JPEG-2000 Decoder Documentation

Introduction

This documentation covers our implementation of the JPEG-2000 decoder. We reference our code as well as **ISO 15444-1** (sometimes listed simply as “the ISO”) and **BPJ2K** to provide the most important details about the decoder design and implementation. Whenever we use a figure or table from these documents, we use the **ISO** or **BPJ2K** prefix on the label. All discussion in this documentation assumes you have either read the original or future project specification handed out to senior project groups at Rose-Hulman. If you have not read it, the following two sections contain a modified copy of the project overview as approved for future projects.

Project Introduction

At the Johns Hopkins University - Applied Physics Lab (JHU/APL), we routinely work with science imagery from the National Aeronautics and Space Administration (NASA). As our scientists develop algorithms and pipelines to process the data, we need efficient tools to read the source imagery.

Some of the NASA missions, such as the Solar Dynamics Observatory (SDO) and Lunar Topography, produce data formatted to the JPEG-2000 image compression standard. Since JPEG-2000 has never been a widely adopted standard and is generally used only by science communities, the tooling for, and optimization of, open source image decoders is limited. To support working with this file format we seek an optimized image reading capability.

As improvements to science instrumentation and technologies allow for higher resolution astronomical imaging, images will inevitably increase in size in the future. It is important to understand the impact of this on the current file format scheme. To address this, we are seeking a benchmark suite to characterize and stress test the image-reading capability developed.

Project Description

The primary goal of this project would be to continue the work started by the 2023-2024 JPEG-2000 Senior Project Group and finish the implementation of a C/C++ library to read JPEG-2000 images. A partial implementation of the decoding pipeline has been created and tested in C, but still requires additional work on decompressing the binary data.

The decoder is focusing on a subset of the **ISO 15444-1** standard and needs to decode the NPJE specification of the JPEG-2000 format with possible support for the EPJE specification. Three image reading schemes need to be supported. The first is image tiling with overlap, the second is random chipping throughout the image, and the third is a low resolution preview of the entire image. The goal is to optimize the speed of the first two schemes and support the third for human verification. The goal of the project is to find clever ways to optimize the tile and chip read speed, such as by minimizing duplicate reads or finding optimal read orders.

The final decoder will need to be benchmarked against other available implementations of the standard. The effectiveness of the decoder will be based on metrics such as speed and throughput depending on the image size (including 10s of GBs), compression profile, location and type of image read, and the contents of the image. The format is designed to be

parallelizable and effective implementations will look for ways to maximize independence among compute cores to maximize performance. Ideally, after CPU performance is optimized GPU performance will be investigated.

Design Overview

The diagram below represents our overall decoder design. The six stages you see are described in detail throughout this documentation and reference important tables and figures from other resources that we have found. Note that these stages do not necessarily correspond one-to-one with the documentation and instead split and/or merge certain sections discussed in **ISO 15444-1** based on our initial understanding of the design and breakdowns found in textbooks and similar sources.

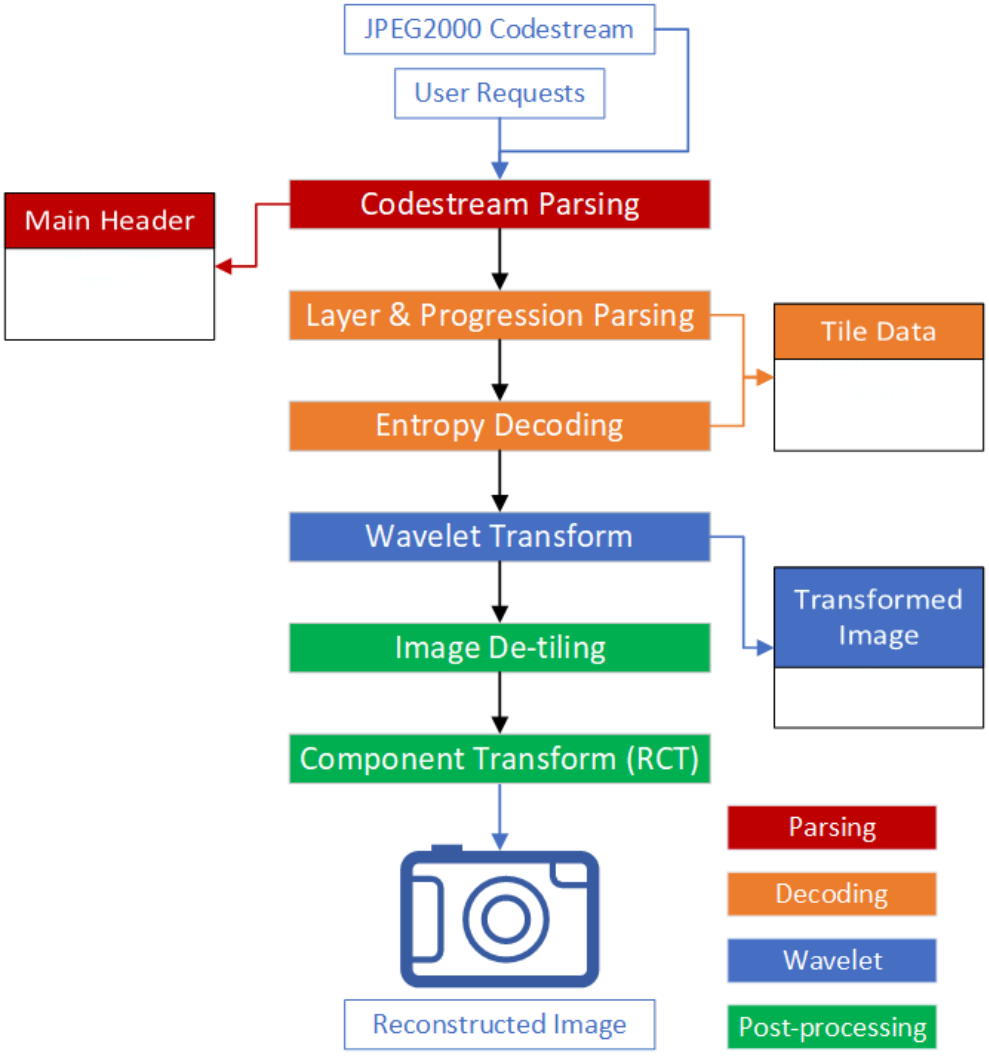


Figure 1, JPEG-2000 General Design Diagram

Codestream Parsing

The codestream parsing stage deals with identifying and gathering the marker segments associated with the information used to encode the file. These segments are found in the main header to discuss properties about the whole file, and they are found in tile headers to discuss properties specific to an image tile.

Layer & Progress Parsing

This stage uses information from the marker segments to identify the profile (e.g. NPJE or EPJE) and properly parse the tile and tile-part data in the file based on the profile. The packet data containing the compressed and encoded image data can then be retrieved from the parsed tiles (and tile-parts if used in the profile).

Entropy Decoding

This stage takes in the packets from each tile containing compressed image data and extracts code blocks that contain wavelet coefficients for image transformation. We initially identified this as a single stage; however, we now believe that this stage should be split up into multiple stages as in **ISO 15444-1**. We discuss this in more detail in the Entropy Decoder section breakdown.

Wavelet Transformation

This stage takes in the identified wavelet coefficients from each region in a tile and combines them into new wavelet coefficients. This is accomplished by recreating the wave specified by the coefficients and combining them into a single new signal and recording the new wavelet coefficients. The final combined wavelet coefficients for each tile and color spectrum can be used to reconstruct the actual image data.

Image De-Tiling

At this stage we have retrieved the original image data contained in each individual tile or region in each tile. We recombine and rearrange the matrices of image data back into a single larger image.

Reversible Component Transform

At this stage we have the partially or fully reconstructed image data for each of the image components (color channels). We combine and decompress the color channels back into a format suitable for viewing, typically RGB.

JPEG-2000 Marker Segments

The JPEG-2000 format uses a format of headers, known as marker segments, and payloads containing image data to compress an image. Before any complex decoding can be accomplished, the key marker segments must be found to identify key attributes about the encoded file. These markers are contained near the start of the file in the main header or throughout the body of the file at the start of collections of data known as tiles. **Table ISO-A-3** contains a list of each marker segment that appears in the **ISO**.

Some marker segments do not appear in the NPJE or EPJE formats, and we do not discuss those segments here. However, some segments are optional but not recommended by the **ISO**. **Table 1** contains an identification for each of these segments by our client as either necessary or unnecessary to decoder functionality based on their private satellite images. We recommend investigating marker segments identified as unnecessary only after the entire decoder is finished.

Table 1, Optional Segments

Client Identified Segment As	Marker Segment	Codestream Value
Necessary	COD	0xFF52
Unnecessary	COC	0xFF53
Unnecessary	PPT	0xFF61
Unnecessary	CRG	0xFF63
Unnecessary	SOP	0xFF91
Unnecessary	EPH	0xFF92

Table ISO-A-3 (ISO 15444-1)

Information	Marker segment
Capabilities Image area size or reference grid size (height and width) Tile size (height and width) Number of components Component precision Component mapping to the reference grid (sub-sampling)	SIZ
Tile index Tile-part data length	SOT, TLM
Progression order Number of layers Multiple component transformation used	COD
Coding style Number of decomposition levels Code-block size Code-block style Wavelet transformation Precinct size	COD, COC
Region of interest shift	RGN
No quantization Quantization derived Quantization expounded	QCD, QCC
Progression starting point Progression ending point Progression order default	POC
Error resilience	SOP
End of packet header	EPH
Packet headers	PPM, PPT
Packet lengths	PLM, PLT
Component registration Optional information	CRG COM

SIZ Marker Segment

SIZ Usage

Function: Provides information about the uncompressed image such as the width and height of the reference grid, the width and height of the tiles, the number of components, component bit depth, and the separation of component samples with respect to the reference grid (see **ISO 15444-1 B.2**).
Usage: Main header. There shall be one and only one in the main header immediately after the marker segment. There shall be only one SIZ codestream.

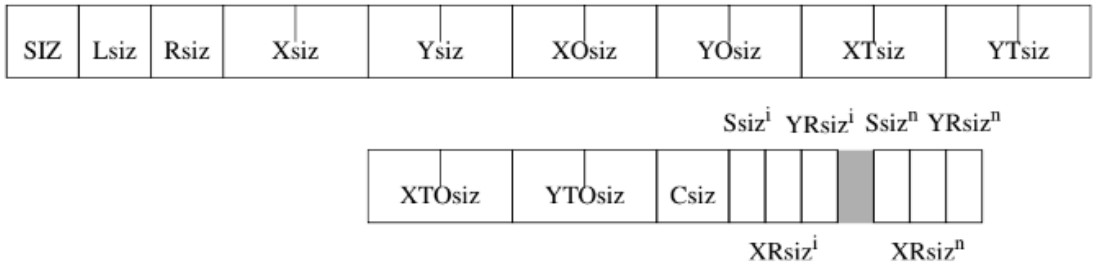


Figure ISO-A-7, Image and Tile Size Syntax (ISO 15444-1)

SIZ Fields

Table BPJ2K-D-9, SIZ Segment Breakdown (BPJ2K)

Parameter	Size (bits)	Values	NPJE	Notes
SIZ	16	0xFF51	0xFF51	Image and tile size marker segment.
Lsiz	16	41 – 49,190	38 + 3·Csiz	Length of marker segment in bytes.
Rsiz	16	see Table 7-7	0000 0000 0000 0010 0100 0000 0000 0010 (Profile-1)	see Table C-1. Other profile values from ISO/IEC 15444-1:2019 Table A.10 are not allowed.
Xsiz	32	$1 – (2^{32}-1)$	image width $(1 – (2^{31}-1))$	Width of reference grid. Equal to the image width with no image offset into the reference grid.
Ysiz	32	$1 – (2^{32}-1)$	image height $(1 – (2^{31}-1))$	Height of reference grid. Equal to the image height with no image offset into the reference grid
XOsiz	32	$0 – (2^{32}-2)$	0	Horizontal offset from the origin of the reference grid to the left side of the image area.
YOsiz	32	$0 – (2^{32}-2)$	0	Vertical offset from the origin of the reference grid to the top of the image area.
XTsiz	32	$1 – (2^{32}-1)$	1,024	Width of one reference tile with respect to the reference grid.
YTsiz	32	$1 – (2^{32}-1)$	1,024	Height of one reference tile with respect to the reference grid.
XTOsiz	32	$0 – (2^{32}-2)$	0	Horizontal offset from the origin of the reference grid to the left edge of the first tile.
YTOsiz	32	$0 — (2^{32}-2)$	0	Vertical offset from the origin of the reference grid to the top edge of the first tile.
Csiz	16	1 – 16,384	Nbands	The number of components in the image.

Parameter	Size (bits)	Values	NPJE	Notes
$Ssiz^i$	8	0000 0000 – 1010 0101	Unsigned: 0 – 31 Signed: 128 – 159	0xxx xxxx Unsigned data 1xxx xxxx signed data x000 0000 – x010 0101 bit depth of data = value + 1
$XRsiz^i$	8	1 – 255	1	Horizontal separation of a sample of the i^{th} component with respect to the reference grid.
$YRsiz^i$	8	1 – 255	1	Vertical separation of a sample of the i^{th} component with respect to the reference grid.

SIZ Implementation

Implemented in parsing.h.

JPEG2000_SIZ

Table 2, Implementation for SIZ

Lsiz	**uint16_t**	length of marker segment in bytes (not including marker), determined by the following equation: Lcod = 38 + 3 * Csiz
Rsiz	**uint16_t**	Denotes capabilities that a decoder needs to properly decode the codestream
Xsiz	**uint32_t**	width of the reference grid
Ysiz	**uint32_t**	height of the reference grid
XOsiz	**uint32_t**	horizontal offset from the origin of the reference grid to the left side of the image area
YOsiz	**uint32_t**	vertical offset from the origin of the reference grid to the top side of the image area
XTsiz	**uint32_t**	width of one reference tile w/ respect to the reference grid
YTsiz	**uint32_t**	height of one reference tile w/ respect to the reference grid
XTOsiz	**uint32_t**	horizontal offset from origin of the reference grid to the left side of first tile
YTOsiz	**uint32_t**	vertical offset from origin of the reference grid to the top side of first tile
Csiz	**uint16_t**	number of components in the image
Components	**JPEG2000_SIZ_COMPONENT**	array of components

JPEG2000_SIZ_COMPONENT

Table 3, Implementation for SIZ_COMPONENT

SSiz	**uint8_t**	precision (depth) in bits and sign of the ith component samples
XRsiz	**uint8_t**	horizontal separation of a sample of ith component with respect to the reference grid
YRsiz	**uint8_t**	vertical separation of a sample of ith component with respect to the reference grid

COD Marker Segment (Coding Style Default):

COD Usage

Function: Describes the coding style, number of decomposition levels, and layering that is the default used for compressing all components of an image (if in the main header) or a tile (if in the tile-part header). The parameter values can be overridden for an individual component by a COC marker segment in either the main or tile-part header.
Usage: Main and first tile-part header of a given tile. There is one and only one in the main header. Additionally, there may be at most one per tile. If there are multiple tile-parts in a tile, and this marker segment is present, it will only be found in the first tile-part (TPsot = 0).
When used in the main header, the COD marker segment parameter values are used for all tile-components that don't have a corresponding COC marker segment in either the main or tile-part header. When used in the tile-part header it overrides the main header COD and COCs and is used for all components in that tile without a corresponding COC marker segment in the tile-part. Thus, the order of precedence is the following:

Tile-part COC > Tile-part COD > Main COC > Main COD

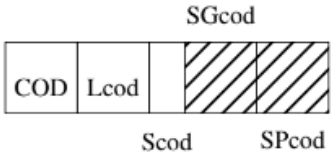


Figure ISO-A-8 – Coding Style Default Syntax (ISO 15444-1)

COD Fields

Progression Order Values

Table ISO-A-16 – Progression Order for the SPcod, SPcoc, and Ppoc Parameters (ISO 15444-1)

Values (bits) MSB LSB	Progression order
0000 0000	Layer-resolution level-component-position progression
0000 0001	Resolution level-layer-component-position progression
0000 0010	Resolution level-position-component-layer progression
0000 0011	Position-component-resolution level-layer progression
0000 0100	Component-position-resolution level-layer progression
	All other values reserved

Code Block Values

Table ISO-A-18 – Width or Height Exponent of the Code-blocks for the SPcod and SPcoc Parameters (ISO 15444-1)

Values (bits) MSB LSB	Code-block width and height
xxxx 0000 — xxxx 1000	Code-block width and height exponent offset value *xcb = value + 2 or ycb = value + 2*. The code-block width and height are limited to powers of two with the minimum size being 2^2 and the maximum being 2^{10}. Further, the code-block size is restricted so that *xcb+ycb* <= 12.
	All other values reserved

Table ISO-A-19 – Code-block Style for the SPcod and SPcoc Parameters

Values (bits) MSB LSB	Code-block style
xxxx xxx0 xxxx xxx1	No selective arithmetic coding bypass Selective arithmetic coding bypass
xxxx xx0x xxxx xx1x	No reset of context probabilities on coding pass boundaries Reset context probabilities on coding pass boundaries
xxxx x0xx xxxx x1xx	No termination on each coding pass Termination on each coding pass
xxxx 0xxx xxxx 1xxx	No vertically causal context Vertically causal context
xxx0 xxxx xxx1 xxxx	No predictable termination Predictable termination
xx0x xxxx xx1x xxxx	No segmentation symbols are used Segmentation symbols are used
	All other values reserved

Transformation Values

Table ISO-A-20 – Transformation for the SPcod and SPcoc Parameters

Values (bits) MSB LSB	Transformation type
0000 0000	9-7 irreversible filter
0000 0001	5-3 reversible filter
	All other values reserved

Precinct Values

Table ISO-A-21 – Precinct Width and Height for the SPcod and SPcoc Parameters (ISO 15444-1)

Values (bits) MSB LSB	Precinct size
xxxx 0000 — xxxx 1111	4 LSBs are the precinct width exponent, $PPx = value$. This value may only equal zero at the resolution level corresponding to the N_LLL band.
0000 xxxx — 1111 xxxx	4 MSBs are the precinct height exponent $PPy = value$. This value may only equal zero at the resolution level corresponding to the N_LLL band.

COD Implementation

Implemented in parsing.h.

Table 4, Implementation for COD

Lcod	**uint16_t**	length of marker segment in bytes (not including marker), determined by the following equation (pg 29 ISO 15444-1): $Lcod = \begin{cases} 12 & \text{maximum\_precincts} \\ 13 + \text{number\_decomposition\_levels} & \text{user-defined\_precincts} \end{cases}$
Scod	**uint8_t**	Coding style for all components
SGcod_Progression_Order	**uint8_t**	Progression order parameter for coding style designated in Scod (independent of components)
SGcod_Number_Layers	**uint8_t**	Number of layers parameter for coding style designated in Scod (independent of components)
SGcod_Multiple_component_Transform	**uint8_t**	Multiple component transform parameter (independent of components)
SPcod_Number_Decomposition_Levels	**uint8_t**	Number of decomposition levels parameter (relates to all components) value: 0 – 32
SPcod_Code_Block_Width	**uint8_t**	Width of the code block (relates to all components)
SPcod_Code_Block_Height	**uint8_t**	Height of the code block (relates to all components)
SPcod_Code_Block_Style	**uint8_t**	Style of the code block (relates to all components)
SPcod_Transformation	**uint8_t**	Transformation (relates to all components)
precincts	**uint8_t***	Array of precincts

COC Marker Segment (Coding Style Component):

COC Usage

Function: Describes the coding style, number of decomposition levels, and layering used for compressing a particular component.
Usage: Main and first tile-part header of a given tile. Optional in both the main and tile-part headers. No more than one per any given component may be present in either the main or tile-part headers.
When used in the main header it overrides the main COD marker segment for the specific component. When used in the tile-part header it overrides the main COD, main COC, and tile COD for the specific component. Thus, the order of precedence is the following:

Tile-part COC > Tile-part COD > Main COC > Main COD

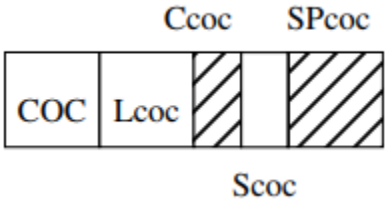


Figure ISO-A-10 – Coding Style Component Syntax (ISO 15444-1)

COC Fields

Parameter Values

The values for SPcoc are the same as SPcod which are referenced above in **Table ISO-A-15**.

Table ISO-A-22 – Coding Style Component Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
COC	16	0xFF53
Lcoc	16	9 — 43
Ccoc	8 16	0 — 255; if Csiz < 257 0 — 16 383; Csiz ≥ 257
Scoc	8	Table A-23
$SPcoc^i$	variable	Table A-15

Scoc Values

Table ISO-A-23 – Coding Style Parameter Values for the Scoc Parameter

Values (bits) MSB LSB	Coding style
0000 0000	Entropy coder with maximum precinct values PPx = PPy = 15
0000 0001	Entropy coder with precinct values defined below
	All other values reserved

COC Implementation

Implemented in parsing.h.

Table 5, Implementation for COC

Lcoc	**uint16_t**	length of marker segment in bytes (not including marker), determined by the following equation (pg 34 ISO 15444-1): $Lcoc = \begin{cases} 9 & \text{maximum\_precincts AND Csiz} < 257 \\ 10 & \text{maximum\_precincts AND Csiz} \geq 257 \\ 10 + \text{number\_decomposition\_levels} & \text{user-defined\_precincts AND Csiz} < 257 \\ 11 + \text{number\_decomposition\_levels} & \text{user-defined\_precincts AND Csiz} \geq 257 \end{cases}$
Ccoc	**uint16_t***	(MAY ONLY CONTAIN 8 BIT VALUE) The index of the component to which this marker segment relates. The components are indexed 0, 1, 2, etc.
Scoc	**uint8_t**	Coding style for all components
SPcoc_Number_Decomposition_Levels	**uint8_t**	Number of decomposition levels parameter
SPcoc_Code_Block_Width	**uint8_t**	Width of the code block
SPcoc_Code_Block_Height	**uint8_t**	Height of the code block
SPcoc_Code_Block_Style	**uint8_t**	Style of the code block
precincts	**uint8_t***	Array of precincts

RGN Marker Segment (Region of Interest):

RGN Usage

Function: Signals the presence of an ROI in the codestream.
Usage: Main and first tile-part header of a given tile. If used in the main header it refers to the ROI scaling value for one component in the whole image, valid for all tiles except those with an RGN marker segment. When used in the tile-part header the scaling value is valid only for one component in that tile. There may be at most one RGN marker segment for each component in either the main or tile-part headers. The RGN marker segment for a particular component which appears in a tile-part header overrides any marker for that component in the main header, for the tile in which it appears. If there are multiple tile-parts in a tile, then this marker segment will only be found in the first tile-part header.

RGN Fields

Parameter Values

Table ISO-A-24 – Region of Interest Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
RGN	16	0xFF5E
Lrgn	16	5 — 6
Crgn	8 16	0 — 255; if Csiz < 257 0 — 16 383; Csiz ≥ 257
Srgn	8	Table A-25
SPrgn	8	Table A-26

ROI - Srgn Values

Table ISO-A-25 – Region-of-interest Parameter Values for the Srgn Parameter (ISO 15444-1)

Values	ROI style (Srgn)
0	Implicit ROI (maximum shift)
	All other values reserved

Table ISO-A-26 – Region-of-interest Parameter Values from SPrgn Parameter (Srgn = 0) (ISO 15444-1)

Parameters (in order)	Size (bits)	Values	Meaning of SPrgn value
Implicit ROI shift	8	0 — 255	Binary shifting of ROI coefficients above the background

RGN Implementation

Implemented in parsing.h.

