An FPGA Implementation of a Custom JPEG Image Decoder SoC Module

George Gabriel Kyrtakas
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/5945
An FPGA Implementation of a Custom JPEG Image Decoder SoC Module

by

George Kyrtsakas

A Thesis
Submitted to the Faculty of Graduate Studies through the Department of Electrical and Computer Engineering in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science at the University of Windsor

Windsor, Ontario, Canada
2017
An FPGA Implementation of a Custom JPEG Image Decoder SoC Module

by

George Kyrtakas

APPROVED BY:

B. Boufama
Computer Science

M. Khalid
Electrical and Computer Engineering

R. Muscedere, Advisor
Electrical and Computer Engineering

February 14, 2017
Co-Authorship Declaration

I hereby declare that this thesis incorporates material that is result of joint research, as follows: the Verilog code presented in Appendix A is the outcome of a joint effort between myself, George Kyrtsakas, and my supervisor, Dr. Roberto Muscedere.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

I declare that, to the best of my knowledge, my thesis does not infringe upon anyones copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright
CO-AUTHORSHIP DECLARATION

owner(s) to include such material(s) in my thesis.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.
Abstract

An important feature of today’s mobile devices is their ability to capture and display high resolution photos in an acceptable time frame. The vast majority of images are stored on disk using the JPEG codec for compression. With increasing pixel counts on both image sensors and screens, software solutions will struggle in their ability to decode JPEG image data, since they rely solely on increasing CPU power. The need is becoming clearer for hardware acceleration to replace the CPU when decoding large images.

This thesis presents a System-on-Chip module that is able to relieve the CPU of the computationally intense task of decoding a JPEG image. This SoC module was developed and tested on an FPGA that features an ARM Cortex A9 and a Xilinx Artix-7 FPGA. The SoC module was able to outperform software running on the onboard CPU by about 4 times, while being more accurate to the original image.
Dedication

To my family, this work is the culmination of twenty-four and a half years of continual love and support from you. This is as much your achievement as it is mine. Thank you.
Acknowledgments

I would like to thank my Supervisor, Dr. Muscedere, for bringing this project to my attention, and for his work, upon which this project was built.

I would like to thank my committee members, Dr. Khalid and Dr. Boufama, for their advice and for sitting on my committee.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co-Authorship Declaration</td>
<td>iv</td>
</tr>
<tr>
<td>Abstract</td>
<td>vi</td>
</tr>
<tr>
<td>Dedication</td>
<td>vii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The JPEG Codec</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation of Research</td>
<td>2</td>
</tr>
</tbody>
</table>

Co-Authorship Declaration

Abstract

Dedication

Acknowledgments

List of Figures

1 Introduction

1.1 The JPEG Codec

1.2 Motivation of Research
## CONTENTS

1.3 Thesis Outline ........................................... 3

2 The JPEG Standard ........................................ 4
   2.1 The Encoding Process ................................ 4
      2.1.1 Colour Space Conversion .......................... 5
      2.1.2 Component Subsampling .......................... 8
      2.1.3 Block Splitting .................................. 9
      2.1.4 2-dimensional Discrete Cosine Transform ....... 9
      2.1.5 Quantization and Zig Zag Order ................. 10
      2.1.6 Entropy Coding .................................. 11
   2.2 The Decoding Process ................................ 12
      2.2.1 Entropy Decoding ................................. 12
      2.2.2 Huffman Decode .................................. 13
      2.2.3 YCbCr to RGB .................................... 15
   2.3 File Structure and Restart Markers ................... 16
   2.4 Survey of JPEG Images on the Internet ................ 17
   2.5 Summary .............................................. 18

3 Previous Research ......................................... 19
   3.1 Software JPEG Decompression ......................... 19
      3.1.1 libjpeg .......................................... 19
      3.1.2 libjpeg-turbo ................................... 20
      3.1.3 NanoJPEG ........................................ 20
      3.1.4 jpeg2000 ......................................... 21
   3.2 Hardware JPEG Decompression .......................... 21
   3.3 Discrete Cosine Transform ............................ 22
   3.4 Fast Huffman Decoding ................................. 23
3.5 JPEG Codec in Hardware ........................................ 23
  3.5.1 High Performance JPEG Decoder Based on FPGA ....... 23
  3.5.2 Hardware Support of JPEG ............................... 24
  3.5.3 FPGA Based Baseline JPEG Decoder ...................... 24
  3.5.4 Hardware JPEG Decoder and Efficient Post-Processing ... 24
  3.5.5 CUDA-Based Acceleration of the JPEG Decoder .......... 24
  3.5.6 A JPEG Huffman Decoder using CAM ..................... 25
3.6 Summary ......................................................... 25

