao and B. M. Baas, "A 1080p H.264/AVC baseline residual encoder for a fine-grained many-core system," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 7, pp. 890–902, Jul. 2011.

[70] Z. Xiao, S. Le, and B. Baas, "A fine-grained parallel implementation of a H.264/AVC encoder on a 167-processor computational platform," in *Proc. IEEE Asilomar Conference on Signals, Systems, and Computers (ACSSC)*, Nov. 2011, pp. 2067–2071.

[71] S. Kulkarni, "Implementation of context-based adaptive binary arithmetic coding on KiloCore processor arrays," Master's thesis, University of California, Davis, CA, USA, Mar. 2021. [Online]. Available: http://vcl.ece.ucdavis.edu/pubs/theses/2021-2.skulkarni/

[72] B. Liu and B. M. Baas, "Parallel AES encryption engines for many-core processor arrays," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 536–547, Mar. 2013.

[73] E. O. Adeagbo, "Energy-efficient pattern matching methods on a fine-grained many-core platform," Master's thesis, University of California, Davis, CA, USA, Mar. 2017. [Online]. Available: http://vcl.ece.ucdavis.edu/pubs/theses/2017-1.Adeagbo/

[74] A. Stillmaker, B. Bohnenstiehl, L. Stillmaker, and B. Baas, "Scalable energy-efficient parallel sorting on a fine-grained many-core processor array," *Journal of Parallel and Distributed Computing*, vol. 138, pp. 32–47, Apr. 2020.

[75] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "A 5.8 pJ/Op 115 billion Ops/sec, to 1.78 trillion Ops/sec 32 nm 1000-processor array," in *Proc. IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, Jun. 2016, pp. 1–2.

[76] S. Wu and B. M. Baas, "Display stream compression decoders for fine-grained many-core processor arrays," *IEEE Transactions on Circuits and Systems—Part II: Express Briefs*, vol. 68, no. 5, pp. 1730–1734, May 2021.

[77] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "KiloCore: A 32 nm 1000-processor array," in *Proc. IEEE HotChips Symposium on High-Performance Chips (HotChips)*, Aug. 2016, pp. 1–23.

[78] A. Stillmaker, B. Bohnenstiehl, and B. Baas, "The design of the KiloCore chip," in *Proc. ACM/IEEE Design Automation Conference (DAC)*, Austin, TX, Jun. 2017.

[79] R. Apperson, Z. Yu, M. Meeuwsen, T. Mohsenin, and B. Baas, "A scalable dual-clock FIFO for data transfers between arbitrary and haltable clock domains," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 10, pp. 1125–1134, Oct. 2007.

[80] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "KiloCore: A fine-grained 1,000-processor array for task-parallel applications," *IEEE Micro*, vol. 37, no. 2, pp. 63–69, Mar./Apr. 2017.

[81] B. Bohnenstiehl, "Design and programming of the KiloCore processor arrays," Ph.D. dissertation, University of California, Davis, CA, USA, Mar. 2020. [Online]. Available: http://vcl.ece.ucdavis.edu/pubs/theses/2020-1.bbohnenstiehl/

[82] M. Hildebrand, “Mapper2,” Github, 2020. [Online]. Available: https://github.com/hildebrandmw/Mapper2.jl

[83] S. Wu and B. M. Baas, “Indexed color history many-core engines for display stream compression decoders,” in *Proc. IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Nov. 2020, pp. 1–4.

[84] T. McKay and P. C. Konsor. (2019) Intel Power Gadget. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-power-gadget.html

[85] S. Zhu, Z. Yu, S. Cui, Z. Yu, and X. Zeng, “H.264 video parallel decoder on a 24-core processor,” in *Proc. IEEE 10th International Conference on ASIC*, Oct. 2013, pp. 1–4.

[86] W. Hamidouche, M. Raulet, and O. Déforges, “4K real-time and parallel software video decoder for multilayer HEVC extensions,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 26, no. 1, pp. 169–180, Jan. 2016.

[87] A. Wieckowski, G. Hege, C. Bartnik, C. Lehmann, C. Stoffers, B. Bross, and D. Marpe, “Towards a live software decoder implementation for the upcoming versatile video coding (VVC) codec,” in *Proc. IEEE International Conference on Image Processing (ICIP)*, Oct. 2020, pp. 3124–3128.

[88] M. Butler, “‘Bulldozer’ a new approach to multithreaded compute performance,” in *Proc. IEEE HotChips Symposium on High-Performance Chips (HotChips)*, Aug. 2010, pp. 1–17.