Table 6, Implementation for RGN

Lrgn	**uint16_t**	length of marker segment in bytes (not including marker)
Crgn	**uint16_t**	The index of the component to which this marker segment relates. The components are indexed 0, 1, 2, etc.
Srgn	**uint8_t**	ROI style for the current ROI (**Table ISO-A-25** above shows the value for the Srgn parameter)
SPrgn	**uint8_t**	Parameter for ROI style designated in Srgn

QCD Marker Segment (Quantization Default):

QCD Usage

Function: Describes the quantization default used for compressing all components not defined by a QCC marker segment. The parameter values can be overridden for an individual component by a QCC marker segment in either the main or tile-part header.
Usage: Main and first tile-part header of a given tile. Only one in the main header, and at most one for all tile-part headers of a tile.
When used in the tile-part header it overrides the main QCD and main QCC for the specific component. The order of precedence is the following:

Tile-part QCC > Tile-part QCD > Main QCC > Main QCD

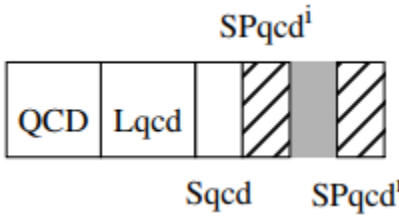


Figure ISO-A-13 – Quantization Default Syntax (ISO 15444-1)

QCD Fields

Parameter Values

Table ISO-A-27 – Quantization Default Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
QCD	16	0xFF5C
Lqcd	16	4 — 197
Sqcd	8	Table A-28
$SPqcd^i$	variable	Table A-28

Quantization Values

Table ISO-A-28 – Quantization Default Values for the Sqcd and Sqcc Parameters (ISO 15444-1)

Values (bits) MSB LSB	Quantization style	SPqcd or SPqcc size (bits)	SPqcd or SPqcc usage
xxx0 0000	No quantization	8	Table A-29
xxx0 0001	Scalar derived (values signalled for N_LLL subband only). Use Equation E.5.	16	Table A-30
xxx0 0010	Scalar expounded (values signalled for each subband). There are as many step sizes signalled as there are subbands.	16	Table A-30
000x xxxx — 111x xxxx	Number of guard bits 0 — 7		
	All other values reserved		

Table ISO-A-30 – Quantization Values for the SPqcd and SPqccc Parameters (irreversible transformation only) (ISO 15444-1)

Values (bits) MSB LSB	Quantization step size values
xxxx x000 0000 0000 — xxxx x111 1111 1111	Mantissa, μ_b, of the quantization step size value (see Equation E.3)
0000 0xxx xxxx xxxx — 1111 1xxx xxxx xxxx	Exponent, ε_b, of the quantization step size value (see Equation E.3)

Reversible Step Size Values

Table ISO-A-29 – Reversible Step Size Values for the SPqcd and SPqcc Parameters (reversible transform only)

Values (bits) MSB LSB	Reversible step size values
0000 0xxx — 1111 1xxx	Exponent, ε_b, of the reversible dynamic range signalled for each subband (see Equation E.5)
	All other values reserved

QCD Implementation

Implemented in parsing.h.

Table 7, Implementation for QCD

Lqcd	**uint16_t**	Length of marker segments in bytes (not including marker), determined by following equation: $$Lqcd = \begin{cases} 4 + 3 \cdot \text{number\_decomposition\_levels} & \text{no\_quantization} \\ 5 & \text{scalar\_quantization\_derived} \\ 5 + 6 \cdot \text{number\_decomposition\_levels} & \text{scalar\_quantization\_expounded} \end{cases}$$
Sqcd	**uint8_t**	Quantization style for all components
SPqcd	**uint8_t***	Quantization step size value for the ith subband in the defined order (the number of parameters is the same as the number of subbands in the tile-component with the greatest number of decomposition levels)

QCC Marker Segment (Quantization Component):

QCC Usage

Function: Describes the quantization used for compressing a particular component.
Usage: Main and first-tile part header of a given tile. Optional in both the main and tile-part headers. No more than one per any given component may be present in either the main or tile-part headers.
When used in the main header it overrides the main QCD marker segment for the specific component. When used in the tile-part header it overrides the main QCD, main QCC, and tile QCD for the specific component. The order of precedence is the following:

Tile-part QCC > Tile-part QCD > Main QCC > Main QCD

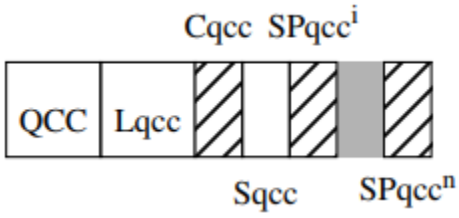


Figure ISO-A-14 – Quantization Component Syntax (ISO 15444-1)

QCC Fields

Parameter Values

Table ISO-A-31 – Quantization Component Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
QCC	16	0xFF5D
Lqcc	16	5 — 199
Cqcc	8 16	0 — 255; if Csiz < 257 0 — 16 383; Csiz ≥ 257
Sqcc	8	Table A-28
$SPqcc^i$	variable	Table A-28

QCC Implementation

Implemented in parsing.h.

Table 8, Implementation for QCC

Lqcc	**uint16_t**	Length of marker segments in bytes (not including marker), determined by following equation: $$Lqcc = \begin{cases} 5 + 3 \cdot \text{number\_decomposition\_levels} & \text{no\_quantization AND Csiz} < 257 \\ 6 & \text{scalar\_quantization\_derived AND Csiz} < 257 \\ 6 + 6 \cdot \text{number\_decomposition\_levels} & \text{scalar\_quantization\_expounded AND Csiz} < 257 \\ 6 + 3 \cdot \text{number\_decomposition\_levels} & \text{no\_quantization AND Csiz} \ddagger 257 \\ 7 & \text{scalar\_quantization derived AND Csiz} \ddagger 257 \\ 7 + 6 \cdot \text{number\_decomposition\_levels} & \text{scalar\_quantization\_expounded AND Csiz} \ddagger 257 \end{cases}$$
Cqcc	**uint16_t**	Index of the component to which this marker segment relates. Indexed 0, 1, 2, etc. **(Either 8 or 16 bit value depending on Csiz value)**
Sqcc	**uint8_t**	Quantization style for this component
SPqcc	**uint8_t***	Quantization value for each subband in the defined order

POC Tiles:

POC Usage

Function: Describes the bounds and progression order for any progression order other than specified in the COD marker segments in the codestream.
Usage: Main and tile-part headers. At most one POC marker segment may appear in any header. However, several progressions can be described with one POC marker segment. The progression order of a given tile is determined by the presence of the POC or the values of the COD in the following order of precedence:

Tile-part POC > Main POC > Tile-part COD > Main COD

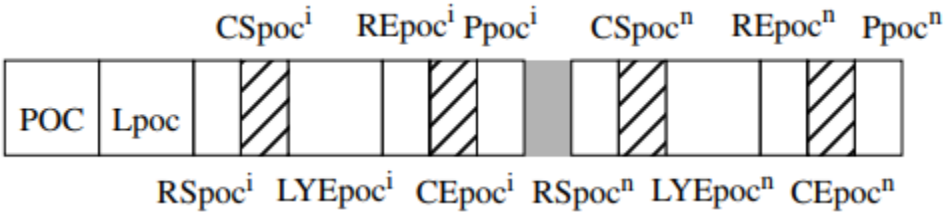


Figure ISO-A-15 – Progression Order Change Tile Syntax (ISO 15444-1)

POC Fields

Parameter Values

Table ISO-A-16 is above in the COD Fields section. Use this to reference the values for Ppoc.

Table ISO-A-32 – Progression Order Change, Tile Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
POC	16	0xFF5F
Lpoc	16	9 — 65 535
$RSpoc^i$	8	0 — 33
$CSpoc^i$	8 16	0 — 255; if Csiz < 257 0 — 16 383; Csiz ≥ 257
$LYEpoc^i$	16	0 — 65534
$REpoc^i$	8	$RSpoc^i$ — 33
$CEpoc^i$	8 16	$CSpoc^i$ — 255; if Csiz < 257 $CSpoc^i$ — 16 383; Csiz ≥ 257
$Ppoc^i$	8	Table A-16

POC Tiles Implementation

Implemented in parsing.h.

Table 9, Implementation for POC Tiles

RSpoc	**uint8_t**	Resolution level index (inclusive) for start of a progression. One value for each progression change in this tile or tile-part. The number of progression changes can be derived from the length of the marker segment.
CSpoc	**uint16_t**	Component index (inclusive) for start of a progression. Indexed 0, 1, 2, etc. (Either 8 or 16 bits depending on Csiz value). One value for each progression change in this tile or tile-part.
LYEpoc	**uint16_t**	Layer index (exclusive) for the end of a progression. The layer index always starts at zero for every progression. Packets that have already been included in the codestream are not included again. One value for each progression change in this tile or tile-part.
RESpoc	**uint8_t**	Resolution level index (exclusive) for the end of a progression. One value for each progression change in this tile or tile-part.
CEpoc	**uint8_t**	Component index (exclusive) for the end of a progression. The components are index 0, 1, 2, etc. (Either 8 or 16 bits depending on Csiz value). One value for each progression change in this tile or tile-part.
Ppoc	**uint16_t**	Progression order. One value for each progression change in this tile or tile-part

POC Marker Segment (Progression Order Change):

POC Usage

Reference the section above regarding the POC function and usage details.

POC Fields

Reference the section above regarding the POC fields.

POC Implementation

Implemented in parsing.h.

This struct contains an array of POC tiles which contains the main functionality for POC. (reference the POC tile section above for implementation details)

Table 10, Implementation for POC

Lpoc	**uint16_t**	Length of marker segments in bytes (not including marker), determined by following equation: $Lpoc = \begin{cases} 2 + 7 \cdot \text{number\_progression\_order\_change} & \text{Csiz} < 257 \\ 2 + 9 \cdot \text{number\_progression\_order\_change} & \text{Csiz} \ddagger 257 \end{cases}$
tiles	**JPEG2000_POC_TILES***	Array of POC tiles

TLM Marker Segment (Tile-part Lengths):

TLM Usage

Function: Describes the length of every tile-part in the codestream. Each tile-part's length is measured from the first byte of the SOT marker segment to the end of the bit stream data of that tile-part. The value of each individual tile-part length in the TLM marker segment is the same as the value in the corresponding Psot in the SOT marker segment.
Usage: Optional use in the main header only. There may be multiple TLM marker segments in the main header.

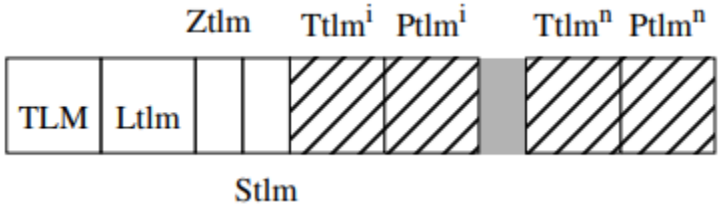


Figure ISO-A-17 – Tile-part Length Syntax (ISO 15444-1)

TLM Fields

Parameter Values

Table ISO-A-33 – Tile-part Length Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
TLM	16	0xFF55
Ltlm	16	6 — 65 535
Ztlm	8	0 — 255
Stlm	8	Table A-34
$Ttlm^i$	0 if ST = 0 8 if ST = 1 16 if ST = 2	tiles in order 0 — 254 0 — 65 534
$Ptlm^i$	16 if SP = 0 32 if SP = 1	13 — 65 535 13 — $(2^{32}-1)$

Size Parameters

Table ISO-A-34 – Size Parameters for Stlm (ISO 15444-1)

Values (bits) MSB LSB	Parameter size
xx00 xxxx	ST = 0; Ttlm parameter is 0 bits, only one tile-part per tile and the tiles are in index order without omission or repetition
xx01 xxxx	ST = 1; Ttlm parameter 8 bits
xx10 xxxx	ST = 2; Ttlm parameter 16 bits
x0xx xxxx	SP = 0; Ptlm parameter 16 bits
x1xx xxxx	SP = 1; Ptlm parameter 32 bits
	All other values reserved

TLM Implementation

Implemented in parsing.h.

Table 11, Implementation for TLM

Ltlm	**uint16_t**	Length of marker segments in bytes (not including marker), determined by following equation: $$Ltlm = \begin{cases} 4 + 2 \cdot \text{number\_of\_tile-parts\_in\_marker\_segment} & \text{ST=0 AND SP=0} \\ 4 + 3 \cdot \text{number\_of\_tile-parts\_in\_marker\_segment} & \text{ST=1 AND SP=0} \\ 4 + 4 \cdot \text{number\_of\_tile-parts\_in\_marker\_segment} & \text{ST=2 AND SP=0} \\ 4 + 4 \cdot \text{number\_of\_tile-parts\_in\_marker\_segment} & \text{ST=0 AND SP=1} \\ 4 + 5 \cdot \text{number\_of\_tile-parts\_in\_marker\_segment} & \text{ST=1 AND SP=1} \\ 4 + 6 \cdot \text{number\_of\_tile-parts\_in\_marker\_segment} & \text{ST=2 AND SP=1} \end{cases}$$
Ztlm	**JPEG2000_POC_TILES***	**I**ndex of this marker segment relative to all other TLM marker segments present in the current header. The sequence of (Ttlm, Ptlm) pairs from this marker segment is concatenated, in order of increasing Ztlm, with the sequences of pairs from other marker segments. The jth entry in the resulting list contains the tile index and tile-part length pair for the jth tile-part appearing in the codestream
Stlm	**uint8_t**	Size of the Ttlm and Ptlm parameters
Ttlm	**uint8_t***	Tile index of the ith tile-part. Either none or one value for every tile-part. The number of tile-parts in each tile can be derived from this marker segment (or the concatenated list of all such markers) or from a non-zero TNsot parameter, if present
Ptlm	**uint16_t***	Length, in bytes, from beginning of the SOT marker of the ith tile-part to the end of the bit stream data for that tile-part. One value for every tile-part
number_of_tile_parts	**uint32_t**	Number of tile parts

PLM Marker Segment (Packet Length, Main Header):

PLM Usage

Function: A list of packet lengths in the tile-parts for every tile-part in order.
Usage: There may be multiple PLM marker segments. Both the PLM and PLT marker segments are optional and can be used together or separately.

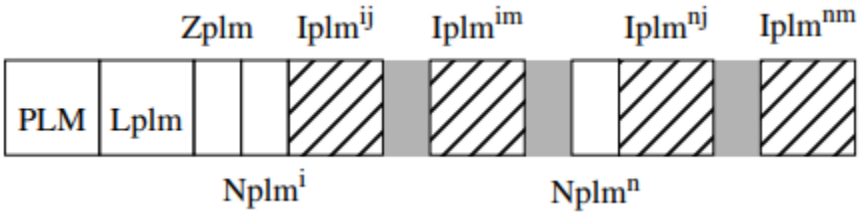


Figure ISO-A-18 – Packets Length, Main Header Syntax (ISO 15444-1)

PLM Fields

Parameter Values

Table ISO-A-35 – Packets Length, Main Header Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
PLM	16	0xFF57
Lplm	16	4 — 65 535
Zplm	8	0 — 255
$Nplm^i$	8	0 — 255
$Iplm^{ij}$	variable	Table A-36

Packet Length Values

Table ISO-A-36 – Iplm, Iplt List of Packet Lengths (ISO 15444-1)

Parameters (in order)	Size (bits)	Values	Meaning of Iplm or Iplt values
Packet length	8 bits repeated as necessary	0xxx xxxx 1xxx xxxx x000 0000 — x111 1111	Last 7 bits of packet length, terminate number[a] Continue reading[b] 7 bits of packet length

PLM Implementation

Implemented in parsing.h.

Table 12, Implementation for PLM

Lplm	**uint16_t**	Length of marker segments in bytes (not including marker)
Zplm	**uint8_t**	Index of marker segment relative to all other PLM marker segments present in current header (more info in **ISO/IEC 15444-1:2000(E)** document)

Not currently implemented is a 2D array of Iplm and array of Nplm. More information on these parameters are available in **ISO/IEC 15444-1:2000(E)** document.

PLT Marker Segment (Packet Length, Tile-part Header):

PLT Usage

Function: A list of packet lengths in the tile-part.
Usage: There may be multiple PLT marker segments per tile. Both the PLM and PLT marker segments are optional and can be used together or separately. They appear in any tile-part header before the packets whose lengths are described in more detail in **ISO 15444-1** documentation.

PLT Fields

Packet Length Values

Table ISO-A-37 – Packet Length, Tile-part Headers Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
PLT	16	0xFF58
Lplt	16	4 — 65 535
Zplt	8	0 — 255
$Iplt^i$	variable	Table A-36

Table ISO-A-36 is in the PLM Fields section above for reference of Iplt values.

PLT Implementation

Implemented in parsing.h.

Table 13, Implementation for PLT

Lplt	**uint16_t**	Length of marker segments in bytes (not including marker)
Zplt	**uint8_t**	Index of marker segment relative to all other PLT marker segments present in current header (more info in **ISO/IEC 15444-1:2000(E)** document)
Iplt	**uint64_t ***	Length of ith packet
nPackets	**uint16_t**	Number of packets

PPT Marker Segment (Packed Packet Headers, Tile-part Header):

PPT Usage

Function: A collection of the packet headers from one tile or tile-part.
Usage: Appears in any tile-part header before the packets whose headers are described in detail in **ISO 15444-1** documentation.
If there is no PPM marker segment (NPJE/EPJE formats) then the packet headers can be distributed either in PPT marker segments or distributed in the codestream. The packet headers can't be in both a PPT marker segment and the codestream for the same tile. If the packet headers are in PPT marker segments, they shall appear in a tile-part header before the corresponding packet data appears. There may be multiple PPT marker segments in a tile-part header.

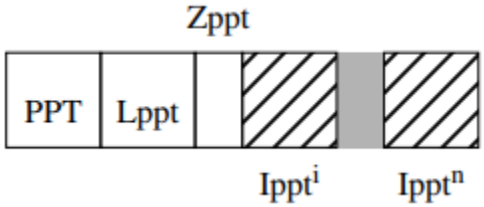


Figure ISO-A-21 – Packed Packet Headers, Tile-part Header Syntax

PPT Fields

Parameter Values

Table ISO-A-39 – Packet Header, Tile-part Headers Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
PPT	16	0xFF61
Lppt	16	4 — 65 535
Zppt	8	0 — 255
$Ippt^i$	variable	packet headers

PPT Implementation

Implemented in parsing.h.

Table 14, Implementation for PPT

Lppt	**uint16_t**	Length of marker segments in bytes (not including marker)
Zppt	**uint8_t**	Index of marker segment relative to all other PPT marker segments present in current header
Ippt	**uint64_t ***	Packet header for every packet in order in the tile-part

SOP Marker Segment (Start of Packet):

SOP Usage

Function: Marks the beginning of a packet within codestream.
Usage: Optional. May be used in the bit stream in front of every packet. Shall only be used if indicated in the proper COD marker segment. If PPM or PPT marker segments are used then the SOP marker segment may appear immediately before the packet data in the bit stream. Fixed length.

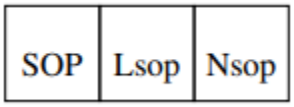


Figure ISO-A-22 – Start of Packet Syntax

SOP Fields

Parameter Values

Table ISO-A-40 – Start of Packet Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
SOP	16	0xFF91
Lsop	16	4
Nsop	16	0 — 65 535

SOP Implementation

Implemented in parsing.h.

Table 15, Implementation for SOP

Lsop	**uint16_t**	Length of marker segments in bytes (not including marker)
Nsop	**uint16_t**	Packet sequence number

CRG Pair:

CRG Pair Usage

Function: Allows specific registration of components with respect to each other. For coding purposes the samples of components are considered to be located at reference grid points that are integer multiples of XRsiz and YRsiz. The CRG marker segment describes the “center of mass” of each component’s samples with respect to the separation. This has no effect on decoding the codestream.
Usage: Main header only. Only one CRG may be used in the main header and is applicable for all tiles.

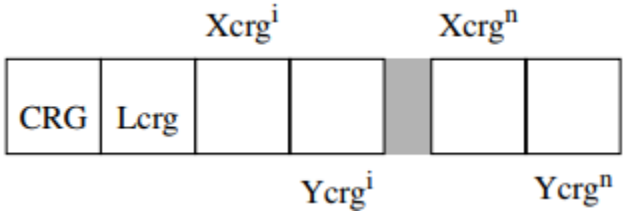


Figure ISO-A-23 – Component Registration Syntax (ISO 15444-1)

CRG Pair Fields

Parameter Values

Table ISO-A-42 – Component Registration Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
CRG	16	0xFF63
Lcrg	16	6 — 65 534
$Xcrg^i$	16	0 — 65 535
$Ycrg^i$	16	0 — 65 535

CRG Pair Implementation

Implemented in parsing.h.

Table 16, Implementation for CRG Pair

Xcrg	**uint16_t**	Value of horizontal offset for a component
Ycrg	**uint16_t**	Value of vertical offset for a component

CRG Marker Segment (Component Registration):

CRG Usage

Reference section above on CRG pairs for more details.

CRG Fields

Reference section above on CRG pairs for more details.

CRG Implementation

Implemented in parsing.h.

Table 17, Implementation for CRG

Lcrg	**uint16_t**	Length of marker segments in bytes (not including marker)
pairs	**JPEG2000_CRG_PAIR***	Array of CRG pairs

COM Marker Segment (Comment):

COM Usage

Function: Allows unstructured data in the main and tile-part header.
Usage: Repeatable as many times as desired in either or both the main or tile-part headers. This marker segment has no effect on decoding the codestream.

COM Fields

Parameter Values

Table ISO-A-43 – Comment Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
COM	16	0xFF64
Lcom	16	5 — 65 535
Rcom	16	Table A-44
$Ccom^i$	8	0 — 255

Rcom Registration Values

Table ISO-A-44 – Registration Values for the Rcom Parameter (ISO 15444-1)

Values	Registration values
0	General use (binary values)
1	General use (IS 8859-15:1999 (Latin) values)
	All other values reserved

COM Implementation

Implemented in parsing.h.

Table 18, Implementation for CRG

Lcom	**uint16_t**	Length of marker segments in bytes (not including marker)
Rcom	**uint16_t**	Registration value of marker segment
Ccom	**uint8_t***	Byte of unstructured data

SOT Marker Segment (Start of Tile-Part):

SOT Usage

Function: Marks the beginning of a tile-part, the index of its tile, and the index of its tile-part. The tile-parts of a given tile shall appear in order in the codestream. However, tile-parts from other tiles may be interleaved in the codestream. Therefore, the tile-parts from a given tile may not appear contiguously in the codestream.
Usage: Every tile-part header. Shall be the first marker segment in a tile-part header. There shall be at least one SOT in a codestream. There shall be only one SOT per tile-part. Fixed length.