4 Proposed Solution ................................................ 26
  4.1 Development Board ........................................... 26
  4.2 Communication Protocols .................................... 27
    4.2.1 AXI3 ...................................................... 28
    4.2.2 Control Interface ........................................ 29
    4.2.3 Data Transfer ............................................ 30
  4.3 Hardware Design ............................................... 30
    4.3.1 Top Level Module - user_logic.v ....................... 31
    4.3.2 decode.v ................................................ 32
    4.3.3 blocker.v and header.v ................................. 32
    4.3.4 stream.v and huff.v .................................... 32
    4.3.5 idctcol.v and idctrow.v ............................... 35
    4.3.6 colourmap.v ............................................. 35
  4.4 Software Interface ........................................... 36
    4.4.1 Memory Organization .................................... 36
    4.4.2 Software Responsibilities .............................. 37
  4.5 Summary ......................................................... 38
5 Results

5.1 Test Structure .................................................. 39
  5.1.1 ZedBoard Configuration ................................. 40
  5.1.2 Test Image Database ................................. 40
  5.1.3 Testing Process ................................. 41

5.2 Testing for Accuracy ........................................... 42
  5.2.1 Mean Squared Error ................................. 42
  5.2.2 Peak Signal to Noise Ratio ................................. 42
  5.2.3 Accuracy Results ................................. 43

5.3 Testing for Speed ........................................... 44
  5.3.1 Speed Results ................................. 45

5.4 Hardware Reports ........................................... 47

5.5 Summary .................................................. 49

6 Summary .................................................. 50

6.1 Conclusions .................................................. 50

6.2 Recommendations for Future Work ................................. 51

References .................................................. 52

A Verilog Code .................................................. 54

A.1 user_logic.v .................................................. 54

A.2 decode.v .................................................. 71

A.3 blocker.v .................................................. 78

A.4 header.v .................................................. 82

A.5 stream.v .................................................. 100

A.6 huff.v .................................................. 105

A.7 dpram.v, dparam.v, dpsram.v, asyncmem.v ................................. 125
## CONTENTS

A.8 idctrow.v and idctcol.v ........................................... 127
A.9 colourmap.v and zigzagcont.v ..................................... 132

### B C Code

B.1 hwjpeg.c ................................................................. 137
B.2 hwmap.c and hwmap.h .................................................. 144
B.3 psnr.c ..................................................................... 146
B.4 ljpeg.c and ljpegt.c ..................................................... 149

### C Bash Scripts

C.1 iwhbyd.sh ................................................................. 159
C.2 pccompanion.sh .......................................................... 162
C.3 md5gen.sh ................................................................. 163

### Vita Auctoris

165
# List of Figures

2.1 An Image of Detroit ................................................. 6
2.2 Luminance Component of an Image of Detroit ....................... 7
2.3 Chrominance Components of an Image of Detroit (Cb-Left,Cr-Right) 7
2.4 4:1:0 Subsampling Configuration .................................... 8
2.5 An 8x8 block and its 2D-DCT ....................................... 10
2.6 Example of a Luminance Quantization Table ....................... 10
2.7 Zig Zag Order ....................................................... 11
2.8 JPEG Encoding Process ............................................. 12
2.9 Extended Huffman Decode .......................................... 14
2.10 Usenet JPEG Image Survey Results ............................... 17

3.1 Loeffler’s 8-point Forward DCT ................................. 22
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Block Definitions for Figure 3.1</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>AXI3 Read Burst</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Hardware Design Block Diagram</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>JPEG DC Huffman Tree Example</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Huffman Table corresponding to Figure 4.3</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Loeffler’s 8-point IDCT</td>
<td>35</td>
</tr>
<tr>
<td>4.6</td>
<td>Memory Organization</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Accuracy Results</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Mobile vs. Desktop CPU</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>Decode Time Results</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Resource Usage on the FPGA</td>
<td>48</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Digital imaging is an extremely popular method of capturing important moments of peoples lives. With the rapid advancements in image sensing technology, especially on the smart phone platform, one of the problems that arises is the ability to decode stored image data, which is very commonly stored using the JPEG codec.