SOT Fields

Parameter Values

Table ISO-A-5 – Start of Tile-part Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
SOT	16	0xFF90
Lsot	16	10
Isot	16	0 — 65 534
Psot	32	12 — $(2^{32}-1)$
TPsot	8	0 — 254
TNsot	8	Table A-6

Number of Tile-parts Values

Table ISO-A-6 – Number of Tile-parts, TNsot, Parameter Value

Value	Number of tile-parts
0	Number of tile-parts of this tile in the codestream is not defined in this header
1 — 255	Number of tile-parts of this tile in the codestream

SOT Implementation

Implemented in parsing.h.

Table 19, Implementation for SOT

Lsot	**uint16_t**	Length of marker segments in bytes (not including marker)
Isot	**uint16_t**	Tile index. This number refers to the tiles in raster order starting at the number 0.
Psot	**uint32_t**	Length, in bytes, from the beginning of the first byte of this SOT marker segment of the tile-part to the end of the data of that tile-part. Only the last tile-part in the codestream may contain a 0 for Psot. If the Psot is 0, this tile-part is assumed to contain all data until the EOC marker.
TPsot	**uint8_t**	Tile-part index. There is a specific order required for decoding tile-parts; this index denotes the order from 0. If there is only one tile-part for a tile then this value is zero. The tile-parts of this tile shall appear in the codestream in this order, although not necessarily consecutively.
TNsot	**uint8_t**	Number of tile-parts of a tile in the codestream. Two values are allowed: the correct number of tile-parts for that tile and zero. A zero value indicates that the number of tile-parts of this tile is not specific to this tile-part.

EOC Marker Segment (End of Codestream):

EOC Usage

Function: This indicates the end of the codestream.
Usage: This is the last marker in a codestream. There will only be one EOC per codestream.

EOC Fields

Parameter Values

Table ISO-A-8 – End of Codestream Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
EOC	16	0xFFD9

EOC Implementation

Implemented in decoder.c

We do not have a Struct for this marker segment in parsing.h like the others above. Instead, we compare this raw parameter value of 0xFFD9 in our code with the codestream markers we are parsing. This is implemented in the file decoder.c and the specific function **parse_JPEG2000** which starts on line 194. The comparison of the values is on line 364 within the function.

SOC Marker Segment (Start of Codestream):

SOC Usage

Function: This marks the beginning of a codestream.
Usage: This is the first marker in a codestream. There will only be one SOC per codestream. This is of fixed length, found in the main header.

SOC Fields

Parameter Values

Table ISO-A-4 – Start of Codestream Parameter Values (ISO 15444-1)

Parameter	Size (bits)	Values
SOC	16	0xFF4F

SOC Implementation

Implemented in decoder.c

We do not have a Struct for this marker segment in parsing.h like the others above. Instead, we compare this raw parameter value of 0xFF4F in our code with the codestream markers we are parsing. This is implemented in the file decoder.c and the specific function **jump_to_start_of_codestream** found on line 3.

Codestream Parsing

Overview

Given a JPEG-2000 codestream and user requests, the decoder needs to determine which portions of the compressed codestream need to be decoded. This stage parses through the marker segment locations in the codestream to identify information about the region(s) to be decoded. This prepares a struct containing nearly all relevant information about the file necessary for decoding the image data.

General Design

The general form of the encoded JPEG-2000 image is that of headers followed by their corresponding payloads. Each header contains information about the information that follows, and the payload is either further headers or encoded image data. **Figure BPJ2K-D-1** shows the layout of the file, with each header containing marker segments detailing the encoding properties or structure of the encoded file. This figure only applies to the NPJE format, other formats may follow a slightly different structure. The contents of the file will always start with the Main Header and end with an End of Codestream (EOC) marker.

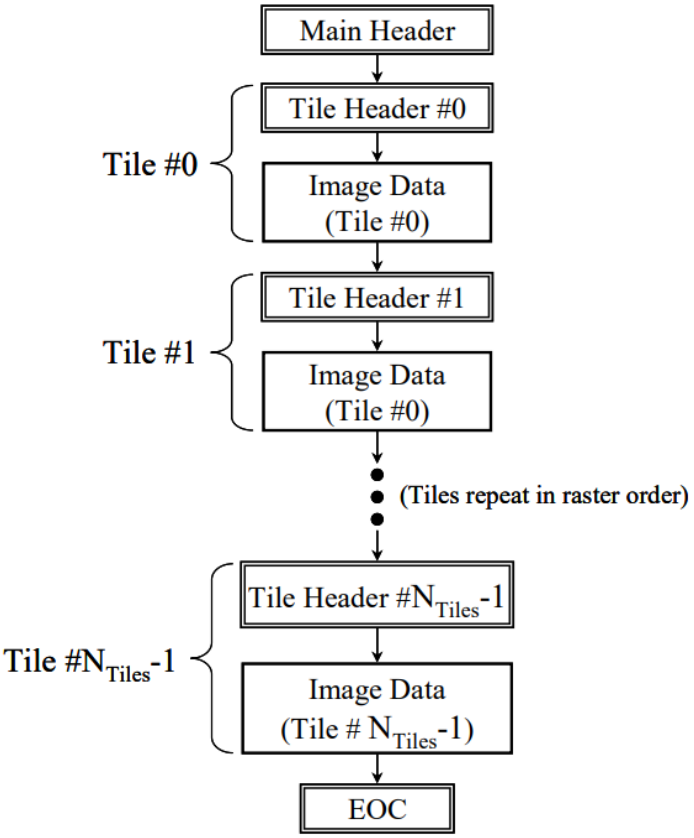


Figure BPJ2K-D-1 High-level Layout of NPJE JPEG-2000 Compressed File

Segment Marker Format

Each header type, both the Main Header and Tile Header(s), contain one or more marker segments as previously discussed. Both types are guaranteed to appear and are essential in determining how a file is encoded and should be unencoded. Typically, the Main Header contains content about the entire file and the Tile Headers contain information relevant to their Tile (or Tile-Part in non NPJE profiles); however, occasionally the Tile Header may override information from the Main Header. Additionally, some marker segments inside the Main or Tile Header may override previous marker segments in their own headers.

Main Header Decoding

The main header is the start of the encoded file and always begins with a Start of Codestream (SOC) marker followed by a SIZ marker. It then contains marker segments that pertain to how the entire file is encoded, often including information about tile sizes, packet information, custom encoding options, and comments. Note that the order matters for some marker segments, but these marker segments appear to be outside the scope of the project and aren't discussed. The main header is then always followed by a Tile Header (or theoretically the EOC marker).

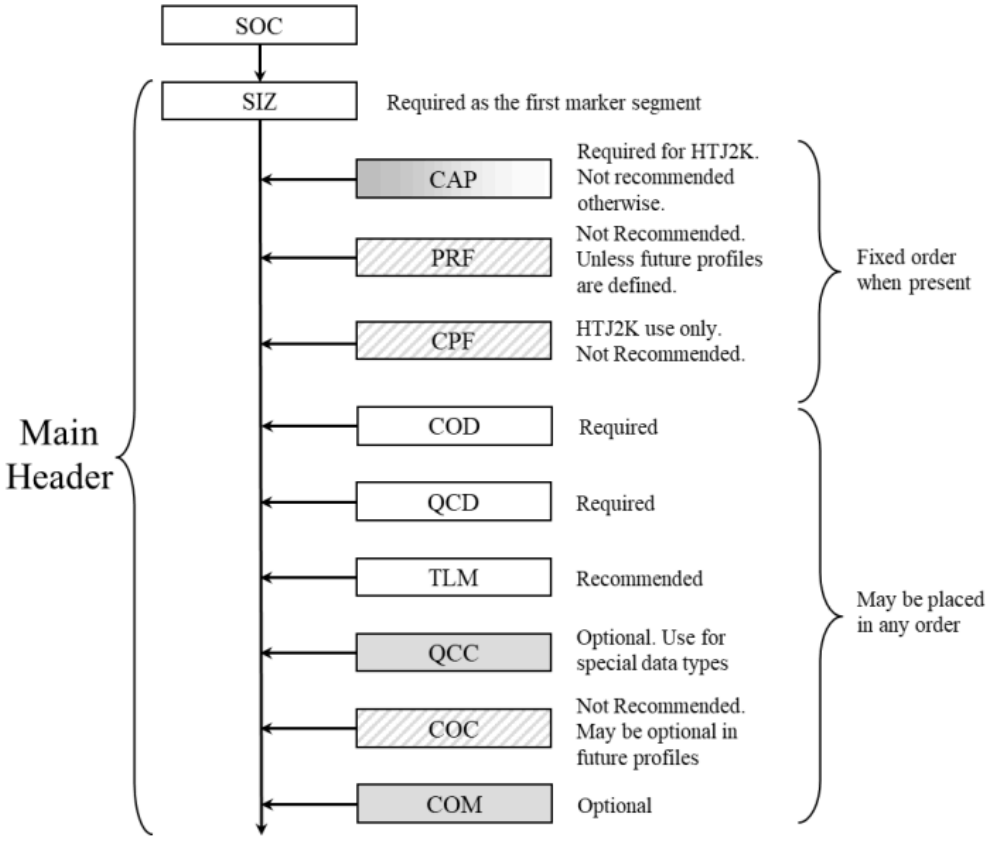


Figure D-2 Layout of JPEG 2000 main header for NPJE

Figure BPJ2K-D-2 Layout of JPEG-2000 Main Header for NPJE

Tile Header Decoding

Tile Headers always follow the Main Header or another tile and contain information about the encoded image data following it. They always begin with a Start of Tile (SOT) marker and end with a Start of Data (SOD) marker, but they may otherwise be empty. Any information in a Tile Header overrides information in the Main Header and together they provide the information to decode any data within a tile. The data within a tile always follows the SOD marker and ends at the next SOT marker or the EOC marker.

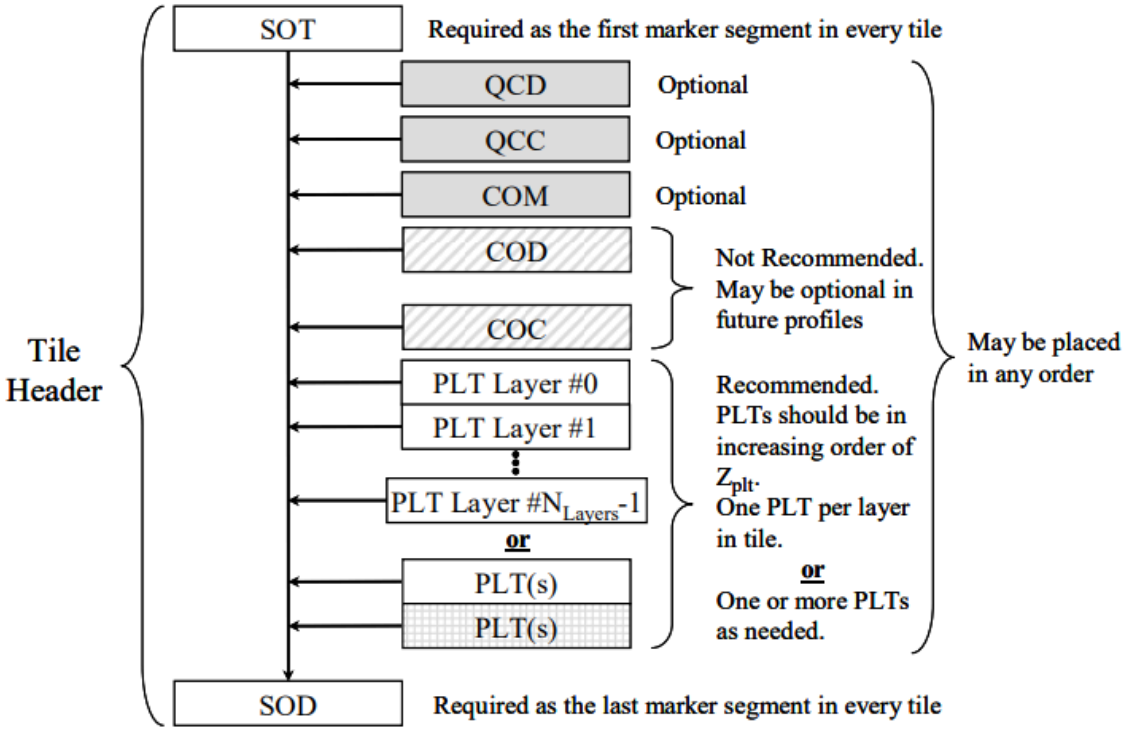


Figure BPJ2K-D-3 Layout of a Single JPEG-2000 Tile Header for NPJE

Implementation

Function Overview

- **Load functions:** Reads in bytes for each specified JPEG-2000 header component. Implemented in the main_header_decode_loop function in decoder.c.
- **Endian flip functions:** JPEG-2000 is in big endian, so for variables larger than 8 bits, we have an endian flip function within each JPEG-2000 Struct. Implemented in parsing.c.
- **Print functions:** For each JPEG-2000 Struct, we have a print function that prints each important variable within the struct for testing purposes. Implemented in parsing.c.

- **Parse JPEG-2000:** For each component, parse it into one of the structs created for the main header contents. These structs are identified in the JPEG-2000 Structs section. Implemented in parsing.c

Parsed Structures

Tile Part Header

Table 20, JPEG2000 Tile Part Header

SOT	**JPEG2000_SOT**	Start of tile-part
COD	**JPEG2000_COD**	Coding style default
COC	**JPEG2000_COC**	Coding style component
QCD	**JPEG2000_QCD**	Quantization default
QCC	**JPEG2000_QCC**	Quantization component
RGN	**JPEG2000_RGN**	Region of interest
POC	**JPEG2000_POC**	Progression order change
PPT	**JPEG2000_PPT**	Packed packet headers, tile-part header
PLT	**JPEG2000_PLT**	Packet length, tile-part header
COM	**PEG2000_COM**	Comment
plt_defined	**bool**	True or false depending on if the PLT marker segment is defined in the tile-part header

Packet

Table 21, JPEG2000 Packet

index	**uint16_t**	Index of packet
size	**uint64_t**	Size of packet
layer	**uint16_t**	Layer of packet
resolution	**uint16_t**	Resolution of packet
component	**uint16_t**	Component of packet

precinct	**uint16_t**	Precinct of packet
raw_data	**uint8_t***	Generic byte array to store raw data

Tile Part

Table 22, JPEG2000 Tile Part

nPackets	**uint16_t**	Number of packets in tile-part
header	**JPEG2000_TILE_PART_HEADER**	Header struct for the tile-part
packets	**JPEG2000_PACKET***	Packets in the tile-part

Tile

Table 23, JPEG2000 Tile

nParts	**uint16_t**	Number of tile-parts within the tile (always 1 for NPJE)
tile_parts	**JPEG2000_TILE_PART***	Array of tile-part structs within the tile (always length 1 for NPJE)

Main Header

Table 24, JPEG2000 Main Header

SIZ	**JPEG2000_SIZ**	Main header size
COD	**JPEG2000_COD**	Coding style default
COC	**JPEG2000_COC**	Coding style component
QCD	**JPEG2000_QCD**	Quantization default
QCC	**JPEG2000_QCC**	Quantization component
RGN	**JPEG2000_RGN**	Region of interest
POC	**JPEG2000_POC**	Progression order change
TLMS[255]	**JPEG2000_TLM**	Array of length 255 of TLM structs
num_TLMs	**uint8_t**	Number of TLMs
PLM	**JPEG2000_PLM**	Packet length, main header
CRG	**JPEG2000_CRG**	Component registration
COM	**JPEG2000_COM**	Comment

JPEG2000

Table 25, JPEG2000

header	**JPEG2000_MAIN_HEADER**	Main header struct of the JPEG-2000 image
number_of_tiles	**u_int16_t**	Number of tiles in the JPEG-2000 image
parts_per_tile	**u_int8_t***	Parts per tile in the JPEG-2000 image
number_of_tile_parts	**u_int16_t**	Number of tile parts in the JPEG-2000 image
tiles	**JPEG2000_TILE***	Array of tiles in the JPEG-2000 image

Layer & Progression Parsing

Overview

The goal of this stage is to use the relevant market segments previously parsed from the main header and tile header to obtain the encoded packets inside each tile-part. These packets are separated based on which layer, resolution layer, component, and precinct they correspond to. This will allow us to filter the decoding process upon request, which is required for the low resolution image preview and tile chipping use cases.

Design

The key to this stage is to understand the structure of how each image tile is encoded, and thus how it needs to be decoded. The order and method for obtaining the actual tile data, contained in the packets and code blocks of a tile, depends on the encoding options used.

Tile Structure

Each tile consists of a contiguous region of packets. These packets are encoded pieces of image data that will need to be decoded through the entropy decoder. For the progression parsing stage, we are only concerned with the order in which the packets are stored since that can affect which part of the image they represent. Each packet has a corresponding layer, resolution level, component, and precinct, but this information is not directly stored with the packet. Instead, we must determine this information based on the progression order (which can change between NPJE and EPJE formats) and where the packet is located in the aforementioned contiguous region.

Layers

The JPEG-2000 image data is distributed between multiple layers. Each layer may have its own default settings, such as the number of coding passes per code block.

Resolutions

Each JPEG-2000 image has several resolution levels. The first level, R0, provides the lowest amount of image resolution. The next level, R1, can be combined with R0 to provide a higher quality image. This algorithm is covered in greater detail in the wavelet transformation section.

Components

The image components in JPEG-2000 images often correspond to the images' colorspace components. While there can be more image components, we assume that the images will only have the 3 YCbCr components. These color values are converted to the RGB colorspace in the component transformation section of the pipeline.

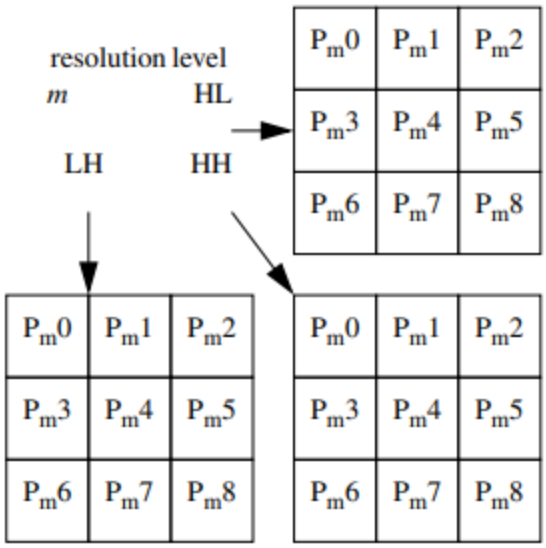


Figure ISO-B-10 Diagram of Precincts of One Resolution Level of One Component (ISO 15444-1)

Precincts

Precincts are groupings of code blocks, similar to layers. However, precincts may be non square and of variable size. **Figure ISO-B-11** demonstrates how code blocks are organized within a precinct to form a packet. Both NPJE and EPJE formats should only have 1 precinct, so precincts are not a concern of the current project.

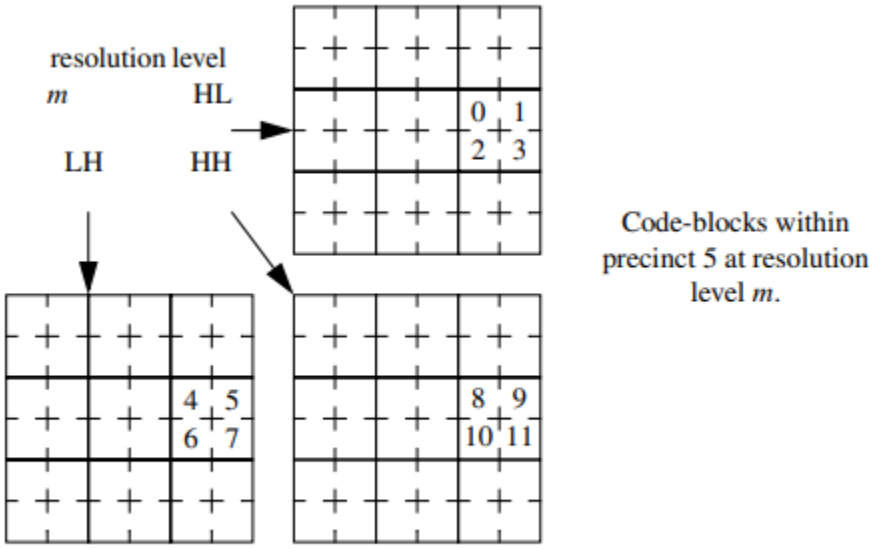


Figure ISO-B-11 Diagram of Code-Blocks Within Precincts at One Resolution Level (ISO 15444-1)

LRCP

LRCP progression encoding is the standard order for NPJE encoded images. The packets are first grouped by their layer. The number of layers is read and stored along with the header information. These groups are further subdivided by their resolution layer. The maximum resolution level is stored with the header information. (Note that we must add 1 in the implementation to account for R0, the initial resolution layer) The groups are again subdivided by the image component. This is also stored with the header information. Finally, these groups are subdivided by the precinct they belong to. This is not relevant with NPJE images since each tile only has 1 precinct, but this should be kept in mind if expanding to other formats. Additionally, there is only one tile-part per tile in the NPJE format. Despite this, our decoder currently has the capability to read and store the other tileparts. The image below provides an overall view of the packet order for an NPJE image.

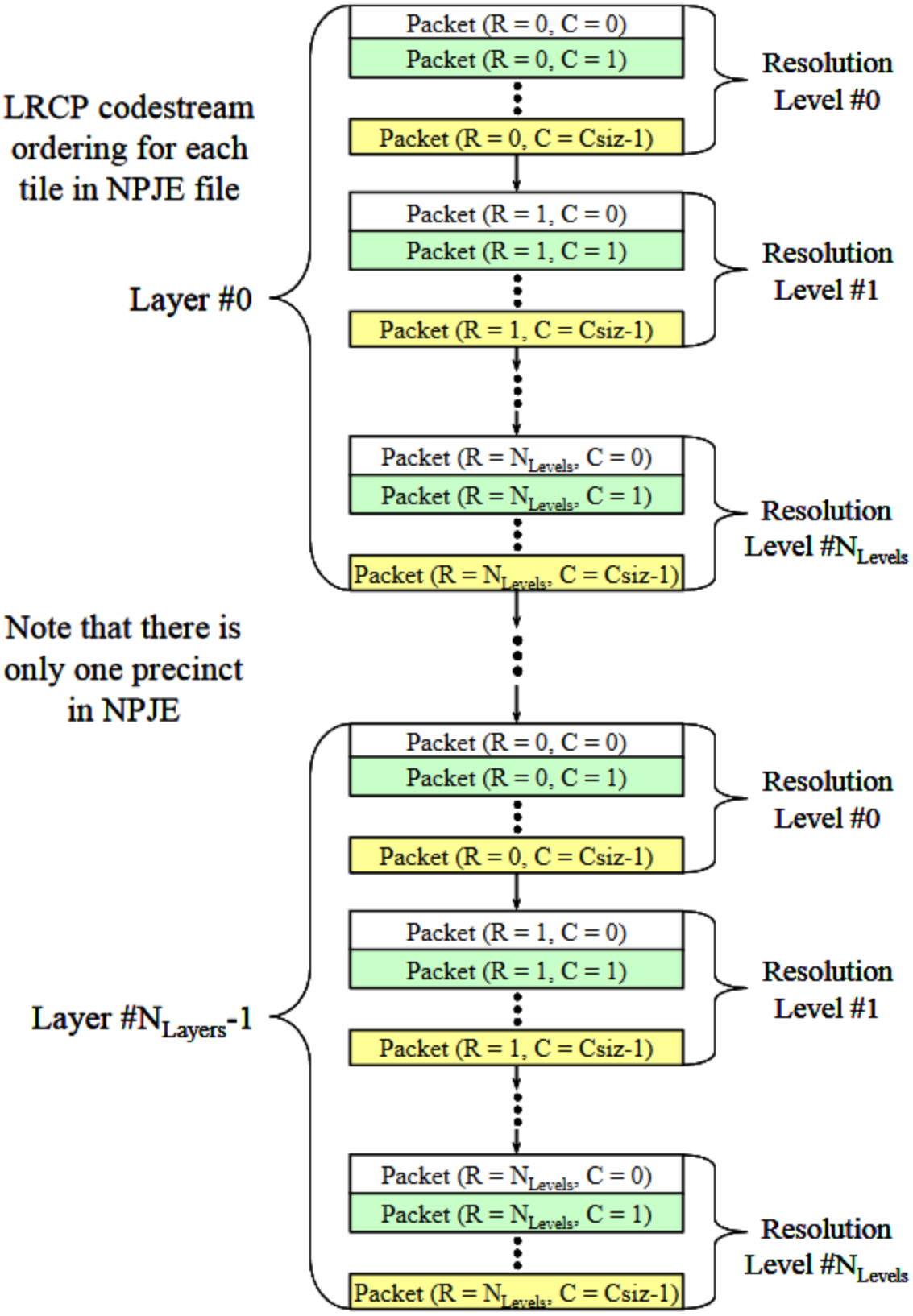


Figure BPJ2K-D-4: A High Level Layout of an NPJE Compressed JPEG-2000 (ISO 15444-1)

RLCP

RLCP progression encoding is the standard order for EPJE encoded images. The only difference between this encoding and LRCP (and between NPJE and EPJE altogether) is that the resolution levels make up the first subdivision, with layer number coming afterwards. This allows for a certain resolution level to be read more easily in an EPJE image, since it will exist in one contiguous region. A resolution layer in an NPJE image will be spread across the entire file since each layer will contain data on the requested resolution level. Additionally, since EPJE is essentially a reordering of an NPJE file, we expect that each file will only have 1 precinct.

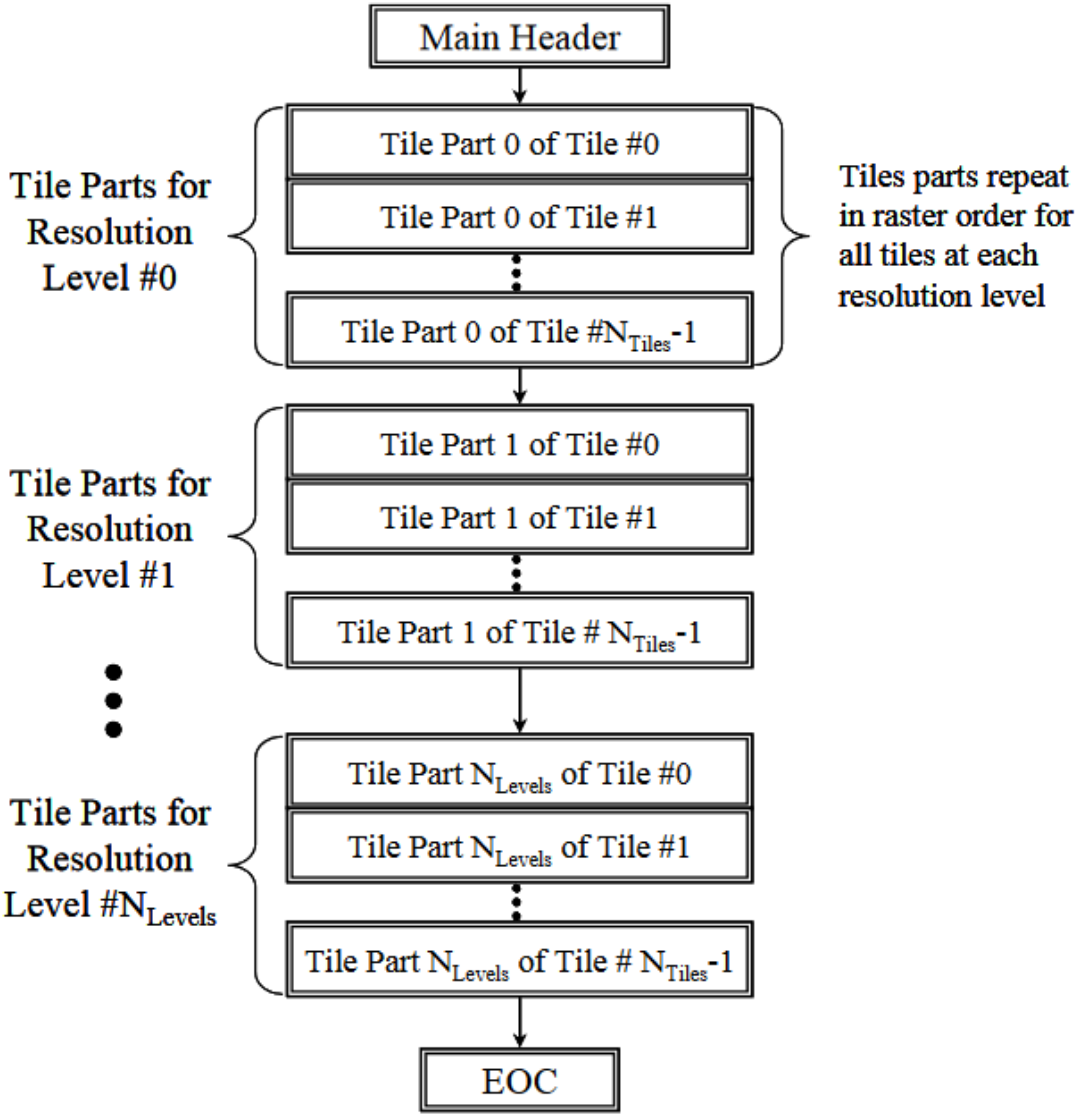


Figure BPJ2K-E-4: A High Level Layout of an EPJE Compressed JPEG-2000 (ISO 15444-1)

Implementation

Implemented in lrcp_parsing function in decoder.c.

In the planning stages of our decoder, we expected this area to be more complex, so we separated it into its own stage. After reviewing the implementation, we realized that this stage can be accomplished by a single function and could be combined with the codestream parsing stage. This likely only needs to change if additional options for the NPJE format need to be supported, or if EPJE support is added.

Entropy Decoding

Overview

Once the necessary entropy coded data needed to satisfy the user's request has been found, it needs to be decoded. This stage of the decoder must properly track what bitplanes are being decoded in which code blocks to ensure that the data is properly ordered for further processing. The output of this stage is predominantly the wavelet coefficients that can be used to begin reconstruction of an image's pixel data.

This section is unfinished and missing details due to an incomplete understanding of the stage from **ISO 15444-1**. Thus we offer only a barebones and highly simplified understanding of the stage in this document with references when possible. The actual complexity of this stage warrants the fact that our original six stage decoder design is not reflective of relatively equal complexity and effort between each stage.

Design

Packets (See **Annex B.9, ISO 15444-1)**

Packets contain the compressed image data corresponding to a specific tile, layer, component, resolution level, and precinct in a contiguous 8-bit aligned segment. Specifically, packets contain code-block information in a precinct and thus can be empty if the precinct has no information on any of the subbands at a specific resolution level. The data from each subband appears in the orders of LL, HL, LH, HH, with resolution level zero containing only the LL subband, and other levels containing only the HL, LH, and HH bands.

Each packet begins with a packet header and is followed by a packet body containing all code-bytes contributed by any relevant code-blocks. We cannot fully explain the contents of these packets, but attempt to provide information and references where possible. We believe the following figures to be particularly helpful.

Note from the ISO - Table ISO-B-3 shows an example of code-block coding passes that form packets. In Table ISO-B-3 the variables a, b, and c are code-block coding passes where a = significance propagation pass, b = magnitude refinement pass, and c = cleanup pass (see ISO 15444-1 Annex D).

Table ISO-B-3 Example of Packet Formation (ISO 15444-1)

	code-block 0	code-block 1	code-block 2	...	code-block 10	code-block 11	
MSB	c	0	0	...	c	0	
	a	0	0	...	a	0	packet 0
	b	0	0	...	b	0	
	c	c	0	...	c	0	
	a	a	0	...	a	0	
	b	b	0	...	b	0	packet 1
	c	c	c	...	c	c	
	...	...	...	...	...	...	
	a	a	a	...	a	a	etc.
LSB	b	b	b	...	b	b	
	c	c	c	...	c	c	

Table ISO-B-2 Example of Layer Formation (Only One Component Shown) (ISO 15444-1)

Resolution level	0			...	m				...	N_L		
Precinct	$P_0$0	$P_0$1	...	...	P_m0	P_m1	...	P_m8	...	P_{NL}0	P_{NL}1	...
Layer 0	Packet 0	Packet 0	...	...	Packet 0	Packet 0	...	Packet 0	...	Packet 0	Packet 0	...
Layer 1	Packet 1	Packet 1	...	...	Packet 1	Packet 1	...	Packet 1	...	Packet 1	Packet 1	...
...	...		...	...	...	...	...	...	...	...	...	...

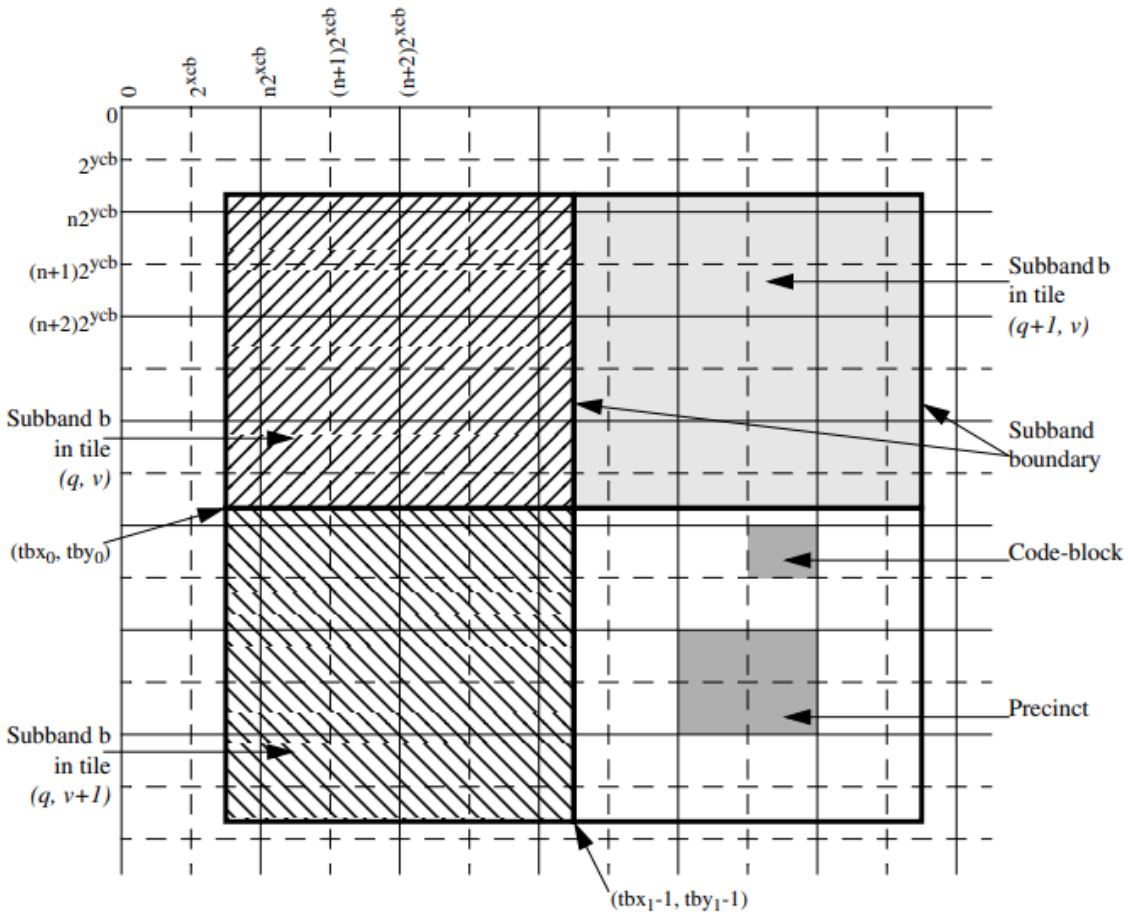


Figure ISO-B-9 Code-Blocks and Precincts in Subband *b* From Four Different Tiles (ISO 15444-1)

Packet Headers (See **Annex B.10, ISO 15444-1)**

Each packet header can contain the following information:

- Zero length packet
- Code-block inclusion
- Zero bit-plane information
- Number of coding passes
- Length of compressed code-block image data from a specific code-block

All data is assembled from most significant bit to least significant bit, with each byte appended to the header once assembled. If the byte appended is equal to 0xFF, the next byte will pad an extra zero bit into the MSB. The last byte cannot be 0xFF, and the last byte is fully packed for alignment.

Zero Length Packet

The first bit in the packet header is 0 if the packet is empty. Otherwise, the value is 1 and the rest of the packet exists.

Code-Block inclusion

This contains information whether a code-block is included in the packet. Code-blocks that have appeared in previous packets contain a binary 1 or 0 to indicate if the code-block is also included in this layer. If it's the first occurrence, then a sequence of bits from a tag tree is included to determine what portion is included in the current layer. The entire tag tree may not be included. We recommend reading **ISO 15444-1 B.10.4** for more clarity.

Zero Bit-Plane Information

If the code-block is included for the first time, this field specifies the number of bit-planes included in the codeblock. See **ISO 15444-1 B.10.5** for details.

Number of Coding Passes

This indicates the number of coding passes from each code-block in this packet using the codewords in **Table ISO-B-4.**

Table ISO-B-4 Codewords for the Number of Coding Passes for Each Code-Block

Number of coding passes	Codeword in packet header
1	0
2	10
3	1100
4	1101
5	1110
6 to 36	1111 0000 0 to 1111 1111 0
37 to 164	1111 11111 0000 000 to 1111 11111 1111 111

Length of the Compressed Code-Block Image Data

The packet identifies the number of bytes, but follows a consistent rule for the MQ decoder that the block cannot end in 0xFF and will readjust boundaries as necessary to fix that. We recommend reading **ISO 15444-1 B.10.7** for more details.

Tag Trees

Overview of Tag Trees:
A Tag Tree is a form of a quad tree, so each node can have up to four children. Each node's value is the minimum of its children's values. With clever encoding and decoding, the tag trees offer a convenient way to store a lot of information in a few bits.

How Tag Trees are decoded:

Decode:

for each layer $w, w = 0 : (W - 1)$
  for each leaf node $q_Z(i, j)$
    for each $q_z(i_z, j_z), 0 \leq z \leq Z$
      if $(S_z(i_z, j_z) = 0)$
        if $(z > 0)$ and $cv_z(i_z, j_z) < cv_{(z-1)}(i_{z-1}, j_{z-1})$
          $cv_z(i_z, j_z) = cv_{(z-1)}(i_{z-1}, j_{z-1})$ // Due to tag tree construction
        if $(cv_z(i_z, j_z) \leq w)$
          if next bit is '**1**,' set $S_z(i_z, j_z) = 1$ and $q_z(i_z, j_z) = cv_z(i_z, j_z)$
          else increment $cv_z(i_z, j_z)$ by 1.

Decoding Algorithm for Tag Tree Decoder (The JPEG 2000 Suite Pg 39)

Where W is the maximum number of layers, Z is the number of levels in the tag tree, $cv_z(i_z, j_z)$ denotes the current value of the node at (i,j) at level z, $S(i_z, j_z)$ denotes the state of the node at (i,j) at level z, and $q(i_z, j_z)$ denotes the actual value that was originally encoded into the tag tree at the node (i,j) at level z.

For an example, take the encoding of **Figure Suite-1.22**

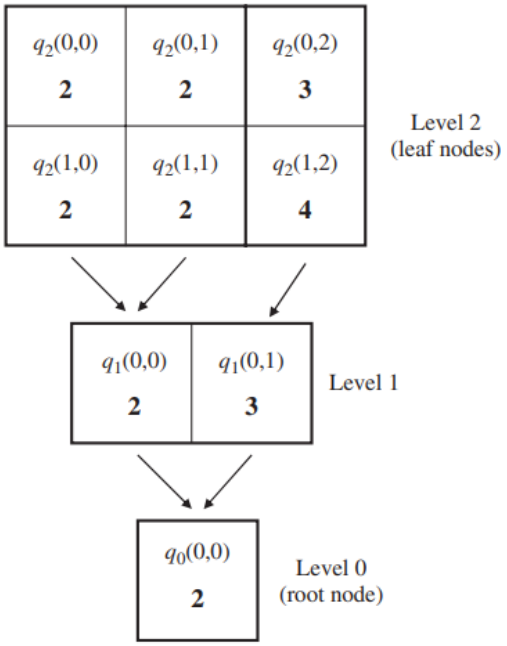


Figure Suite-1.22 (The JPEG 2000 Suite)

The encoded bits of **Figure Suite-1.22** are 0 0 1111011 110 1. The spaces are purely to denote the separation between layers.

To run the algorithm, initialize the current values and states of the tag tree to 0.

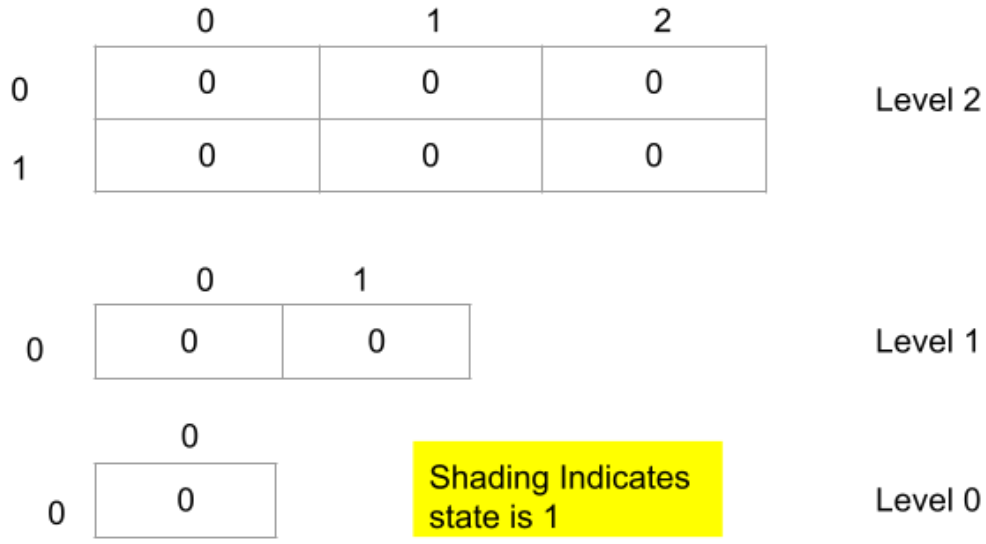


Figure 2, Tag Tree Decoding Example Initial State And Values

Layer 0

Start at the root node, the root node’s state is not yet 1, and the root’s current value is less than or equal to the layer. We read in the bit, and we read a 0. The current value goes to a 1 for the root node. All other nodes will now have their value go to a 1. This is because all other nodes will have their parents value exceed the node’s own value, and the node's new value exceeds the layer.

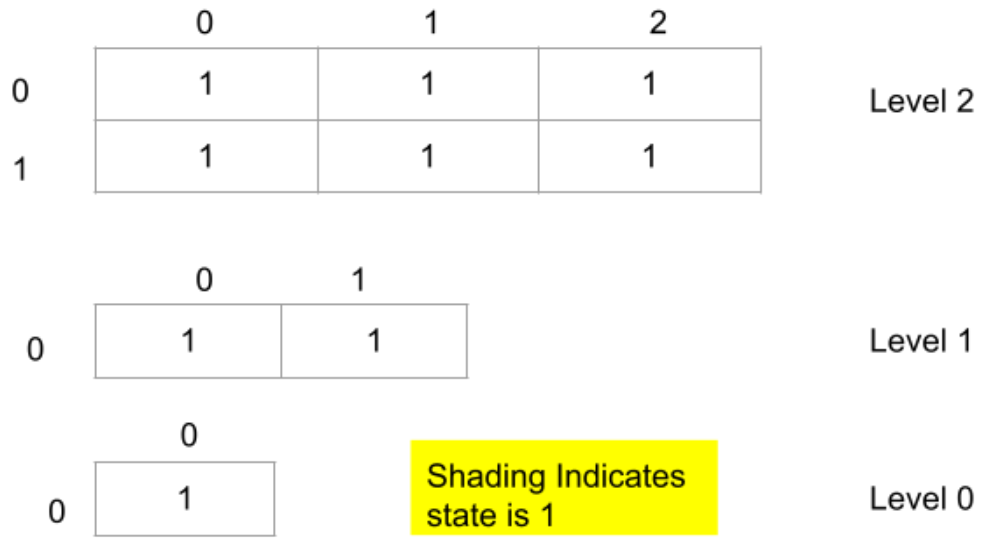


Figure 3, Tag Tree Decoding Example End of Layer 0 State And Values

Bits read 0

Bits remaining 0 1111011 110 1

Layer 1

Start at the root node, the execution ends up being the same as layer 0. The root node’s current value is less than or equal to the layer, and the next bit was a 0, so the root increments to 2. All others go to 2 and the layer is concluded.

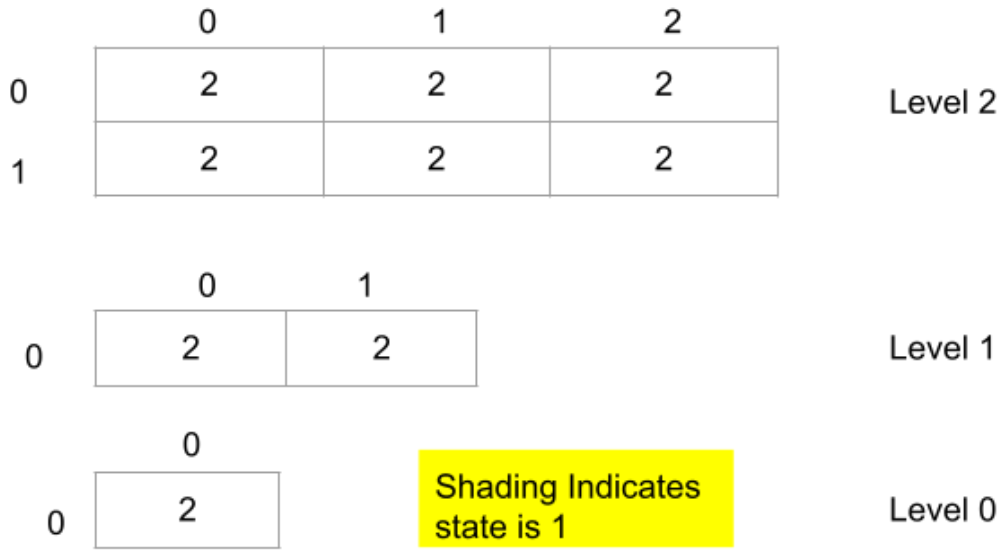


Figure 4, Tag Tree Decoding Example End of Layer 1 State And Values

Bits read 0

Bits remaining 1111011 110 1

Layer 2

Start at the root node. Its current value is less than or equal to layer 2, so we read in the next bit. The next bit is a 1, so we set the state for the root node to 1. This has the effect of freezing the value, which is at 2, so we will say that this node is locked.