1.1 The JPEG Codec

In 1992 the Joint Photographic Experts Group (JPEG) released the JPEG codec standard under ITU-T Recommendation T.81 and in 1994 as ISO/IEC 10918-1 [1]. Their goal was to facilitate the movement of images between computers in a low bandwidth setting by compressing the image using a multitude of techniques that would not greatly affect image fidelity. The JPEG standard outlines the process for encoding and decoding a JPEG image as well as four different modes of operation: sequential DCT-based or Baseline Sequential, Progressive, Lossless, and Hierarchical.
Baseline Sequential, hereinafter referred to as Baseline, encodes an image from left to right and top to bottom in blocks. Baseline is by far the most common mode of operation of the JPEG standard. Progressive JPEGs use multiple passes of Baseline at increasing levels of quality, which would be ideal in a web setting where the image can be displayed first at lowest quality, but the entire set of passes must be stored in memory to complete the decode which vastly increases the amount of resources required to decode a single image. Lossless and Hierarchical are so rarely used that they will not be in the scope of this thesis.

Baseline JPEG is an inherently lossy image codec due to the processes it uses to encode and decode image data. These processes were selected after careful consideration from the JPEG group for their ability to represent multiple data well in a compressed image, however they were not chosen for their ability to be implemented in parallel. Decoding a JPEG is a serial process that was implemented in a time when personal computers had only one processor and so the codec is largely designed to be run on a single thread of execution.

1.2 Motivation of Research

In the past, people would store their photographs physically, in albums, and to view these images they were only required to open the album. The increasing use of smartphones has caused the decline of the physical photograph and an increase in the number of people who carry their entire photo collection on their mobile device. Since the invention of the smartphone, pixel counts on-screen and in onboard image sensors have skyrocketed, with trends pointing to 8K displays with upwards of 30 megapixel (MP) cameras. Devices today rely on software libraries and increasing CPU power to decode these images in a timely manner. They also use thumbnails and pre-rendered files at different resolutions, but these methods will only prove to be costly in a future
when extremely high resolution images become the norm.

The focus of this thesis is to present a System-on-Chip (SoC) module that will alleviate the pressure on the CPU when the user wants to view their images. This decoder module will attempt to remove the CPU from the process almost entirely and act as a coprocessor dedicated to decoding JPEG baseline images.

1.3 Thesis Outline

Chapter 1 serves to introduce the project and motivations behind it, discussing the current state of the market and its reliance on the JPEG standard. Chapter 2 goes in depth on the JPEG standard, showing the encoding and decoding process and discussing the limitations imposed on hardware by the architecture of the standard. Chapter 3 introduces previous works that aimed to improve the performance of the standard or the standard itself, either by hardware or software.

The proposed solution is presented in Chapter 4 and includes discussions on the implementation and its platform, the design of the hardware, and the design of the accompanying software. Chapter 5 presents the results produced when testing the proposed solution for accuracy and speed, while also explaining how those tests were performed. Chapter 6 draws conclusions from the work and gives recommendations on future work.
Chapter 2

The JPEG Standard

The JPEG standard, while almost a quarter century old, remains the industry standard for encoded image data. This chapter serves to introduce the encoding and decoding methods defined in the standard, and details each part of the decode process.

2.1 The Encoding Process

The encoding process starts with a colour space conversion, subsampling by colour component, and splitting each component into blocks depending on the subsampling factor. The data is then put through a 2-dimensional discrete cosine transform (DCT), it is quantized and then entropy coded. The output of this entropy coding is what makes up the data streams that will be stored in the output file. All other necessary information is stored in the header of the file, such as the image dimensions and the tables used for quantization and Huffman coding.
2.1.1 Colour Space Conversion

The input image data, usually in Red-Green-Blue (RGB) format, is converted to the YCbCr colour space, which uses one channel for luminance (Y), or brightness, and two channels for chrominance (Cb, Cr), or colour differencing. Separating the brightness plays an important role in the compression of the image data, as the human eye is more sensitive to changes in brightness over a small area than it is to changes in colour over a small area. This effect is shown in Figure 2.2, which shows the Y channel of a selected image, and Figure 2.3 which shows the Cb and Cr channels of that same image. Looking at these figures, there is more visual information stored in the luminance channel than in the two chrominance channels. So typically, the two chrominance channels are subject to greater compression throughout the different encoding stages than the luminance channel. The first example of that is in the optional channel subsampling process.
Figure 2.1: An Image of Detroit
2. THE JPEG STANDARD

Figure 2.2: Luminance Component of an Image of Detroit

Figure 2.3: Chrominance Components of an Image of Detroit (Cb-Left,Cr-Right)
2.1.2 Component Subsampling

The JPEG standard outlines an optional process to further reduce the amount of data required to store an image while having a minimal effect on visual fidelity. Subsampling is the process of reducing the resolution of a colour component, which is usually only applied to the chrominance components. A Minimally Coded Unit (MCU) is a macro block comprised of the blocks of each colour component that represent a given region of the image. In Figure 2.4, assuming the MCU represents the upper left corner of an image, or the starting block, and assuming each colour component subblock is of size 8 x 8, there is much more detail in the Y component as each value in the MCU has a corresponding Y value. The chrominance components, however, have been subsampled to one-half the original vertical resolution, and one-quarter the original horizontal resolution, so their values correspond to more than one value in the MCU.