Now we got node q1(0,0). This node's current value does not exceed its parents value, so we check and see it does not exceed the layer either. We read the next bit and see that it is a 1, and we lock q1(0,0). Proceeding, we lock, we read a 1 for q0(0,0) and lock it. As we go in raster order of the leaf nodes, checking ancestors when necessary, we go on to examine q0(0,1). Its ancestors are all locked, so we ignore them. Q1(0,1)'s current value neither exceeds its parents nor exceeds the layer, so we read the next bit. It is a 1, so we lock q0(0,1).

Now we go to node q2(0,2), but we haven't read the parent yet. So we go to the parent q1(0,1) and we see that the parent's current value neither exceeds the root's nor the layer. So we read the next bit. The bit is a 0, so we must increment the state to 3. Back to q2(0,2), its current value is less than its parents current value, so its current value goes to its parents current value.

For node q2(1,0), we read a 1, and lock it. For q2(1,1), we read a 1 and lock it.

For node q2(1,2), we check q1(0,1). As q1(0,1) exceeds its parent and the current layer, we do nothing. Returning to q2(1,2) we see that its current value is less than its parent's current value, so we increment.

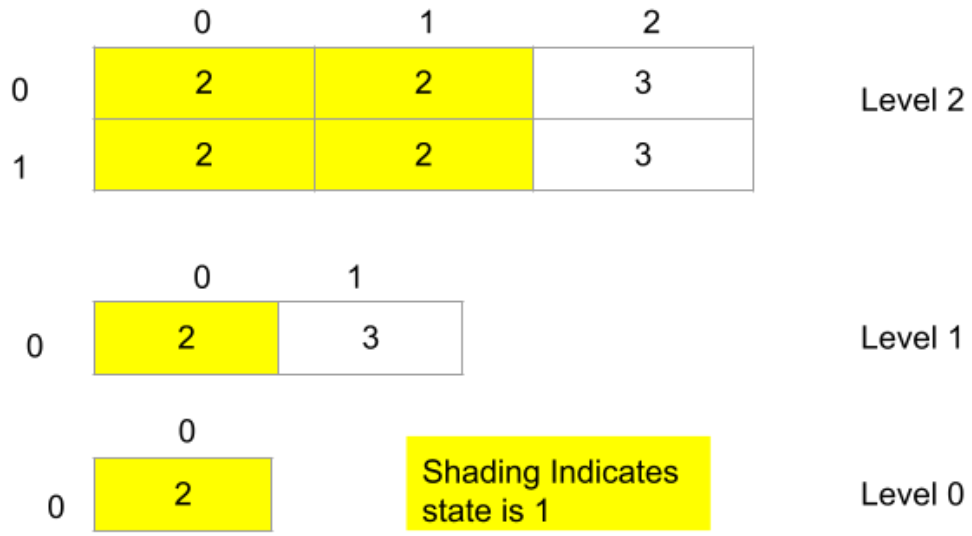


Figure 5, Tag Tree Decoding Example End of Layer 2 State And Values

Bits read 1111011

Bits remaining 110 1

Layer 3

We go to the first unlocked node. Q2(0,2). We examine the unlocked parent of that node, q1(0,1). Q1(0,1)'s value neither exceeds the root's value nor the layer, so we read the next bit. The next bit is a 1, so we set the state of q1(0,1) to a 1. Returning to q2(0,2) we notice that it's current value does not exceed the parent's current value nor the layer, so we read a bit. The next bit is a 1, so we lock q2(0,2).

We then go to q2(1,2). As its current value does not exceed its parents and its current value is less than or equal to the layer, we read a bit. The bit is 0, so we increment the current state to 4.

Figure 6, Tag Tree Decoding Example End of Layer 3 State And Values

Bits read 110

Bits remaining 1

Layer 4.

The first unlocked node is q2(1,2). All its parents are locked and ignored. As the current value neither exceeds its parents nor the layer, we read the next bit. The next bit is a 1, so we lock the q2(1,2). At this point all values are locked, so for all future layers no more bits will be read.

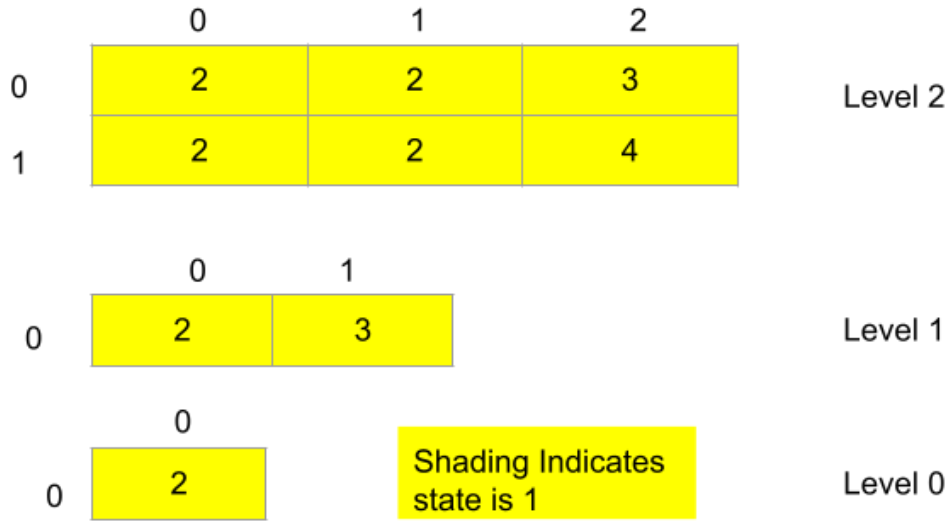


Figure 7, Tag Tree Decoding Example End of Layer 4 State And Values

Bits read 1

Bits remaining (none)

Notice how the values recovered are the exact same of the encoded tag tree from **Figure Suite-1.22**. If you are interested in how the tag tree was encoded, we encourage you to read **The JPEG 2000 Suite Section 1.3.5.1**.

MQ Decoder

The MQ coder is a type of arithmetic coder used to compress the image data in the JPEG-2000 format. It uses pairs of Contexts and Binary Decisions (CX and D in **ISO 15444-1**) to produce the compressed image data at each step (CD). Each action appears to apply to the code-blocks contained across the image packets, but they also seem to require other information included in the packets. Annex D in **ISO 15444-1** appears to detail the relation in how the bits are modeled across code-blocks and how they are used in an arithmetic encoder.

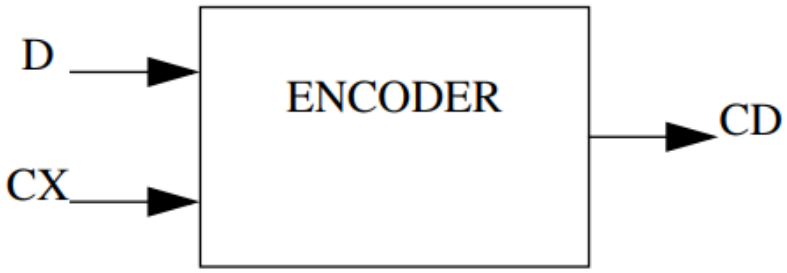


Figure ISO-C-1 Arithmetic Encoder Inputs and Outputs (ISO 15444-1)

Annex C and D of **ISO 15444-1** prominently detail the actual encoding steps used by the format; however, they lack in detail on the decoding steps. As we believe this transformation is entirely reversible since no quantization is used in the NPJE images necessary for the project, it should be possible to reverse engineer each step to produce an equivalent MQ decoder from the encoder. This indicates that although the encoder is described as probabilistic, the probabilities should converge to the same compressed data for a set of decisions and contexts.

We do not discuss further details about the MQ decoder design here and instead recommend looking through the **ISO** and other resources to determine how it functions. Connecting this to the rest of the decoder is the primary area of interest for the continuation of this project. In the implementation section, we discuss the JJ2000 MQ decoder implementation which we believe is the best place to examine for a functional implementation, and an attempt to copy this implementation in C can already be found in **mq_decoder.c.** Likely the best route to finish the Entropy Decoding stage is to reverse engineer the JJ2000 implementation.

Decoding Loop

We don’t discuss with certainty the details of how the MQ decoder fits into the image decoder, but we do wish to point out what we believe to be the best starting point for understanding the design. Annex D of **ISO 15444-1** discusses how each code-block has three main passes through the encoder, a significant propagation, magnitude refinement, and cleanup pass. We believe each code-block bit-plane (see **ISO 15444-1** Annex B and Section 5.3 for more information about bit-planes) goes through the encoder in a loop repeating each type of pass as necessary until the image is encoded. Therefore, reversing this process to decode the image is what we believe to be the best way to understand how this program is implemented. **ISO 15444-1** Annex D contains the most information on this loop with helpful figures, but we copy some of the figures below for reference.

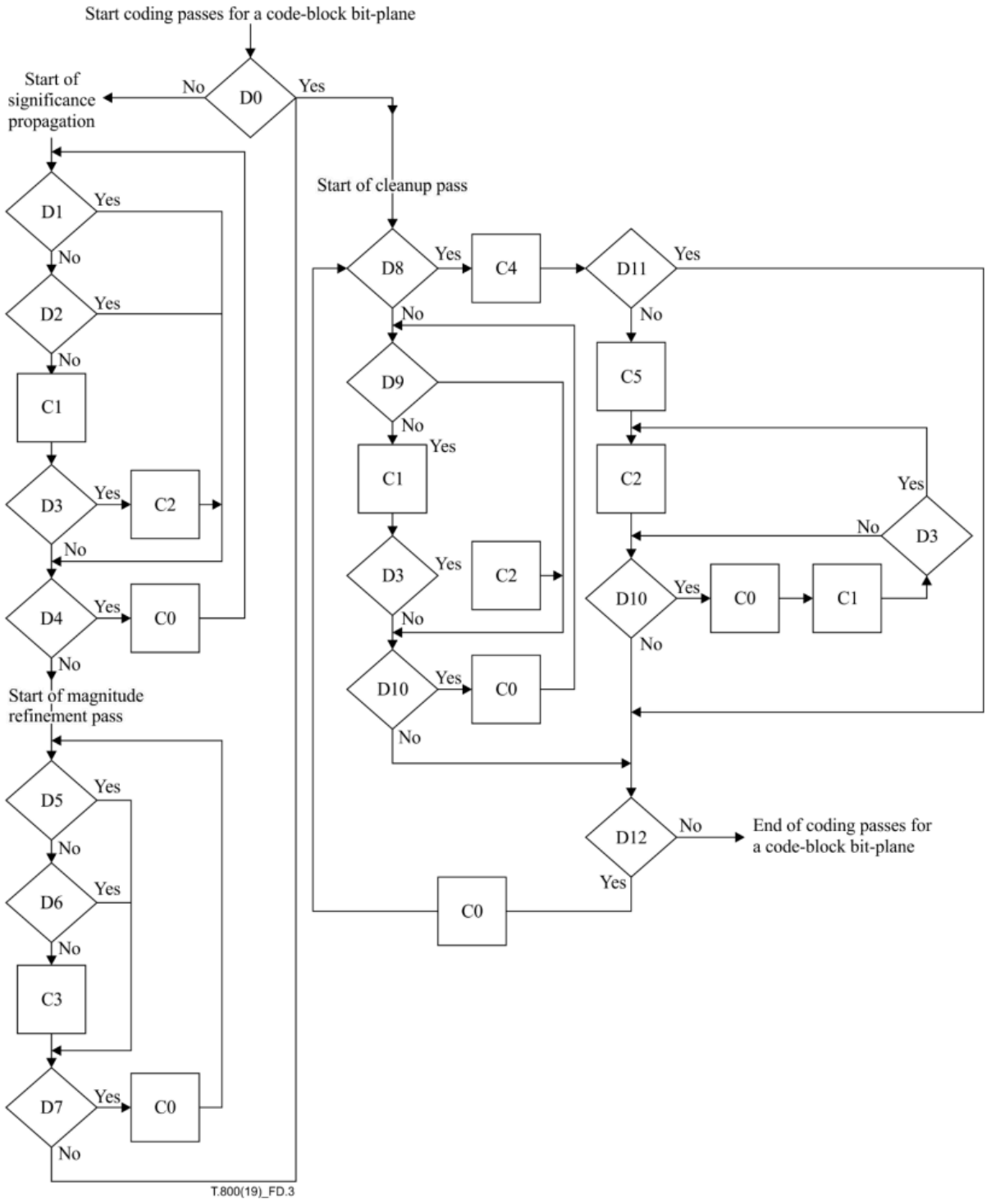


Figure ISO-D-3 Flow Chart for all Coding Passes on a Code-Block Bit-Plane

Table ISO-D-10 Decisions in the Context Model Flow Chart

Decision	Question	Description
D0	Is this the first significant bit-plane for the code-block?	See D.3
D1	Is the current coefficient significant?	See D.3.1
D2	Is the context bin zero? (see Table D.1)	See D.3.1
D3	Did the current coefficient just become significant?	See D.3.1
D4	Are there more coefficients in the significance propagation?	
D5	Is the coefficient insignificant?	See D.3.3
D6	Was the coefficient coded in the last significance propagation?	See D.3.3
D7	Are there more coefficients in the magnitude refinement pass?	
D8	Are four contiguous undecoded coefficients in a column each with a 0 context?	See D.3.4
D9	Is the coefficient significant or has the bit already been coded during the Significance Propagation coding pass?	See D.3.4
D10	Are there more coefficients remaining of the four column coefficients?	
D11	Are the four contiguous bits all zero?	See D.3.4
D12	Are there more coefficients in the cleanup pass?	

Table ISO-D-11 Decoding in the Context Model Flow Chart

Code	Decoded symbol	Context	Brief explanation	Description
C0	–	–	Go to the next coefficient or column	
C1	Newly significant?	Table D.1, 9 context labels	Decode significance bit of current coefficient (significance propagation or cleanup)	See D.3.1
C2	Sign bit	Table D.3, 5 context labels	Decode sign bit of current coefficient	See D.3.2
C3	Current magnitude bit	Table D.4, 3 context labels	Decode magnitude refinement pass bit of current coefficient	See D.3.3
C4	0 1	Run-length context label	Decode run-length of four zeros Decode run-length not of four zeros	See D.3.4
C5	00 01 10 11	UNIFORM	First coefficient is first with non-zero bit Second coefficient is first with non-zero bit Third coefficient is first with non-zero bit Fourth coefficient is first with non-zero bit	See D.3.4 and Table C.2

Implementation

Our Implementation:

Implemented in coefficient.c

We have an incomplete implementation of parsing packet headers. This implementation has not been connected to the rest of the decoder, and no structs exist to hold the parsed data from this file. With our implementation, in the decoder.c file we leave off with the lrcp_parsing storing the raw data of the packets. That function is most likely where the coefficient.c file should begin to be linked.

Implemented in tag_tree_decoder.c

For the Tag Tree Decoder, we follow the implementation of JJ2000's TagTreeDecoder (src/main/java/ucar/jpeg/JJ2000/j2k/codestream/reader/TagTreeDecoder.java) very closely. Because we did not implement the Entropy Decoder, our Tag Tree Decoder's update function is incomplete. The reading of specific bits or bytes needs to be added in order for it to work. As such, the Tag Tree Decoder is not connected to any of the other work in our decoder. It would most likely get connected to the coefficient.c, which parses some of the packet headers.

Resources:

We have located some of the core decoding operations related to the MQ Decoder in JJ2000, a Java implementation of JPEG-2000. We have JJ2000 added to our repo, for we believe this implementation is the most readable of the implementations we have found so far. This implementation has an abstract class for the EntropyDecoder, located in src/main/java/ucar/jpeg/JJ2000/j2k/entropy/decoder/EntropyDecoder.java, and an implementation of that abstract class as the StdEntropyDecoder. There is also an MQDecoder, which claims to have some optimizations.
When following the execution of the JJ2000 decoder, you can see the process that it follows JJ2KDecoder (src/main/java/ucar/jpeg/JJ2KDecoder.java) -> CmdLnDecoder (src/main/java/ucar/jpeg/JJ2000/j2k/decoder/CmdLnDecoder.java) -> Decoder (src/main/java/ucar/jpeg/JJ2000/j2k/decoder/Decoder.java). The run function shows instantiating and performing the key stages of the decoder. The Decoder then creates a HeaderDecoder (src/main/java/ucar/jpeg/JJ2000/j2k/codestream/reader/HeaderDecoder.java) which is used to create the EntropyDecoder, which eventually calls the MQDecoder. Tracing the path from the Decoder to calling the EntropyDecoder may require debugging, as we have not had success finding what eventually calls the EntropyDecoder.
A more complete overview of the file structure of JJ2000 is offered by **ISO 15444-5 B.3.**

Wavelet Transform

Overview

Once the wavelet coefficients have been reconstructed, the inverse discrete wavelet transform (IDWT) is performed. This is done to reconstruct a specific spatial region of the JPEG-2000 image based on the coefficients to decode. Wavelet transformations are common in image processing and allow for image features to be modeled as a waveform that can be easily reconstructed with minimal extra data (the coefficients). **Implemented in wavelet.c.**

Implementation

Implementation Overview

The previous step outputs 2D matrices of wavelet coefficients, known as subbands. Each resolution layer is composed of four subbands, LL, HL, LH, and HH, and are based on the types of passes applied to the original image. For example, the HL subband of the nth resolution layer is the result of applying a horizontal high-pass filter and a vertical low-pass filter to the $(n-1)$th resolution layer. (Note that the 0th resolution layer is the original image) The HL, LH, and HH for each resolution layer are provided by the previous steps, but the LL subband is the reconstructed $(n+1)$th resolution layer. Therefore, to reconstruct any resolution layer, the algorithm needs the previous resolution layers as well.

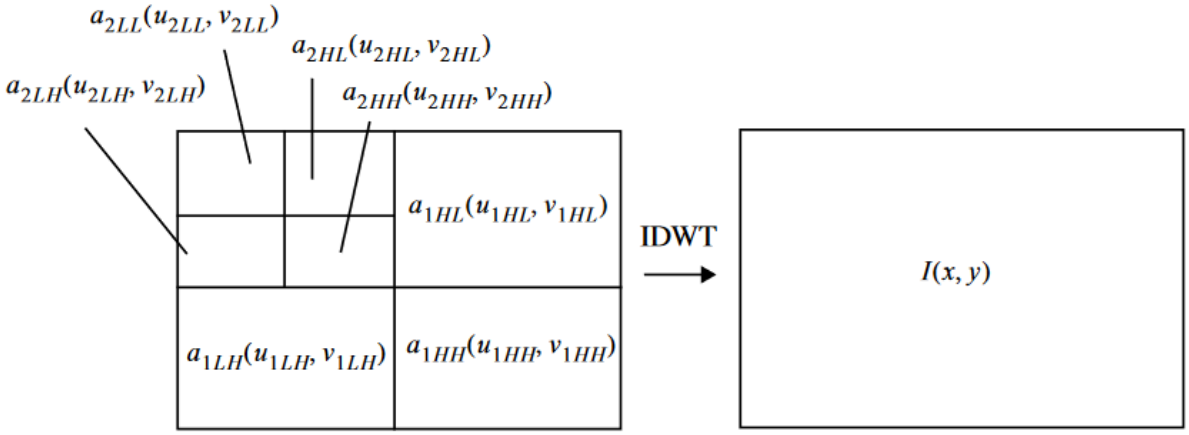


Figure ISO-F-2: A High Level Overview of the IDWT Procedure (ISO 15444-1)

IDWT Procedure

The IDWT procedure is an iterative process that repeatedly combines four regions of subbands into a reconstructed resolution layer. Ultimately, it is a wrapper that applies the 2D-SR procedure to each level of the image. Notably, this procedure requires the inputs in the form of $[N_L LL, N_L HL, N_L LH, (N_L-1)HH, (N_L-1)HL, (N_L-1)LH, \ldots, 0HH, 0HL, 0LH, 0HH]$ where the

image has N_L resolution layers and $N_L LL$ refers to the LL subband of the N_Lth layer, for example. Additionally, since this is an iterative process, any resolution layer can be returned by performing fewer iterations of the 2D-SR algorithm, so long as all the subbands are provided.

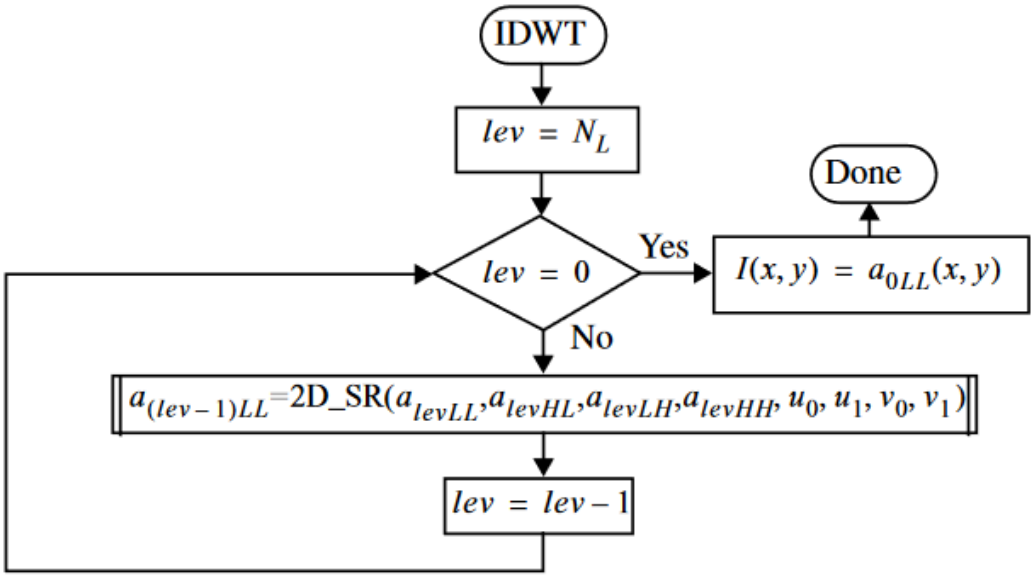


Figure ISO-F-3: A Flow Diagram for the IDWT Procedure (ISO 15444-1)

2D-SR Procedure

The 2D-SR procedure takes four subbands, $[nLL, nHL, nLH, nHH]$, and reconstructs the subband $(n-1)LL$. The number of coefficients in this new subband is equal to the total number of coefficients in the input subbands. The procedure itself starts by interleaving the four subbands, which is described in detail in the 2D-interleave procedure section. Then, the procedure calls the HOR-SR and VER-SR procedures. Both procedures are variations of the 1D-SR procedure. HOR-SR applies the 1D-SR to every horizontal row of the interleaved result while VER-SR applies the algorithm to every vertical column. After both passes are complete, the procedure returns the reconstructed subband.

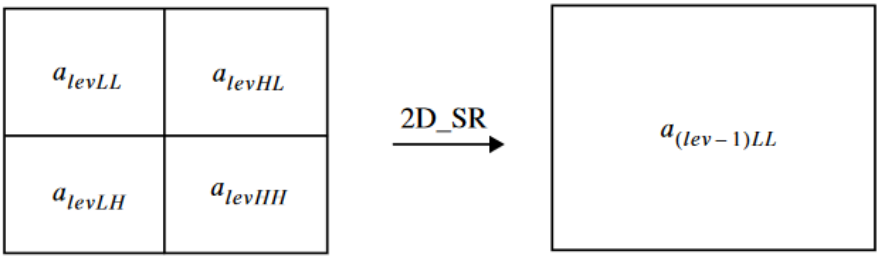


Figure ISO-F-5: One Level of Reconstruction from Four Subbands (ISO 15444-1)

2D-INTERLEAVE Procedure

The 2D-INTERLEAVE procedure takes four matrices representing the subbands and outputs an interleaved matrix. The size of the resulting matrix is the size of the LL and HH subbands summed. For example, if the LL subbands is 6 by 5 and the HH subband is 4 by 3, the interleaved matrix will be 10 by 8.

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

Figure 8.1, Example LL subband

2	2	2
2	2	2
2	2	2
2	2	2

Figure 8.2, Example HL subband

3	3	3	3
3	3	3	3
3	3	3	3

Figure 8.3, Example LH subband

4	4	4
4	4	4
4	4	4

Figure 8.4, Example HH subband

1	2	1	2	1	2	1
3	4	3	4	3	4	3
1	2	1	2	1	2	1
3	4	3	4	3	4	3
1	2	1	2	1	2	1
3	4	3	4	3	4	3
1	2	1	2	1	2	1

Figure 8.5, Interleaving Result

1D-SR Procedure

The 1D-SR applies the lifting-based filter to a 1D array of coefficients. As described earlier, this is applied to each row and column of the interleaved array. The array must first be extended so that the filter can be properly applied to each element. This extension depends on the parity of the starting and ending indices, denoted by i_0 and i_1 respectively in the following tables.

Table ISO-F-2 Extension to the left

i_0	$i_{left_{5-3}}$	$i_{left_{9-7}}$
even	1	3
odd	2	4

Table ISO-F-3 Extension to the right

i_1	$i_{right_{5-3}}$	$i_{right_{9-7}}$
odd	1	3
even	2	4

Once this extension is applied, the filter is applied to each element in the array. This happens in two passes, one for the even indices and one for the odd indices. Separate algorithms are applied on each pass. These algorithms are provided below.

$$X(2n) = Y_{ext}(2n) - floor(\frac{Y_{ext}(2n-1) + Y_{ext}(2n+1) + 2}{4}) \text{ for } floor(\frac{i_0}{2}) \leq n < floor(\frac{i_1}{2}) + 1$$

$$X(2n+1) = Y_{ext}(2n+1) + floor(\frac{X(2n) + X(2n+2)}{2}) \text{ for } floor(\frac{i_0}{2}) \leq n < floor(\frac{i_1}{2})$$

Equations ISO-F-5 and ISO-F-6 Wavelet Lifting-based Filtering

Image De-Tiling

Overview

In this stage of the decoder, the image tiles are re-assembled spatially using a matrix to form the reconstructed final image after decoding has taken place. This process includes any necessary upsampling needed to bring all codestream components to the same spatial resolution. This includes processing tile parts within tiles from the original image, and then upsampling the processed image to return the final image. Unfortunately there is a lack of documentation available for this section of the decoder, so most of the information found on image de-tiling is from additional resources to the **ISO 15444-1** and **BPJ2K** documents referenced throughout this documentation. Also note that the **ISO 15444-1** documentation does not describe any specific upsampling techniques.

Upsampling Techniques

To achieve fast performance times, there needs to be a balance between computational efficiency and visual quality when it comes to upsampling techniques. The following source provides an insightful quantitative analysis between different upsampling techniques: https://pixinsight.com/doc/docs/InterpolationAlgorithms/InterpolationAlgorithms.html#__Upsampling_Examples__

Chosen Upsampling Technique

We have chosen to use nearest-neighbor interpolation for upsampling during the image de-tiling stage. This is mainly because of its performance benefits as it is the least computationally complex option. OpenJPEG also uses this upsampling technique, so if we used one that is more complex it would most likely lead to slower results in comparison to OpenJPEG which we will be benchmarking against. It may result in blockier image quality, but as our main focus is performance and throughput, this made the most sense to us.

Tiles and Tile-parts

Each coded tile is represented by a sequence of packets. The rules dictating the order that the packets of a tile appear within the codestream is specified in more detail in Annex B.12 of **ISO 15444-1**. It is possible for a tile to contain no packets, in the event that no samples from any image component map to the region occupied by the tile on the reference grid. For the NPJE compression profile, there is only a single tile-part per tile. Our current code implementation functions for the NPJE profile only at this point.

De-tiling Process

1. Perform ceiling division on both the image and tile widths, as well as the image and tile heights to get the number of tiles in the image.

2. Loop through the tiles in both the x and y direction and process each tile as you loop through.
    a. For the NPJE compression profile you will only need to process the tile as a whole. For other compression profiles – EPJE, etc., there may be more than one tile-part per tile. In this case you will also need to process each tile-part within the process_tile function.
3. Call the nearest_neighbor_upsampling function to create the upsampled image as a matrix, based on the upsampling factor provided in the function.
4. Free the memory allocated for the decompressed image and return the final upsample image matrix.

Function Overview

Implemented in detiling.c

Helper Functions

- The ceiling division function computes the ceiling of the division of two integers, ensuring that the result is rounded up to the nearest integer.

Tile Processing

- This function processes an entire tile and updates the decompressed image accordingly.
- It iterates through each element in the tile and copies its value to the corresponding position in the decompressed image. Since there is only one tile-part per tile in the NPJE format of JPEG-2000, this function processes the singular tile-part within the tile.

Nearest Neighbor Upsampling

- This function performs nearest neighbor upsampling to bring all codestream components to the same spatial resolution.
- It calculates the target dimensions for the upsampled image based on the original width, height, and the specified upsampling factor.
- Memory is allocated for the upsampled image.
- Nearest neighbor upsampling is applied, where each pixel in the target image is assigned the value of the nearest pixel in the original image. If the upsample factor passed into the function is 1, then no upsampling will occur.
- The resulting upsampled image is returned.

De-tile Image

- This is the main function responsible for reconstructing the decompressed image from the tiled data.
- It extracts relevant information from the JPEG-2000 header, such as image dimensions and tile sizes.
- It calculates the number of tiles in both the horizontal and vertical directions.

- Memory is allocated for the decompressed image.
- It iterates through each tile, retrieves the tile data, and processes it using the process_tile function.
- After processing all tiles, it performs upsampling on the decompressed image using the nearest_neighbor_upsampling function.
- Finally, it frees the memory allocated for the decompressed image and returns the upsampled image.

Component Transform

Overview

This stage of the decoder returns the image data back to its original image form so that it can be viewed after being fully decoded. This is often used to color correct an image. The encoded JPEG-2000 images are stored in the YCbCr format, so we need to convert the image to the traditional RGB colorspace. To do this, the formula below is applied to the image. Note that this stage expects three separate image components (Y_0, Y_1, Y_2) and outputs three separate image components (I_0, I_1, I_2). These components will need to be combined for the final image to be viewable.

$$I_1(x, y) = Y_0(x, y) - floor(\frac{Y_2(x, y) + Y_1(x, y)}{4})$$
$$I_0(x, y) = Y_2(x, y) + I_1(x, y)$$
$$I_2(x, y) = Y_1(x, y) + I_1(x, y)$$

Equations ISO-G-12 through ISO-G-14 Inverse RCT Transforms

Implementation

Implemented in component_transform.c

Benchmarking

Overview

In order to know that we have achieved a performance improvement, it is necessary to understand the performance of OpenJPEG, and compare it to the performance of our implementation. Furthermore, benchmarking can help isolate specific steps or processes and help determine areas for improvement. We can compare the strengths and weaknesses of various implementations for types of images, and we can ensure consistency of results with the benchmarking. Here, we detail the benchmarking software in its current state and detail some future modifications that would ensure a properly tested system.

Tools

We are currently using perf, a performance analyzing tool for Linux. We chose Perf for its simplicity, capabilities, and limited dependencies This tool supports obtaining exact statistics from the CPUs Performance Monitoring Unit (PMU) and collecting statistics about the processing associated with certain functions. This tool also has support for multiple threads.

Known Implementations to Benchmark

Several of the implementations listed in **Table 26** function by either converting the image into another format (such as PNG) or by making a viewer with the image in it. As such, when benchmarking an existing implementation, care must be taken to only benchmark the portion of the process that decodes the image, and not any additional work such as converting the image or rendering it. For example, we have modified OpenJPEG to accept a possible output type of .none, which simply ends the program as soon as the image is decoded rather than converting the image and outputting to a new file with that format.

Table 26, JPEG-2000 Decoders

Decoder	**Language**	**Comments**
OpenJPEG	C	What the client specifically asked us to benchmark.
JJ2000	Java	
Jasper	C++	
Our Decoder	C	Necessary for showing improvements
Kakadu	C++	Closed source, may not be able to obtain. Claims very high speeds.
Grok	C++	Claims improvements from Kakadu in certain cases

Methods
To measure the metrics we will use the exact statistics obtained from the PMU. To determine possible improvements we plan to collect statistics about the processing associated with certain functions.

Environment
We are using a virtual machine on a type 1 hypervisor with Rocky Linux and no GUI. The machine has 64 GB of memory, which matches our clients. The machine also has a 16MB cache.
4 cores of the Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10 GHz (not hyperthreaded)
This environment has dedicated resources and no one else will be using them, this should maximize reproducibility of the results.

Additionally, one should consider running the commands as sudo so perf can collect statistics from the kernel, or changing the kernel's perf paranoid setting. The kernel could also be compiled with debugging, which may provide additional information to the benchmark.

If you wish to obtain a virtual machine, you should work with the sysadmin. They may need to change certain parameters on the virtual machine in order for the environment to support perf or any other type of benchmarking.

Metrics
The clients machines have 64GB of ram, and they deal with images in the 10s of GBs. As such, we find speed to be more important than memory efficiency. We are also interested in how the size of the image influences the speed of decoding. Therefore, our primary metrics are speed (seconds) and throughput (bits/second). When choosing between solutions with similar speeds, we could add secondary metrics such as memory usage. Several more metrics can be chosen based on what perf stat is capable of collecting.

Parameters
We will vary the size of the images to see how it influences the decoding speed, and we will vary the number of threads (if applicable) allocated to the program and see how that influences speed. Therefore, our parameters are image size (GigaBytes) and number of threads.

The client primarily cares about tiling. Because NPJE, our current focus, has only one tile, our only method at the moment is decoding the whole image.

Interpreting Results
Depending on the statistic, the unit can vary and whether maximizing or minimizing the statistic is best. For example, a good decoder will maximize CPU usage, because this means little time is spent idle during operation and the decoder implements parallelization well. Conversely, factors such as time should be minimized.

Design

Exact statistics:

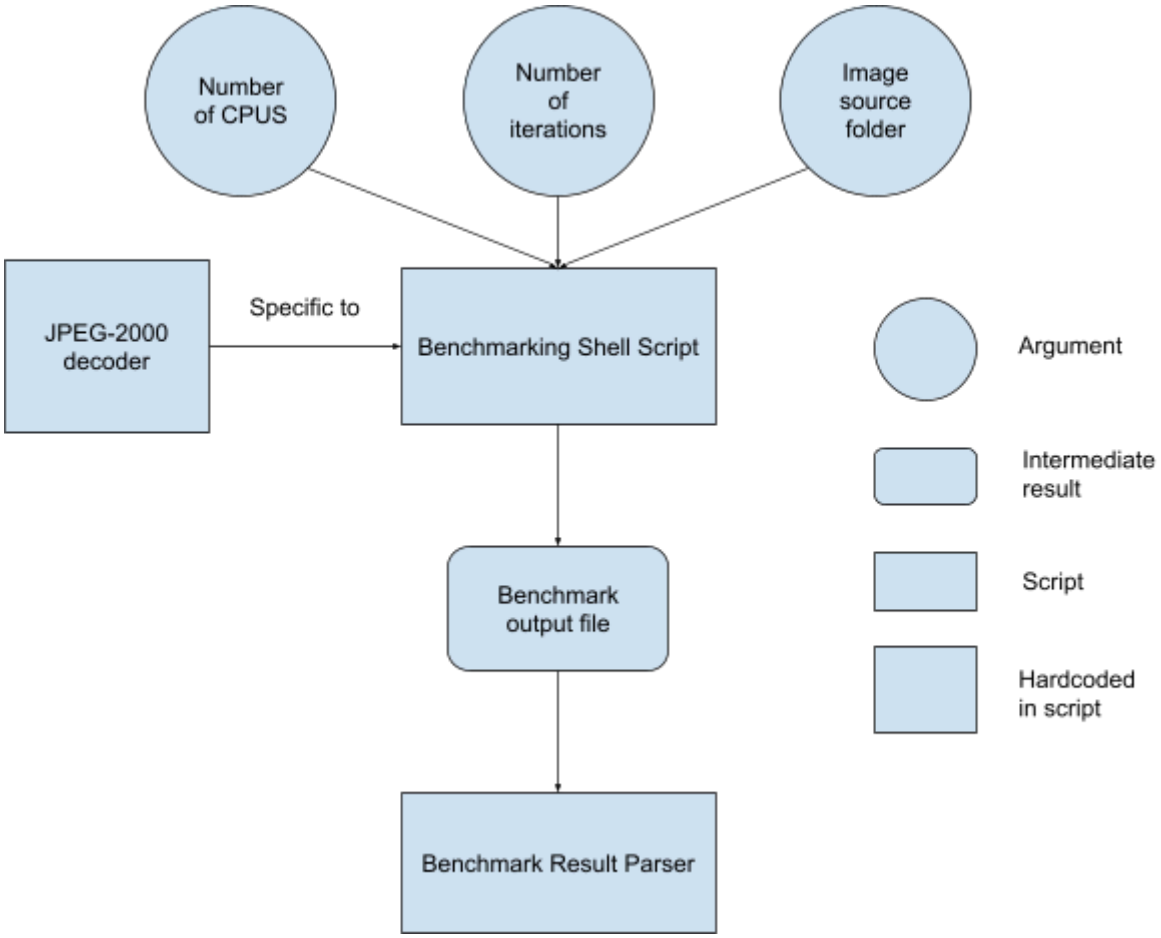


Figure 9, Diagram of Overview of Exact Statistics Benchmark

Our benchmark can be run from the command line with arguments for the number of iterations, source image folder, and maximum number of CPUs. The benchmark will then run perf stat on every image in the folder for each number of CPUs for each number of iterations for a set JPEG-2000 decoder to collect the exact statistics. Because the syntax varies depending on the implementation, a shell script must be created for each implementation. A machine readable file is generated that can then be sent to our benchmark result parser which can read in all the information from the benchmark.

We allow specification of CPUs, an image source folder, and number of iterations to help with collecting data. Despite perf collecting exact statistics, factors such as branch prediction or underlying operating system processes may introduce some variation in performance, so we run multiple iterations to minimize the variation. The tests will start running on only 1 cpu. Additional CPUs will be added until the number of CPUs specified is reached in order to see how results differ with parallelization. The image source folder allows for testing multiple images at once to examine how factors such as image size can influence performance.

The output from the benchmark is concatenated into a file with machine readable format. A parser exists in the repository to parse the information from the machine readable format. Thus a user has significant potential to automate benchmarking for the exact statistics.

Recording:

Our process for recording is much less defined, but follows two steps.

1. Use perf record with a command (presumably decoding an image) to collect information about the process executed.
2. Use perf report to calculate statistics and report information in a hierarchical structure that makes finding overhead for specific functions simple.

When doing the perf record, one should be aware that a very large amount of data is generated, so when decoding medium or larger images, sampling should be used rather than recording every bit of the execution. While less precise, the system could easily run out of storage if sampling is not used. Furthermore, to get information like the overview of specific functions requires getting information about the call graph. We currently use dwarf for that.

For reporting, the viewer for reporting has a built-in help that shows a variety of commands that assist in looking through the results.

Dataset

Considerations

We aimed to collect a dataset with a wide variety of image types (grayscale and 4 channel) and a wide variety of image sizes.

Table 27, Dataset Details

Images	Size	Obtained
Small	<100MB	8
Medium	>100MB	11
Large	>1GB	1
Very Large	>10GB	1

Future Improvements

Overview

We have a capability to automatically parse some of the results of the benchmark for the exact statistic counting. This means that if the data is forwarded to other tools, we could make graphs and find aggregate statistics automatically.

For the perf recording, several tools exist that give improved visualizations, such as Flame Graphs. A future team should consider using different tools to assist with understanding the results. Additionally, the perf recording for multiple threads records the whole system, so if the team can find a way to record just the decoding process or figure out which system resources are used by the process and which by other processes.

Instrumentation (adding custom events for the benchmark to monitor) can assist with understanding performance.

Testing

Overview

Our primary goal with testing is to ensure the system's correctness, while also assessing the system's compatibility and completeness. The system's core decoding functions are the primary focus of testing. Our main testing strategy was to use white box testing to assess unit methods' functionality, and black box testing to assess the system's overall functionality, by having it attempt to parse through and decode an actual image.

Test Plan

We created detailed unit tests for the different decoder stages. For example, the image de-tiling stage is located in **detiling.c** and the corresponding tests for that section of the decoder are in the **detiling_test.c** file. These tests allow us to determine these sections of the decoder are functioning properly.
If there are sections of the decoder where a detailed unit test suite is not feasible, we compare the output to a ground truth. This ground truth will be based on a generated image with the specified image format.
Once all parts of the decoder have passed the specified tests, then the parts will be integrated and tested together until the entire system is complete. Subsequent more complex generated images should be used to confirm the integrated system, once you reach this point in testing. Currently, the entropy decoding section is not complete and passing functional unit testing. This will be a main focus before being able to work on full system testing.

Test Execution

Tests are executed by generating and running test files like mentioned above in the Test Plan section. These test files are separate from the execution, and the execution of the program does not depend on the test files. A test file is created for each decoder part, and a separate test file will be used for various integration and system tests.

Testing Resources

- Unity is our software testing framework
- GeoExpress has been used to generate expected JPEG-2000 images in the NPJE format from various other image formats (typically .tif files found from EarthExplorer)
    - The GeoExpress software is mentioned below in the References section
    - When using the GeoExpress software, these are the settings used to generate a JPEG-2000 image in the NPJE format:
        - Format: JPEG 2000 (Part-1)
        - Compression: Lossless
        - Advanced → Format-Specific
            - Profile: NPJE
            - Precincts: Uncheck box to use

        - Uncheck use 9-7 wavelet
    - EarthExplorer is also listed below in the References section where you can use to download many different satellite images of differing sizes
- VirtualBox has been used to test compliance with the client's environment
- JJ2000 is a Java implementation of a JPEG-2000 decoder that can be used to compare different functionality of the decoder to our implementation
    - `-i "D:\School\JPEG2000\m_3811946_ne_11_h_20160803_20161004.jp2" -cdstr_info` → used as the run configuration. If you run it without a configuration, it will tell you which flag to add for help, and then from there you can see all the flags
    - `-i` is the input file flag, the `-cdstr_info` flag gives extra output
    - This software is listed in the References section below where you can access its GitHub repository to download

Repository Overview

Top Level

Github Repository Link: https://github.com/rhit-kosikoaj/JPEG-2000/tree/main
At the highest level, the Github repository for this project is separated into three folders. Our work is localized in the JPEG-2000-master directory. We have included two other implementations of JPEG-2000 decoders as reference material, namely JJ2000 (a Java implementation) and openjpeg (a C implementation).

JPEG-2000 File Structure

Within our project's master directory, we have included the various code files that make up our implementation, a selection of JPEG-2000 images to test the system on, and the files necessary for Unity and our benchmarking program to function. Unity is our unit test framework, and our codebase contains several examples of unit testing with Unity. Currently, our system is divided into multiple files representing each step of our decoder. Usually, each decoder step corresponds with a file; for example, the wavelet decomposition step is confined to wavelet.c. The exceptions are the codestream and progression parsing which are combined into decoder.c, and the entropy decoding step which is separated into three files: mq_decoder.c, tag_tree_decoder.c, and coefficient.c. There are also several files that contain various structs we use, namely the matrix and various parsing structs. These structs are included in the files matrix.c and parsing.c, respectively. The entry point of the program is JPEG2000_decoder.c, which requires a file path to the encoded image.

Benchmarking File Structure

Within our project's master directory, we have a folder titled "BasicBenchmark". That folder contains demo scripts for benchmarking and a sample output.
The myFirstBenchmark.c is useful for a very simple, multithreaded application to benchmark, and thus useful for learning how to benchmark with. myFirstBenchmark.c calls otherProgram.c. The makefile can make these two programs.
There are three example benchmark scripts. Two of these scripts are for perf record, one of which employs sampling. The perf record scripts have a comment showing how to bring up the result viewer afterwards. One script is for perf stat that runs on myFirstBenchmark.
Our implementation of a benchmark for OpenJPEG is available as a script: this is the script from the Diagram of Overview of Exact Benchmark Statistics. A sample output from this script is available in the benchmark directory. The parser, from the same diagram, is also available as a python file.
All of the scripts that take arguments have an example in their files for how to run them.
All of the scripts generate output, and the output goes into the outputs directory created within the benchmarking directory.

Performance Improvements

Overview

There are many areas for performance improvements within the current decoder prototype. The most computational power is going to be used during the Entropy Decoding stage, and therefore there will most likely be the largest area for performance improvements in that stage of the decoder. There are also a few improvements we have already discovered that can be made to potentially increase the performance of the decoder.

Future Improvements

Remove #pragma pack(1) from the JPEG-2000 header parsing code and other headers.

- This flag removes gaps in the structs making it easier to load and access fields in the structs, reducing memory at the cost of some performance. This was added to increase development speed, but it does have a negative impact on the performance.

Multiprocessing using openMPI or a similar library.

- The format of JPEG-2000 is designed to be highly parallelizable, and we can attach each tile or portion of a tile to a separate core in order to improve performance . After the parsing stages, each processor should need minimal global information and should predominantly only need information from the tile headers to decode each packet and reconstruct the tile.

Alternative MQ Decoder implementations.

- There appears to be multiple implementations of the decoder or similar decoders with some minor performance improvements. The JJ2000 implementation has custom decoder methods that it claims makes the decoding faster than the standard implementation. There may be further improvements possible.

Memoization

- The client previously stated that they do not believe main memory would be a large concern for the decoder. Therefore it would be effective to try to keep as many results as possible stored in main memory and never offloaded to a disk. This may be impossible for particularly large images, but there may be a pattern of consistent calls to the MQ decoder or wavelet transform where we can automatically return the result without doing the same calculation on the same input.

Lessons Learned

Overview

This section of the documentation goes into detail about the various lessons our group has learned throughout this project. This includes the most and least important aspects of the JPEG-2000 decoder, and where the next group should spend the majority of their time.