Subsampling is expressed as a ratio, A:B:C, where A is the horizontal reference, B is the horizontal chrominance count, and C is the vertical chrominance count in Figure 2.3, the subsampling is expressed as 4:1:0. The JPEG standard allows for subsampling as long as the total number of blocks that make up an MCU does not exceed 10, meaning there are 195 valid combinations of subsampling in the YCC colourspace.
2. THE JPEG STANDARD

2.1.3 Block Splitting

Following the optional subsampling, the data is split into blocks, the size of which is determined by the directional subsampling factor. The default block size is 8 x 8 and increases by a multiple of 8 for both subsampling factors which would make the block size 32 x 16 in Figure 2.4.

2.1.4 2-dimensional Discrete Cosine Transform

After block splitting, each block undergoes a 2-dimensional discrete cosine transform (DCT) that converts the data from the spatial domain to the frequency domain. A 2D DCT is equivalent to performing a 1D DCT, shown in Equation 2.1, on each row followed by each column.

\[
X_k = \sum_{n=0}^{N-1} x_n \cos \left( \frac{k \pi}{N} (n + 0.5) \right), \quad k = 0 \to (N-1) \quad (2.1)
\]

The 2D DCT is an important part of the JPEG encode process because it tends to focus the information towards the low frequency coefficients and away from the high frequency coefficients, which represent data that the human eye would have a very hard time discerning. The lower frequency coefficients reside in the upper left corner of the block, as shown in Figure 2.5, and the high frequency coefficients reside in the lower right hand corner. The upper-left-most value after the DCT is called the DC component, and the remaining values are called AC components, this is important as they will be encoded differently in the final stage of the encoding process.
2. THE JPEG STANDARD

2.1.5 Quantization and Zig Zag Order

Quantization is another lossy part of JPEG encoding, where the elements of a block are divided by a corresponding value in the quantization tables. These tables place an emphasis on preserving the low frequency values of a block, while very commonly reducing the high frequency values to zeroes. Typically, there are two quantization tables split between the luminance and the chrominance channels.

\[
\begin{bmatrix}
72 & 69 & 65 & 63 & 62 & 60 & 55 & 51 \\
68 & 66 & 63 & 60 & 58 & 55 & 52 & 49 \\
65 & 63 & 61 & 58 & 55 & 51 & 49 & 48 \\
62 & 59 & 57 & 54 & 51 & 48 & 46 & 45 \\
60 & 56 & 52 & 51 & 49 & 47 & 45 & 44 \\
57 & 52 & 49 & 48 & 45 & 44 & 44 & 44 \\
54 & 50 & 47 & 46 & 43 & 43 & 46 & 46 \\
51 & 48 & 47 & 46 & 44 & 41 & 43 & 47 \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
-606 & 38 & 4 & 3 & 4 & 0 & 0 & 0 \\
42 & -6 & 3 & -4 & 0 & 0 & 0 & 0 \\
6 & -4 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 5 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

Figure 2.5: An 8x8 block and its 2D-DCT

\[
\begin{bmatrix}
3 & 2 & 2 & 3 & 4 & 6 & 8 & 10 \\
2 & 2 & 2 & 3 & 4 & 9 & 10 & 9 \\
2 & 2 & 3 & 4 & 6 & 9 & 11 & 9 \\
2 & 3 & 4 & 5 & 8 & 14 & 13 & 10 \\
3 & 4 & 6 & 9 & 11 & 17 & 16 & 12 \\
4 & 6 & 9 & 10 & 13 & 17 & 18 & 15 \\
8 & 10 & 12 & 14 & 16 & 19 & 19 & 16 \\
12 & 15 & 15 & 16 & 18 & 16 & 16 & 16 \\
\end{bmatrix}
\]

Figure 2.6: Example of a Luminance Quantization Table

After quantization, the high concentration of non-zero coefficients in the upper left corner of the block is taken advantage of again when the block is ordered by zigzag. Shown in Figure 2.7 is zigzag order which allows the next stage, entropy coding, to...
perform operations that compress the block data further than if the data was taken in order.