Most Important

- **Entropy Decoding:** This is the largest stage of the JPEG-2000 decoder, and the one that is currently least implemented. This will be a large focus for the following group, and as it is one of the most computationally complex parts of the decoder it will most likely greatly affect the overall performance of the JPEG-2000 decoder. By spending more time on this stage and refining it, this can result in better performance for the decoder overall. Once this stage is functional, you will be able to integrate each stage of the decoder and have a fully functioning prototype. This is the first step in being able to make any additional performance improvements to the decoder. We recommend reverse engineering the JJ2000 implementation as a starting point for this stage, with the ISO being used primarily as an additional reference since it lacks some details on the decoding aspects.
- **Benchmarking:** With more focus spent on benchmarking, there is a greater chance to find many performance improvements within the JPEG-2000 decoder implementation. By benchmarking separate stages of the decoder, you will be able to determine which stages take the most computational power and focus on those. The Open-JPEG implementation has a difficult coding style to manually analyze for areas of improvement, but it is not difficult to hook up to PERF for automatic analysis of poorly performing areas.
- **Testing:** It will be important to thoroughly test each stage of the decoder, as well as the decoder as a whole once each stage is integrated together on many different images. This will not only confirm that it is functional for more than one type of image, but it may lead to differing performance results depending on the size / type of image. We opted for an incremental approach when testing, that is we moved on to the next stage when a specific NPJE image functioned on it. We believe this is the most effective route as it is difficult to determine whether an individual stage failed, especially the later ones, without viewing the decoded image. Then once a very simple image can be successfully decoded, add the functionality for more complex images in each stage.

Less Important

- **Image De-Tiling:** When we noticed there was a lack of documentation on the image de-tiling stage in the official JPEG-2000 documentation we had been provided, we spent a lot of time trying to research on our own how this stage of the decoding worked. However, we have provided an explanation of the findings we have come across so this

should result in a lot less time spent on research for additional documentation on this stage of the decoder. In addition to this, the image de-tiling stage is a lot less in depth in comparison to some of the other stages of the decoder, which are more important to spend time on. Stages after the Wavelet transform stage are primarily “cleanup” and rearranging stages; they often don’t have complex functionality.

- **Component Transform:** Based on our research, we found that the final component transform would be trivial given the rest of the image is decoded properly. Additionally, an incorrect component transform would not invalidate an image since the underlying structures would still be correct. Therefore, it makes the most sense to focus on the other stages of the decoder instead.

Potential Future Roadmap

Overview

We propose a future roadmap based on the features discussed with the client and from the original project proposal. This map prioritizes incremental functionality before any performance improvements.

Next Year Road Map:

1. Entropy Decoding
2. De-Tiling Verification
3. Component Transformation Verification
4. Verify whole decoder on larger and differently encoded images
    a. NPJE focus, different encodings
    b. EPJE add on
5. Add PERF hooks for benchmarking implementation
6. Gather initial benchmark comparisons between implementation and OpenJPEG
    a. Consider benchmarks for JJ2000 and other implementations
7. Design Improvements
    a. Multi-processing
    b. Memory Improvements
    c. Entropy Decoding improvements
        i. Possible examples in JJ2000
8. GPU support

Citations and Resources

Citations

Information technology - JPEG 2000 image coding system - Part 1: Core coding system, ISO/IEC 15444-1:2019.

BIIF Profile for JPEG 2000, Version 01.20 (BPJK01.20), Feb. 28 2023

P. Schelkens, A. Skodras and T. Ebrahimi, *THE JPEG-2000 SUITE*. Chichester, United Kingdom: John Wiley & Sons Ltd, 2009.

Resources

Relevant Standards

Useful:
ISO 15444-1:2019: Part one of JPEG-2000 standard, which contains all of the basic specifications needed for the decoder.
ISO/IEC BIIF PROFILE BPJK01.20 (28 February 2023): Contains the specifications of NPJE and EPJE.
This might be superceded by a version from 27 October 2023, check with the client.
The 27 October version is available from
https://nsgreg.nga.mil/doc/view?i=5468&month=10&day=10&year=2023

Less Useful:
ISO 15444-4:2021: Part four of JPEG-2000 contains information about conformance testing. We did not find this standard particularly useful.
ISO 15444-5:2021: Part five of JPEG-2000 contains information about reference software. This includes an overview of several of the implementations. We did not find it particularly useful.

Textbooks

Useful:
The JPEG 2000 Suite, Peter Shelkens, Athanassios Skodras, Touradj Ebrahimi
ISBN-13: 978-0470721476
Contains in depth explanations of several concepts in JPEG 2000. We used this textbook the most.

Less Useful:
JPEG 2000 Image Compression Fundamentals, Standards and Practice, David Taubman, Michael Marcellin

ISBN-13: 978-0792375197
Also in depth explanations of several concepts in JPEG 2000.
https://link.springer.com/book/10.1007/978-1-4615-0799-4 (Should be available through Rose-Hulman)

Papers

Useful

The JPEG-2000 Still Image Compression Standard:
High level overview of the standard
https://www.ece.uvic.ca/~frodo/publications/jpeg2000.pdf
JPEG2000 - A Short Tutorial
Gaetano Impoco
https://cse.buffalo.edu/courses/cse725/peter/Impoco_2004.pdf
Overview of standard with several algorithms provided

Less useful

An Overview of JPEG-2000:
Michael W. Marcellin , Michael J. Gormish , Ali Bilgin , Martin P. Boliek
https://uweb.engr.arizona.edu/~bilgin/publications/DCC2000.pdf
High level overview
High Performance Scalable Image Compression with EBCOT:
David Taubman
Gives an overview of EBCOT, which has influenced the design of the entropy decoding
https://ieeexplore.ieee.org/document/817132 (not available for free)
JPEG2000: Standard for Interactive Imaging:
David s. Taubman and Micheal W. Marcellin
https://pds-engineering.jpl.nasa.gov/wp-content/uploads/documents/standards/SCRs/3-1003_jpeg2000/Taubman_and_Marcellin.pdf
Overview of JPEG-2000
JPEG 2000: fast access to large grayscale images:
https://www.spiedigitallibrary.org/conference-proceedings-of-spie/6943/1/JPEG-2000-fast-access-to-large-grayscale-images/10.1117/12.777178.full
Should have access through Rose-Hulman
Fast JPEG 2000 decoder and its use in medical imaging
https://www.researchgate.net/publication/3415644_Fast_JPEG_2000_decoder_and_its_use_in_medical_imaging
Hardware Implementation of Tag Tree in JPEG2000 Encoder
https://www.ijert.org/research/hardware-implementation-of-tag-tree-in-jpeg2000-encoder-IJERTV3IS110542.pdf
Coarse-to-Fine Textures Retrieval in the JPEG 2000 Compressed Domain for Fast Browsing of Large Image Databases
https://link.springer.com/chapter/10.1007/11848035_38 pg 282-289
Should be available through Rose-Hulman

Websites

Useful:
https://jpeg.org/jpeg2000/documentation.html, lists several resources about the standard.
https://earthexplorer.usgs.gov/ Useful for obtaining satellite images for benchmarking
https://link.springer.com/referenceworkentry/10.1007/978-0-387-78414-4_99 another high level overview of JPEG-2000 standards. You should have access from Rose-Hulman to download.

Less Useful:
https://nsgreg.nga.mil/JESC-approved.jsp Where we got the BPJK standards from.
https://pixinsight.com/doc/docs/InterpolationAlgorithms/InterpolationAlgorithms.html#__Upsampling_Examples__ Provides image upsampling examples, useful for image de-tiling upsampling technique
https://jpeg.org/jpeg2000/htj2k.html You may be interested in HTJ2K, which is a much faster version of JPEG 2000, but does not have full backwards compatibility.
https://www.openjpeg.org/ Site for OpenJPEG
https://link.springer.com/referenceworkentry/10.1007/0-387-30038-4_117#Sec3_0-387-30038-4_117 high level overview of JPEG-2000 standards. You should have access from Rose-Hulman to download.
https://www.ece.mcmaster.ca/~shirani/multi08/jpeg2000part1.pdf overview from a university
https://www.fit.vut.cz/research/publication-file/12302/postprint.pdf high level overview
https://www.cs.cmu.edu/~guyb/realworld/paper_ieee_ce_jpeg2000_Nov2000.pdf high level overview
https://xavirema.eu/mq-coder-in-matlab/ MQ Entropy Coder implemented in Matlab

Implementations:
See **Table of JPEG-2000 Decoders** in the Benchmarking section.
All official implementations from the ISO are available from
https://standards.iso.org/iso-iec/15444/-5/ed-3/en/

Useful
JJ2000: Available in our repo, from SVN https://code.google.com/archive/p/JJ2000/ and a (probably unofficial) fork on GitHub
https://github.com/Unidata/JJ2000/tree/main?tab=readme-ov-file
OpenJPEG: https://github.com/uclouvain/openjpeg?tab=readme-ov-file

Less Useful
Jasper: https://github.com/jasper-software/jasper
Kakadu: https://kakadusoftware.com/
Grok: https://github.com/GrokImageCompression/grok Claims significant performance improvements over OpenJPEG in certain areas. This one may actually be very useful.

Software

Useful

JPEG2000 Github: https://github.com/rhit-kosikoaj/JPEG-2000/tree/main

Geoexpress 9-5, https://www.extensis.com/support/geoexpress-9-5: Useful for converting images. Has a trial license, so you can only use it for 30 days once you download it.

Less Useful

GPU library for JPEG 2000: https://developer.nvidia.com/blog/accelerating-jpeg-2000-decoding-for-digital-pathology-and-satellite-images-using-the-nvjpeg2000-library/

C# wavelet library: https://github.com/codeprof/TurboWavelets.Net/blob/master/TurboWavelets/Biorthogonal53Wavelet2D.cs

Appendices

Appendix A:

JPEG-2000 Project Plan

Andrew Kosikowski, Campbell Garvin, Seth Marcus, Taylor Goldman

Table of Contents

Version Table

Version #	Main Changes	Date
1	Initial Draft	9/26/2023

Version #	Main Changes	Date
1.1	Added Risk Information	9/29/2023
1.2	Added Team Processes and Risk Mitigation	10/4/2023
1.3	Further expanded Risk Mitigation and Team Processes	10/17/2023
1.4	Added to Project Timeline	10/29/2023
1.5	Tracking and Control Mechanisms Revisions	11/10/2023
1.6	Project Estimates and Team Processes	1/18/2024
1.7	Added Design Overview	2/8/2024
1.8	Added Testing Plan	2/9/2024
1.9	Minor changes to Project Timeline	3/22/2024
2.0	Added Continuous Project Details Throughout	4/19/2024

Introduction

This project is a JPEG-2000 Decoder with a benchmarking suite. At Johns Hopkins University, they often work with science imagery from NASA. Many images they use are JPEG-2000s which is not a widely adopted standard, and there are minimal open source tools and image decoders as available resources. To support working with this file format, they want an optimized image reading capability. We will develop a C library to read JPEG-2000 image files, as well as a benchmark suite to characterize and stress test the image-reading capability developed. We will develop this library using a spiral focused design methodology, where we deliver prototypes as they are finished. This project will be continuous for another group to work on next year, and make continuous improvements on the decoder's performance.

Goals and Objectives

The main goals of the JPEG-2000 Decoder and Benchmarking Project are:

- Allow users to read JPEG-2000 image files
- Stress test the image reading capability developed
- Optimize the speed of executing image reading schemes
- Establish a baseline of JPEG-2000 performance

Project Scope

Here is a list of the main features for our project, provided by the project description and our client. The ones in red are not currently implemented, and will be part of the focus for the next group working on the project.:

1. Read image tiling with overlap
2. Read random chipping through the image
3. Preview image at lower resolutions
4. Benchmarks to measure image processing capabilities
5. Support lossless encoding
6. Support numerically lossless encoding
7. Loads image data into memory

Stakeholders

- Researchers at Johns Hopkins Applied Physics Laboratory
- Employees at Johns Hopkins Applied Physics Laboratory

Project Estimates

For our project, we have been tracking most of our initial progress on a Trello board, where we estimate the amount of time a certain task may take. We have been having team meetings twice a week, in addition to our advisor and client meetings to make sure we continue on track and are documenting our progress we are making.

Project Timeline

Fall Quarter:

- Gather initial software requirements from client
- Research the JPEG-2000 standard
- Begin research on JPEG-2000 compression profiles
- Initial design for first JPEG-2000 decoder prototype
- Begin implementing parsing for JPEG-2000
- Rough decoding implementation for JPEG-2000

Winter Quarter:

- Iteration 1: November 27 - February 19
    - Create benchmark tests
    - Create acceptance tests
    - Develop a functional prototype that supports the NPJE profile
    - Identify processing bottlenecks in the system

Spring Quarter:

- Continuous Development: March 4 - April 8
    - Continuously improve decoder
    - Focus on removing previously discovered bottlenecks
    - Throughput exceeds OpenJPEG by 5%
    - Expand upon benchmarking tests to reflect this improvement
- If decoder is unfinished, plan for a group to continue development next fall
    - Dataset of JPEG2000 images for testing
    - Benchmarking suite
    - Decoder in progress
    - JPEG2000 documentation on our current implementation
        - JPEG2000 Documentation

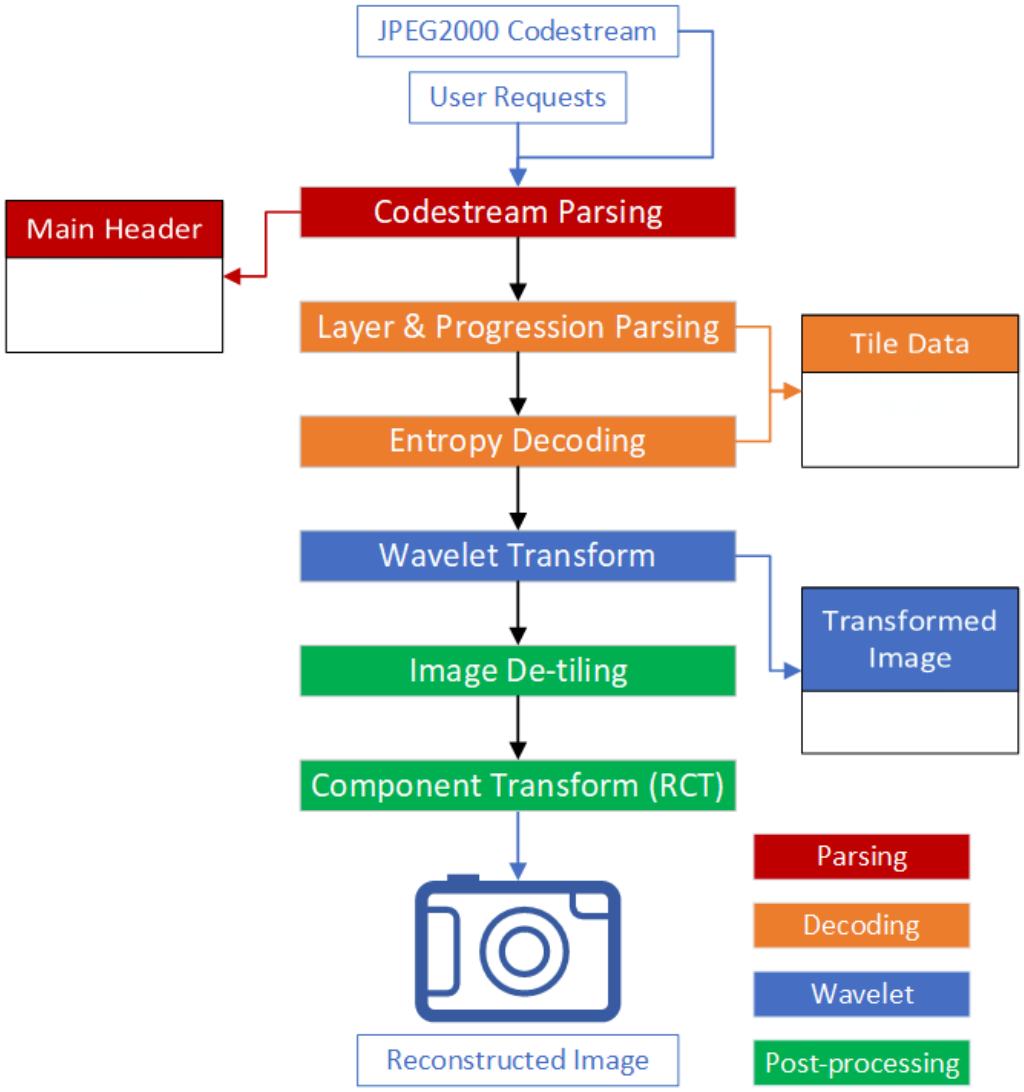
Software Requirements

These are the following software requirements for the project gathered throughout communication with our client.
Needs, Features, Requirements

1. C/C++ library to interface with to support file reading for JPEG2000 on Linux.
2. JPEG200 file reader that optimizes the speed of executing each image reading scheme.
3. Benchmarks measuring the following: speed, throughput, image size, compression, location.
4. Ability to be wrapped in Python or MATLAB code.
5. Highest efficiency loading into memory.
6. The decoder supports NPJE JPEG 2000 profiles.
7. Takes in a file pointer rather than a file name, for compatibility with multiple file formats
8. Library returns a pointer to memory for image data
9. The library supports lossless encoding, including numerically lossless encoding
10. The library can read images in multiple colorspaces, including grayscale
11. The library will be comprised of free to use software

Design Overview

General Design

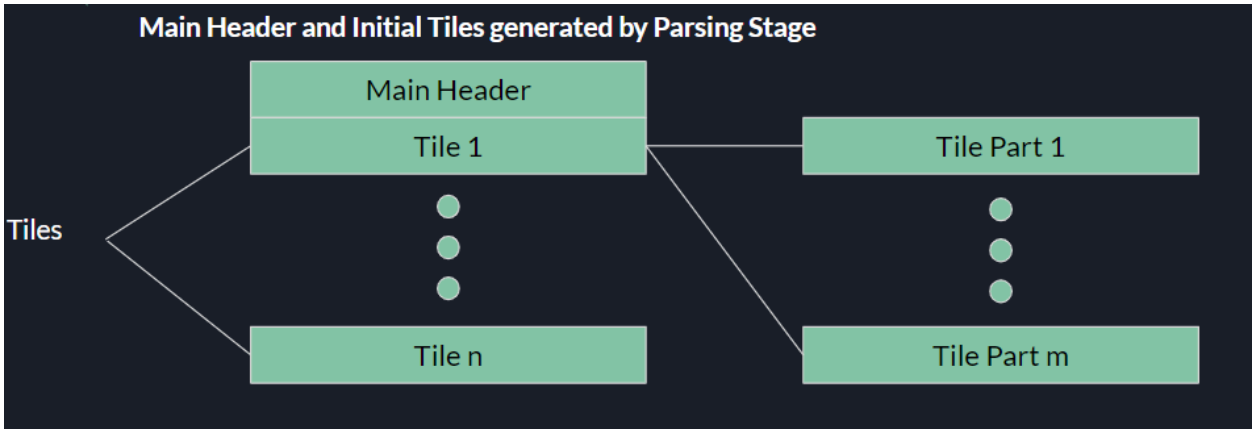


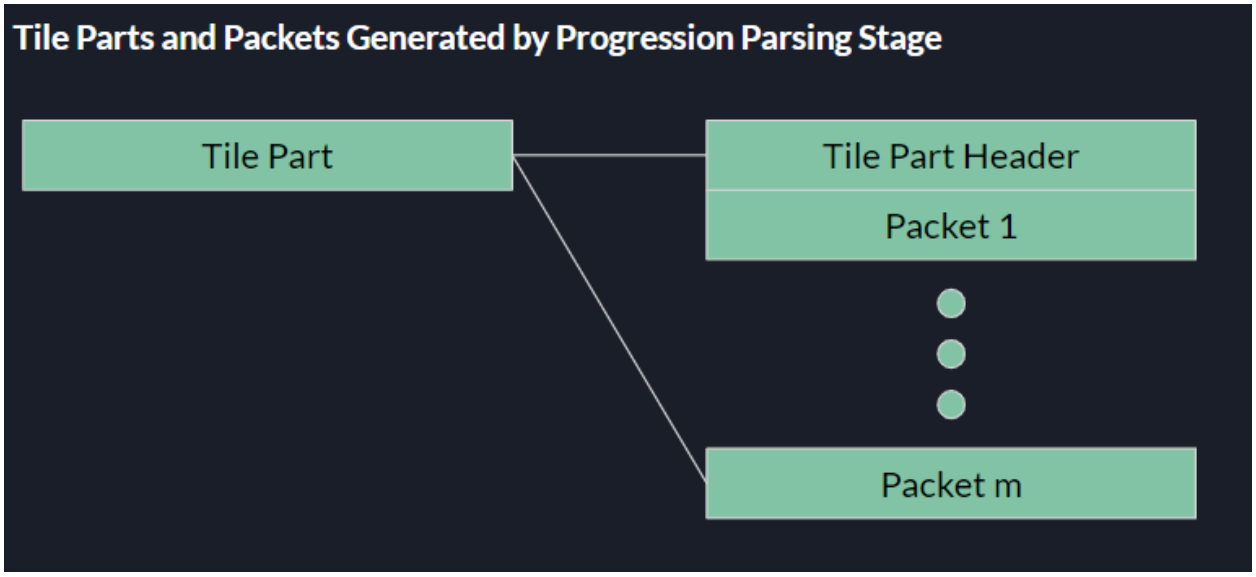
Stage Descriptions

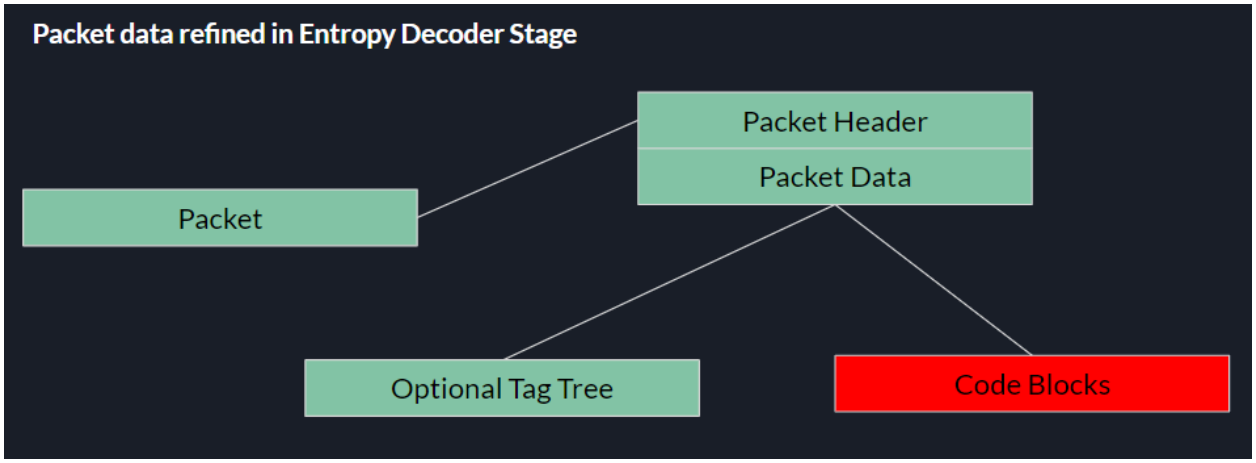
1. Codestream Parsing
    a. Determines which part of the codestream need to be decoded based on user requests
    b. Will parse through main header data and create a Main Header Struct
    c. Will determine characteristics about the decoded image which get outputted in the form of a JPEG2000 Struct
2. Layer & Progression Parsing

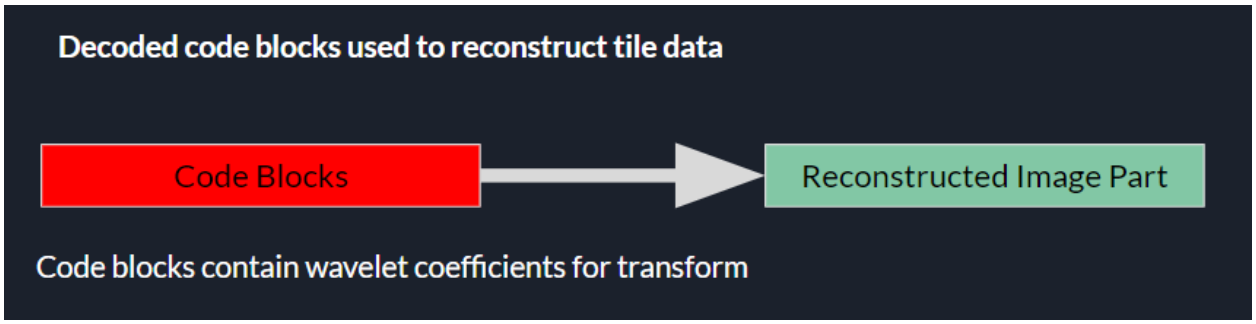
    a. Fills the JPEG2000 Tile Data structs in a specific order based on the user requests
3. Entropy Decoding
    a. Parsed Tile Data structs gets properly ordered for when the image is reconstructed
    b. Parsed data begins to be decompressed and reverts any arithmetic coding to produce wavelet coefficients
4. Wavelet Transform
    a. Does the inverse of a reversible wavelet transform function to support lossless decoding
    b. Transforms wavelet coefficients back into image component data per tile
5. Image De-tiling
    a. Image tiles are re-assembled spatially to form the reconstructed image
6. Component Transform
    a. Dependent on color scheme of image
    b. Applied to return image data back to its original color domain

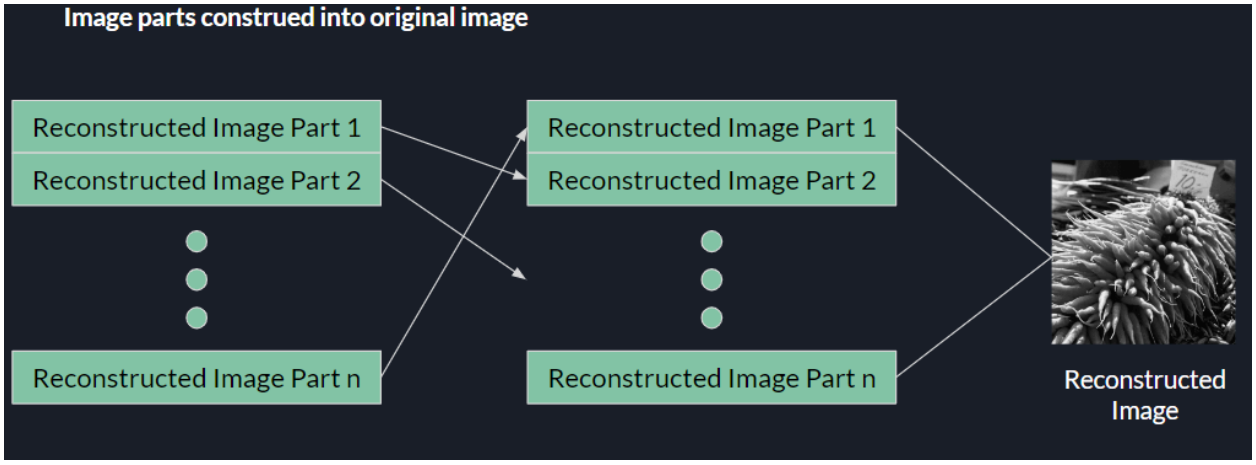
Basic Implementation Data Structure




Tile Parts and Packets Generated by Progression Parsing Stage
Tile Part
Tile Part Header
Packet 1
Packet m


Packet data refined in Entropy Decoder Stage
Packet Header
Packet Data
Packet
Optional Tag Tree
Code Blocks


Decoded code blocks used to reconstruct tile data
Code Blocks
Reconstructed Image Part
Code blocks contain wavelet coefficients for transform



Design Improvements

The following improvements can be made by the group continuing the work on the project. These are current performance improvements that can be made to the decoder:

- Entropy decoding stage takes the most computing power based on benchmarking data
    - This will be our main focus for optimization
- Image de-tiling upsampling method was chosen to utilize the least amount of CPU power to increase performance
- CPU memory utilization is an area of focus throughout the design
- Changes to the way we are parsing main header data
    - We are currently using struct padding to more easily read the bytes in the JPEG2000 codestream, which takes more computing power and will ultimately result in slower performance
    - When making improvements to our design we will work on removing this struct padding and use a more efficient method for our parsing stages of the decoder

Benchmarking Overview

Benchmarking Plan

We intend to establish a baseline for OpenJPEG and other decoders on how fast they decode in seconds and how the size of images influences the decoding speed.
We will use perf to measure both exact statistics from the CPUs Performance Monitoring Unit and show overhead associated with functions to determine where we might target for improvements.

Risks

We will be compiling a list of the top risks we may encounter throughout our project, with associated risk values. In addition to this, we will have a plan for risk mitigation if risks with a high risk value were to occur. This is to ensure that we are proactive and will not be set back due to risks.

Risk Table

This table includes the top risks we have identified for this project, along with their calculated risk value. The value for each risk is calculated based on the probability that it will occur, and how impactful it will be on our overall success for the project if it does occur. (Risk Value = Probability * Impact).

Risk	Probability	Impact	Risk Value
During each iteration, prototype is not fast enough for the client's needs	65%	3	1.95
Poor documentation on existing JPEG standards delays design and development	35%	2	0.70
Information or implementation details we need may not be publicly available	50%	2	1.00
The team does not have enough time or other resources to finish the project	40%	2	0.80
Miscommunication of client needs for the project leads to inadequate design or prototype	30%	2	0.60
Client becomes hard to reach regarding important information	30%	2	0.60
A prototype takes longer to develop than expected	50%	3	1.50
Miscommunications between team members	15%	1	0.15
Requirements change after design and implementation have already begun	20%	3	0.60

Impact Values:

1. **Minimal**: This risk will have an eventual impact on the project, and/or the effect of the impact will be small. It could result in a small delay for the project, and/or it may cause minor issues on the quality of the project.
2. **Important**: This risk will need to be addressed, as it will impact our ability to finish the project on time, and/or have some noticeable impact on the overall success of the project. It may delay us, and/or result in a negative impact on the quality of the final product
3. **Critical**: This risk will need to be addressed immediately, and/or has a significant impact on our ability to deliver a successful product to our client. This can delay our project a large amount if not addressed, and/or could lead to an unacceptable level of performance.

Risk Information Sheets:

We created risk information sheets for the three risks with the highest risk value. Each sheet clarifies the risks and offers methods for mitigating and managing the risks.

Risk Sheet 1

Risk ID: RS1	Date: 10/17/23	Prob: 65%	Impact: Critical	Risk Value: 1.95

Description:

The client may have performance expectations that we are unable to meet with each of our prototypes. If this occurs, we may not be delivering a successful product to our client.

Refinement/context

Subcondition 1: Doesn't meet the benchmark performance test.
Subcondition 2: Client determines that the prototype is not fast enough.

Mitigation/Monitoring

1) Making sure to run benchmark performance tests on our software often.
2) Discuss the performance of our prototypes and benchmark test results with the client weekly.

Management/Contingency Plan/Trigger

If our prototypes aren't meeting the benchmark performance test standards, we can focus on reevaluating our benchmark metrics and make sure they are realistic. If the client has higher performance expectations for our prototypes, we can also try to reevaluate our benchmarking to ensure that it will test more in line to what the client is looking for. We will likely need to dedicate more time as a group to the performance of the prototype we are working on. While doing this, we can look into other prototyping strategies that may result in better performance.

This may result in project delays, which we will need to discuss together to make sure we can meet to specifically address our prototype performance. We will also need to discuss potential project delays with the client, and how this will impact our overall project progress.

Risk Sheet 2

Risk ID: RS2	Date: 10/17/23	Prob: 50%	Impact: Critical	Risk Value: 1.5

Description:

A prototype of our design could take longer than expected to develop.

Refinement/context

Subcondition 1: Unexpected complexity delays iteration
Subcondition 2: Team does not have enough time because of other obligations

Mitigation/Monitoring

1) Researching in advance of parts
2) Allocating time for the project before other obligations can fill that time

Management/Contingency Plan/Trigger

If we feel that a section is not well understood we could reach out to professors at Rose-Hulman who may have knowledge on the subject. For example we could talk to ECE professors about the signal aspects of the standards, math professors on the statistical measures or equations, and CSSE professors on implementation issues. To help prevent this from happening in the first place we can make sure that we have examined a reasonable number of sources and implementations if possible before we go to make a part of the prototype ourselves.

If the team realizes we are not making progress or do not feel like we are allocating enough time to this project, we can all plan on set times to meet well in advance. That way everyone will have to plan their schedules around these set events and will not put anything else in those times making it impossible to meet. This could be in the form of regularly scheduled meetings or blocking out time for if we need emergency meetings.

Risk Sheet 3

Risk ID: RS3	Date: 10/17/23	Prob: 50%	Impact: Important	Risk Value: 1.00

Description:

Certain documentation regarding general information on the JPEG-2000 format or implementation details of specific JPEG-2000 standards may not be publicly available or obtainable.

Refinement/context

Subcondition 1: Some of the documentation may be hidden or unable to be found
Subcondition 2: Documentation is behind paywall and client or team is unable to pay for it

Mitigation/Monitoring

1) Rate potential documentation by importance and attempt to purchase only the most important documentation
2) Ask client for potential sources and contacts to obtain missing documentation

Management/Contingency Plan/Trigger

If we are unable to obtain necessary relevant documentation on JPEG-2000 files, we will need to delay and reevaluate the design process. We likely will need to ask the client for exact images so we can reverse engineer information on specific JPEG-2000 file implementations. We also will need to perform more testing using the OPENJPEG decoder to learn about the general process of decoding a JPEG-2000 image.

This will result in a delay of the project of up to a month as we try to determine the missing implementation and general information from the documentation. As we will be unable to construct an effective or perfectly functional prototype with missing information, we will need to reevaluate client meeting times and design review times to correspond with the additional delay in the general design phase.

Tracking and Control Mechanisms

Team Processes

Our team will focus on a spiral methodology for the project. There will be one main prototype of the final product with the goal of improving the computational speed and efficiency during continuous development sprints. The goal will be to develop a prototype and continuously improve it, while meeting with the client every other week until the client is satisfied with the above improvements. Once we have a benchmark available and a compatible prototype, we can begin to target specific areas for improvement based on what takes the longest. These evaluations can guide our focus for each specific spiral. Within each spiral, we will be meeting weekly and discussing our work completed for the week and what we are planning to work on the following week. These sprints will allow us to keep track of our work and make sure we are progressing at the rate we need to in order to finish our prototype for each spiral.

For ensuring code quality, the codebase will have automated tests on GitHub to verify that images can be successfully processed, decoded, and passed on to another program. Additionally, we will be following the standard Git workflow, requiring code segments to be added to a branch separate from main before being merged into the rest of the codebase. Any merges will require manual review by another team member. We will additionally have benchmark tests that will be run to determine the current speed and efficiency on a variety of test images. We use weekly meetings to assign and measure the progress of individual tasks, along with discussing key design and implementation details. The timeframe of our tasks will be tracked on Trello, along with who is assigned to each task. We will additionally be communicating regularly through Teams and collaborating on documents on Google Drive to make sure that we are able to work through both our individual and collaborative tasks to progress throughout the project.

Project Progression

In order to track our project progression, we have meetings twice a week to go over what we have accomplished, and what we are hoping to accomplish by the time of our next meeting. This way we have goals set in place and we can easily track what we have worked on and how long we have worked on certain tasks. Our Trello boards are the main places we are keeping track of what we're working on and the progress we're making. This is how we track our progression:

1. Completion
    a. We take into account how much we have completed since our last meeting to see that we are making progress towards our next goals.
    b. Trello Boards:
        i. Initial Planning: https://trello.com/b/mgXrcRnT/initial-planning
        ii. Design Planning: https://trello.com/b/MVHAGucf/design-planning
        iii. Iteration 1: https://trello.com/b/5566RFy6/iteration-1

        iv. Final Deliverables: https://trello.com/b/TKg41CcF/final-deliverables
2. Questions / Answers
    a. We go through any questions each of us have for the advisor and/or client during our meetings, and then we are able to get these answers the next time we meet with our advisor / client. This allows us to avoid roadblocks.
3. Software Requirements
    a. We will track our progress by seeing which requirements for the project we have actually completed. (later on in the project when we begin implementation)

Success Metrics

To evaluate our own success of our final project we outline the following metrics:

1. Implemented features and requirements
    a. The amount of features and requirements we were able to fully implement for the client
    b. The correctness of the features and requirements
        i. To be evaluated by following our testing plan: Testing Plan
2. Stakeholder satisfaction
    a. Stakeholder reported satisfaction on the project
    b. If project is unfinished, stakeholder reports satisfaction with current progress and collects final deliverables for another project group
3. Benchmarking metrics
    a. Successful performance for our JPEG-2000 decoder based on the metrics provided by the client
        i. Any increase in throughput compared to OpenJPEG (will initially aim for 5% higher throughput)
    b. Information on impacts of various properties on decoder metric performance
        i. Image size
        ii. Memory available
        iii. CPU cores available

Appendix B:

Needs, Features, Requirements

Needs

01. Current image reading process is slow for large files.
02. No benchmarks to measure the image reading process.
03. Current image preview process is slow for large files.
04. Lacking portable library to read JPEG2000.
05. Images encoded for the image analyst role are not optimized for automated processing.
    a. Images encoded for automated processing are not optimized for image analyst roles.
06. License rules apply
07. Lacking support for other file formats
08. Difficult to tell which software is the best for JPEG2000

Features

01. Read image tiling with overlap.
02. Read random chipping through the image.
03. Preview image at lower resolutions.
04. Benchmarks to measure image processing capabilities.
05. Support numerically lossless encoding.
06. Loads image data into memory.
07. Supports grayscale, 4 color bands of info
08. Compatible with various file formats
09. Outputs pixel data of parts of the image
10. Free to use for any purpose

Requirements

01. C/C++ library to interface with to support file reading for JPEG2000 on Linux.
02. JPEG200 file reader that optimizes the speed of executing each image reading scheme.
03. Benchmarks measuring the following: speed, throughput, image size, compression, location.
04. Ability to be wrapped in Python or MATLAB code.
05. Highest efficiency loading into memory.
06. The decoder supports NPJE JPEG 2000 profiles.
07. Takes in a file pointer rather than a file name, for compatibility with multiple file formats
08. Library returns a pointer to memory for image data
09. The library supports lossless encoding, including numerically lossless encoding
10. The library can read images in multiple colorspaces, including grayscale
11. The library will be comprised of free to use software

Appendix C:

Benchmarking Plan

Overview

This document will detail the purpose of the benchmark, and how we intend to meet those needs.

Purpose

Goals

In order to support our goal of improving upon the baseline of OpenJPEG for decoding JPEG2000 images of the NPJE profile, we must demonstrate that the decoding is faster. The client would also like an accurate baseline of OpenJPEG and information about improvements for OpenJPEG.

Process

We will find a variety of existing implementations to benchmark and images to benchmark with. Then we must establish metrics and a baseline for the various decoders with the metrics we have decided upon. Afterwards we will see how the decoders vary between different types of images. Finally, we will collect more detailed information to try to aid us in locating the reasons behind the differences in performance. This will assist us in finding where we can improve upon.

Decoders to examine

We have found several implementations of the JPEG2000 standard. Below are the decoders we intend to benchmark.

Decoder	**Language**	**Comments**
OpenJPEG	C	What the client specifically asked us to benchmark.
JJ2000	Java	
Jasper	C++	

Our Decoder	C	Necessary for showing improvements
Kakadu	C++	Closed source, may not be able to obtain. Claims very high speeds.
Grok	C++	Claims improvements from Kakadu in certain cases

Metrics

The clients machines have 64GB of ram, and they deal with images in the 10s of GBs. As such, we find speed to be more important than memory efficiency. We are also interested in how the size of the image influences the speed of decoding.
Our metrics are listed:

1. Speed (seconds)
2. Throughput (bits/second)

Parameters

We will vary the size of the images to see how it influences the decoding speed.
We will vary the number of threads (if applicable) allocated to the program and see how that influences speed.
Our parameters are listed:

1. Image Size (GigaBytes)
2. Number of threads

The client primarily cares about tiling. Because NPJE, our current focus, has only one tile, our only method at the moment is decoding the whole image.

Methods

Tools

We are currently using perf, a performance analyzing tool for Linux. This tool supports obtaining exact statistics from the CPUs Performance Monitoring Unit (PMU) and collecting statistics about the processing associated with certain functions. This tool also has support for multiple threads.

Methods

To measure the metrics we will use the exact statistics obtained from the PMU.
To determine possible improvements we plan to collect statistics about the processing associated with certain functions.

Environment and Data

Testing Environment

We are using a virtual machine on a type 1 hypervisor with Rocky Linux and no GUI.
The machine has 64 GB of memory, which matches our clients.
4 cores of the Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10 GHz (not hyperthreaded)
A 16MB Cache
This environment has dedicated resources and no one else will be using them, this should maximize reproducibility of the results

Testing Data

We will procure images with the NPJE profile of various sizes. We plan to have at least one image of 10GB for the sake of testing scaling.

Appendix D:

Testing Plan for JPEG 2000 Decoder

1. Introduction

- The primary goal of the project is to decode specific formats of the JPEG2000 file format
- The testing plan's goal is to provide an outline of procedures to follow when testing
- The testing plan will cover the testing of the following decoder parts:
    - Header parsing
    - Layer and progression parsing
    - Entropy decoding
    - Wavelet transform
    - Image detiling
    - Component transform
- The testing scope covers the entire system and covers unit testing, integration testing, system testing, and acceptance testing

2. Objectives

- The primary goal is to ensure the system's correctness
- The secondary goal is to assess the system's compatibility
- The tertiary goal is to assess the system's completeness
- The system's core decoding functions will be the primary focus of testing
- A successful project will be able to at least parse images in the NPJE format

3. Test Strategy

- The tests will use white box testing to assess unit methods' functionality
- The tests will use black box testing to assess the system's overall functionality
    - These black box tests will require the system to parse a known image, for example.
- The tests will use a wsl environment to assess the system's correctness initially.
- The tests will ensure that the system functions on other linux environments, namely Red Hat OS

4. Test Types

- Unit Testing - Focuses on the correctness of individual parts in the decoder pipeline
- Integration Testing - Focuses on the successful integration of the decoder parts
- System Testing - Focuses on the correctness of the entire decoder
- Acceptance Testing - Focuses on the compatibility of the decoder with the client's resources (e.g. operating systems, virtual environments, etc)

5. Test Plan

- We will create detailed unit tests when feasible
- If a detailed unit test suite is not feasible for one or more decoder parts, we will compare the output to a ground truth
- This ground truth will be based off a generated image with the specified image format
- Once all parts have passed the specified tests, parts will be integrated and tested together until the entire system is completed.
- Subsequent more complex generated images will be used to confirm the integrated system

6. Test Execution

- Tests will be executed by generating and running test files
- These test files are separate from the execution, and the execution of the program should not depend on the test files
- A test file will be created for each decoder part, and a separate test file should be used for various integration and system tests.

7. Risks and Contingencies

- Risk: The decoder cannot parse the formats needed by the client
    - Communicate with client to ensure that the decoder parses the minimum required formats (NPJE)
    - Ensure that the decoder has been thoroughly tested on images of these formats
    - Clearly establish the decoder's capabilities with the client
- Risk: The decoder cannot run on the client's environment
    - Communicate with the client to ensure the decoder runs on the client's operating environment
    - Test the decoder to ensure functionality on these environments

8. Resources

- We will be using Unity as our software testing framework
- We will be using GeoExpress to generate expected images
- We will be using VirtualBox to test compliance with the client's environment

9. Communication

- Communication between team members will primarily happen on Microsoft Teams
- Communication with the client will primarily happen through virtual meetings on Zoom and email

Appendix E:

Next Group Feasibility and Plan

Based on the original project specification handed out at the end of the 2022-2023 Rose-Hulman class year.

Project Introduction (Currently replica from previous year):

At the Johns Hopkins University - Applied Physics Lab (JHU/APL), we routinely work with science imagery from the National Aeronautics and Space Administration (NASA). As our scientists develop algorithms and pipelines to process the data, we need efficient tools to read the source imagery.

Some of the NASA missions, such as the Solar Dynamics Observatory (SDO) and Lunar Topography, produce data formatted to the JPEG-2000 image compression standard. Since JPEG-2000 has never been a widely adopted standard and is generally used only by science communities, the tooling for, and optimization of, open source image decoders is limited. To support working with this file format we seek an optimized image reading capability.

As improvements to science instrumentation and technologies allow for higher resolution astronomical imaging, images will inevitably increase in size in the future. It is important to understand the impact of this on the current file format scheme. To address this, we are seeking a benchmark suite to characterize and stress test the image-reading capability developed.

Project Description:

The primary goal of this project would be to continue the work started by the 2023-2024 JPEG-2000 Senior Project Group and finish the implementation of a C/C++ library to read JPEG-2000 images. A partial implementation of the decoding pipeline has been created and tested in C, but still requires additional work on decompressing the binary data.

The decoder is focusing on a subset of the ISO 15444-1 standard and needs to decode the NPJE specification of the JPEG-2000 format with possible support for the EPJE specification. Three image reading schemes need to be supported. The first is image tiling with overlap, the second is random chipping throughout the image, and the third is a low resolution preview of the entire image. The goal is to optimize the speed of the first two schemes and support the third for human verification. The goal of the project is to find clever ways to optimize the tile and chip read speed, such as by minimizing duplicate reads or finding optimal read orders.

The final decoder will need to be benchmarked against other available implementations of the standard. The effectiveness of the decoder will be based on metrics such as speed and throughput depending on the image size (including 10s of GBs), compression profile, location and type of image read, and the contents of the image. The format is designed to be parallelizable and

effective implementations will look for ways to maximize independence among compute cores to maximize performance. Ideally, after CPU performance is optimized GPU performance will be investigated.

Key Skills (In order of importance):
C, academic research, reverse engineering code, performance benchmarking, parallel computing, CUDA, C++

End of Year Road Map

- Compile dataset of images
    - Possibly generate some more detailed plots and breakdown of OpenJPEG using dataset
- Refactor codebase
    - Readability/Understandability
    - Add additional tests if needed for simple images
    -
- Explanatory documentation of codebase and the subset of the ISO we focus on
    - Reference codebase directly
    - Reference additional documentation and explanatory material found
    - Possible improvements

Next Year Road Map:

1. Entropy Decoding
2. De-Tiling Verification
3. Component Transformation Verification
4. Verify whole decoder on larger and differently encoded images
    a. NPJE focus, different encodings
    b. EPJE add on
5. Add PERF hooks for benchmarking implementation
6. Gather initial benchmark comparisons between implementation and OpenJPEG
    a. Consider benchmarks for JJ2000 and other implementations
7. Design Improvements
    a. Multi-processing
    b. Memory Improvements
    c. Entropy Decoding improvements
        i. Possible examples in JJ2000
8. GPU support