![Zig Zag Order](image)

Figure 2.7: Zig Zag Order

### 2.1.6 Entropy Coding

The final stage of encoding a JPEG is entropy coding which is comprised of Huffman coding and Run Length Encoding (RLE) for the AC coefficients, and Huffman coding and Differential coding for the DC coefficients. These topics are covered in Section 2.2 as they are extremely important to understanding the decoding process. The combined outputs of these encoders are what make up the data streams that are stored in the body of the JPEG file. The entire encoding process is shown by block diagram in Figure 2.8.
2. THE JPEG STANDARD

2.2 The Decoding Process

The JPEG decoding process is the reverse of the encoding process in that the data is entropy decoded, then dequantized and put through a 2D Inverse DCT (IDCT), the blocks of data are reassembled and undergo colour conversion from YCbCr to RGB. The decoding process starts with gathering the necessary information from the header of the JPEG file, such as quantization and Huffman tables, as well as image dimensions. After the Huffman tables and quantization tables have been constructed the decode can begin.

2.2.1 Entropy Decoding

Entropy decoding a JPEG image consists of two decoding processes: one for the DC components using Huffman decoding and Differential decoding, and one for the AC components using Huffman decoding and Run Length Decoding (RLD). These processes are combined during the encoding phase, creating a hybrid encoded structure which makes the JPEG especially taxing to decode.

After a Start of Scan (SOS) marker is found, the data is read bit-by-bit until a
match in the DC Huffman table is found. The decoded value from this match indicates how many bits should be read next so the DC value of the first block can be obtained. The bits are read and decoded using another table that decodes the DC value, and this decoded value is the difference between the DC value of the previous block and the DC value of the current block. If there is no previous block, the previous value is assumed to be 0. With the DC coefficient being decoded, the next part of the process is to decode the associated AC coefficients.

The bitstream is again read bit-by-bit until a match is found in the AC Huffman table. The decoded value is 1 byte in length and has two pieces of information: the most significant 4 bits designate how many Run-Length Encoded (RLE) zeros are to follow the next decoded AC coefficient, and the least significant 4 bits designate how many bits are to be read for the AC coefficient. The AC coefficient is the value of that next number of bits and the run length zeros, a maximum of 16 of them, are the next AC coefficients. The AC decode process is repeated until all 63 AC coefficients have been decoded, then the next block can be decoded, again starting with the DC coefficient.

Entropy decoding in the JPEG standard is a very expensive operation because information is not byte-aligned in this scheme. Many blocks are contained in one scan and there is no information on where the next block will start, so the blocks must be decoded in order, one by one. This severely limits the options available to be able to speed up the decode process, in hardware or software.

2.2.2 Huffman Decode

This section outlines the Huffman decode process. Particular attention should be paid to the amount of bitwise operations that are performed on the datastream, where a CPU is not designed to handle bit-level data in large quantities.
1. Read datastream bit by bit, see Step 1 in Figure 2.9, until the codeword matches a codeword found in the DC Huffman table.

   **Codeword:** 11100
   **Value:** 08

2. Value: 08 is the number of bits that represent the DC Huffman codeword, so read the next 8 bits

3. Codeword: 01101000, see Step 3 in Figure 2.9, is decoded using the following process, for DC values only
   (a) Perform bitwise NOT: 01101000 becomes 10010111
   (b) Decode value as unsigned integer: 10010111b = 151d
   (c) Invert sign of integer: 151d becomes -151d

So the DC difference value of the first Luminance block is -151. To obtain the DC coefficient of that block, since the element is difference encoded, we add -151 to the DC component of the previous block. In this case there is no previous block so the value is taken to be zero.
DC = 0 + (-151)
DC = -151

4. Read datastream bit by bit, see Step 4 in Figure 2.9, until the codeword matches a codeword found in the AC Huffman table.

   Codeword: 11010
   Value: 05

Value: 05 is a hexadecimal byte that is formatted $RRRRSSSS$ where:

   $RRRR$ is the number of Run Length Encoded zeros
   $SSSS$ is the number of bits to read for the next value

So for this example there are 0 RLE zeros to follow the next value, and 5 bits will be read to obtain the 1st AC coefficient of the block.

5. Read $SSSS$, or 5, bits to obtain the AC coefficient, see Step 5 in Figure 2.9,

   Value: 10011b = 19d

So the first AC coefficient is 19

6. Repeat steps 4 and 5 until End of Block marker is found, then repeat whole process for the next block.

2.2.3 YCbCr to RGB

After the 2-dimensional Inverse Discrete Cosine Transform (IDCT), which is discussed in detail in Section 3.3 and Section 4.3.5, the data has to be transformed from YCbCr to the RGB colourspace.
2. THE JPEG STANDARD

\[ R = Y + [1.402 \times (Cr - 128)] \] (2.2)

\[ G = Y - [0.344136 \times (Cb - 128)] - [0.714136 \times (Cr - 128)] \] (2.3)

\[ B = Y + [1.772 \times (Cb - 128)] \] (2.4)

When Equations 2.2, 2.3, and 2.4 are implemented using fixed-point hardware, the overall computational overhead for the conversion of one pixel becomes 4 additions and 4 multiplications. This might not seem expensive, but when multiplied by the number of pixels in an image, becomes a very taxing operation on the system.

2.3 File Structure and Restart Markers

A JPEG image file has a header-body structure, where the header contains peripheral information, and the body contains the encoded image data. JPEGs use markers to designate data that is relevant to certain parts of the image. The markers are 1 byte in length and directly follow the hex byte FF to define their existence. A valid JPEG image file will start with the Start of Image (SOI) marker, or 0xFFD8. Define Quantization Table (DQT) or 0xFFDB, Define Huffman Table (DHT) or 0xFFC4, and Start of Scan (SOS) or 0xFFDA are all important markers, the last of which defines the start of the body of the image. The End of Image (EOI) marker, or 0xFFD9, ends the file.

Restart markers, or 0xFFDy, are used to resynchronize an image if an error occurs. If a restart marker is encountered, all DC differential values are reset to 0, and the bitstream is restarted on a byte boundary following the marker. Values of y in the marker are used to track whether or not large chunks of data are missing. If the last
restart marker encountered is \(0xFFD2\) and the next restart marker encountered is \(0xFFD4\), the decoder knows it is missing a chunk of data from the 3rd restart marker and if enough data is present in the 4th marker, the decoder can replace the data missing using the data in the 4th marker.

Restart markers, however uncommon they may be, present an opportunity for multiprocessing an image decode. Using the restart markers as starting spots, an image can be decoded by separate processes and stitched together as the processes finish.

### 2.4 Survey of JPEG Images on the Internet

In order to get an accurate representation of how the JPEG codec is used, a survey of available JPEG images was performed by Dr. Muscedere, provided in private communication for this thesis. The JPEG images were found on Usenet in 2016, where only the headers were downloaded. The survey was comprehensive, obtaining the headers of approximately 7.35 million JPEGs, where it should be noted that only 2 images were of JPEG2000 format. The results in Figure 2.10 show that the overwhelming majority of JPEG images are still stored in the Baseline JPEG format.

<table>
<thead>
<tr>
<th>Format</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>91%</td>
</tr>
<tr>
<td>Progressive</td>
<td>8%</td>
</tr>
<tr>
<td>Extended (10/12 bit)</td>
<td>&lt;1%</td>
</tr>
</tbody>
</table>

Figure 2.10: Usenet JPEG Image Survey Results

The survey also showed that about 1.2 million, or about 16% of the images used restart intervals. So only 16% of images could benefit from a software-based multi-threaded JPEG decoder that relies on restart intervals to have concurrent processes.
2.5 Summary

This chapter introduced the processes that make up the JPEG codec. It is important to note the complex and serial nature of the JPEG codec, as that will govern the development of a hardware solution. Specifically, the Huffman coding of a JPEG limits its ability to be implemented in parallel because of the indeterminate nature of the next codeword.
Chapter 3

Previous Research

This chapter introduces several works that implement and improve upon the JPEG standard. Each work has its own benefits and limitations, which are discussed in detail to serve as a primer for the rest of the thesis.

3.1 Software JPEG Decompression

3.1.1 libjpeg

The most common way to decode a JPEG is by using the freely available libjpeg, which is a C library that has been available since the first release of the JPEG standard. The source code of libjpeg is a massive collection of files that implement different functionalities based on the system for which it is being compiled to serve. This library carries a lot of bulk with it as it tries to be a catch-all solution for every type of JPEG ever produced, including those that have different block sizes that are
typically not seen in JPEGs.

As of January 2016, the current release version of libjpeg is 9b [2]. It has undergone 9 major version changes since its initial release and is maintained and released by the Independent JPEG Group (IJG).

3.1.2 libjpeg-turbo

A popular fork of the libjpeg project is libjpeg-turbo, which was built to take advantage of innovations in hardware using special instructions that call upon dedicated hardware to perform a single instruction on lots of data. Single Instruction Multiple Data (SIMD) instructions are standard on today's microprocessors and libjpeg-turbo uses these instructions to accelerate the processes in libjpeg. This library is platform dependent because of the differences in SIMD architectures between different companies, NEON for ARM processors and MMX/SSE2 for Intel processors. [3]

libjpeg-turbo is very common in mobile applications due to the popularity of ARM processors in mobile phones. Because of its increase in speed, this project will use libjpeg-turbo as a benchmark for acceleration, but not as a benchmark for accuracy, due to its use of SIMD instructions which use reduced precision data types for calculations.

3.1.3 NanoJPEG

Although libjpeg and libjpeg-turbo are very common, they are not well suited to be implemented on hardware because of their numerous source files and even more numerous configuration options. NanoJPEG is a project that attempts to implement a JPEG decoder in a compact way, without sacrificing too much quality. NanoJPEG is implemented in a single C file, which makes it ideal to be used as a guide for building a JPEG decoder in hardware [4]. It was used as a template in this work for
3. PREVIOUS RESEARCH

the hardware design, as well as verification.

3.1.4 jpeg2000

In the year 2000, the JPEG group released what was supposed to be the successor to the JPEG codec, making multiple improvements including scalable compression, error correction, and reversible wavelet transforms instead of the traditional DCT [5]. The issue with jpeg2000 is that patent licensing concerns have held it back, causing the adoption of jpeg2000 to be near 0% of all images available on the internet. Its predecessor, libjpeg, although it is 25 years old, remains the image compression standard due to its widespread adoption and the fact that there are no licensing concerns about the software.

3.2 Hardware JPEG Decompression

In 2013, a student at the University of Windsor, Dan Macdonald, published a thesis called Hardware JPEG Decompression wherein he proposed a JPEG decoder that offloaded the IDCT and colour conversion portions of the decode from libjpeg to hardware [6]. There were improvements on the time required to decode certain images, but there are limitations that affect the performance of the system.

The project is implemented on a Xilinx FPGA board that does not have a CPU, which requires that the project use valuable resources to implement a soft processor on the FPGA. Having a physical processor on board would have presented a very substantial advantage to the acceleration of the JPEG codec, but that technology was not readily available at that time.
3.3 Discrete Cosine Transform

The Discrete Cosine Transform (DCT) was introduced in 1974 by Ahmed et. al as an algorithm that could be applied to digital signal processing in the area of pattern recognition [7]. Since the DCT and its inverse are integral to the JPEG codec, there was a need to develop a faster version of the algorithm that could be built into hardware or software.

In 1977, Chen et. al produced a fast DCT algorithm which was 6 times faster than Fast Fourier Transform-based DCT implementations at the time [8]. And in 1989, during the early development stages of the JPEG codec, Loeffler et. al introduced a faster algorithm for computing the DCT and IDCT that only required 11 multipliers and 29 adders for an 8-point calculation. Loeffler’s implementation took advantage of Chen’s implementation by factoring coefficients to reduce the number of arithmetic units required, at the expense of an increased critical path. Their designs take advantage of the even and odd symmetry of the DCT to make it a 4-stage algorithm with significantly less hardware. [9]

![Figure 3.1: Loeffler’s 8-point Forward DCT](image-url)
3.4 Fast Huffman Decoding

In 1995, Choi et al. presented a fast Huffman decoder that used high speed pattern matching and tree clustering [10]. Their focus on reducing memory use helped video applications at the time, but there was no mention on other performance metrics that might be useful in the JPEG codec, such as speedup.

3.5 JPEG Codec in Hardware

3.5.1 High Performance JPEG Decoder Based on FPGA

Shan et al. presented a hardware JPEG decoder in which the Huffman decoder was three stages that relied upon code length to calculate the memory locations of outputs. The IDCT was a direct implementation of the Loeffler IDCT, which is the reverse operation of that shown in Figure 3.1. Their test methodology was extremely sparse as they only used one image for testing and they claim to be able to decode at 30 frames per second at a resolution of 1920 x 1080. Asynchronous FIFOs were used to adjust for pipeline stalls. DDR2 and block RAMs were used as line buffer in this design [11].
3. PREVIOUS RESEARCH

3.5.2 Hardware Support of JPEG

Elbadri et. al from the University of Ottawa, in 2005, presented a survey of hardware for the different blocks required by a JPEG encoder and decoder. Their work found an almost 8 times speedup on a 67 MHz FPGA versus a 400 MHz CPU [12].

3.5.3 FPGA Based Baseline JPEG Decoder

In 2000, Yusof et. al proposed a baseline JPEG decoder that was able to decode at 30 frames per second for an image size of 320 x 240 [13]. Their pipeline did not include a Loeffler IDCT but instead used the formal definition of an IDCT to create hardware, which used a significant chunk of their available gates. Of particular interest is their Huffman decoder which used the code length in a feedback to determine the output of a Huffman code.

3.5.4 Hardware JPEG Decoder and Efficient Post-Processing

In 2012, Zhu and Du proposed a hardware JPEG decoder that included three post-processing functions for embedded applications [14]. They included Inner Down-Scaling, Region of Interest decoding, and Partial decoding, all of which would be useful in embedded feature detection applications. Their focus was more on the post-processing than on the inner workings of a JPEG decoder.

3.5.5 CUDA-Based Acceleration of the JPEG Decoder

In 2013, Yan et. al proposed a CUDA based JPEG decoder that was able to double the speed at which JPEGs were decoded. They also noted that their implementation was able to perform IDCT calculations 49 times faster than the CPU implementation, but they did not explain their testing methodology nor did they account for memory
transfer times that are significantly costly in a GPU setting [15].

### 3.5.6 A JPEG Huffman Decoder using CAM

In 1993, Komoto et. al proposed a high-speed and compact-size Huffman decoder using Content Addressable Memory, or CAM [16]. Their design consisted of two CAMs for the AC Huffman codes, each at 162 elements deep, as well as two CAMs for the DC Huffman codes, each at 11 elements deep. The survey mentioned in Section 2.4 showed that 45.4% of JPEG images had more Huffman codes than the proposed design had available memory locations.

The CAM is a fully custom design that is not easily scaled to different implementations. Given that 47.5% of the images in the survey use standard Huffman tables, the CAM approach to Huffman decoding presented will not work with the majority of today’s JPEG images.

### 3.6 Summary

This chapter presented previous works to be taken into consideration when building a hardware solution for decoding JPEGs. While some solutions show promise in their claims, they lack in their real world test results, which is where this thesis will aim to improve upon the previous works.

There has not been much published work in developing a hardware solution for JPEG decoding in the 25 year history of the codec, so there is either a heavy reliance on increasing processor power by industry, or organizations are simply not publishing their work.
Chapter 4

Proposed Solution

This chapter serves to outline and detail the proposed solution for accelerating the JPEG decode process. It covers the design of the SoC module as well as the board it is implemented on, and describes the software required to control it.

4.1 Development Board

The Digilent ZedBoard was used to implement the SoC module and test its functionality. The ZedBoard is a low-cost development board that features the Xilinx Zynq-7000 SoC, as well as several other features that make it ideal for prototyping a hardware design. The features that are of particular interest to this project include:

- ARM A9 Dual-Core CPU
- Xilinx Artix-7 FPGA
- 512 MB DDR3 RAM
4. PROPOSED SOLUTION

- Gigabit Ethernet
- HDMI Output

The ZedBoard having an FPGA and a CPU is a great advantage over other development boards where the CPU must be implemented on the FPGA as a soft processor, taking up valuable resources that could be allocated towards the hardware design. With the CPU and FPGA being on the same chip, communication between them is simplified and latencies are reduced, allowing for faster designs.

To facilitate the implementation of the design and its testing, a modified Linux kernel designed for embedded systems, Linaro, will run on the ARM CPU. Linux will be used to execute code in conjunction with the hardware, to control it and test it, allowing for a more streamlined development environment.

This board was chosen for development because of its features, but also because it was on the lower end of what is available to a consumer. It was certainly possible to develop this solution on a more expensive board with better specifications, but that would only prove this design is possible in a price range that is prohibitive to the consumer. A goal of this project was to implement the design on a relatively inexpensive board, thereby increasing the accessibility of the market, without handcuffing the development process by selecting a board without enough features.

4.2 Communication Protocols

The Zynq-7000 All-Programmable SoC contains a CPU and an FPGA that need to communicate to complete tasks in unison. The ARM CPU dictates the use of an open source communication protocol called AXI which is a specification of the open source ARM Advanced Microcontroller Bus Architecture (AMBA). AMBA Advanced eXtensible Interface (AXI) is used extensively throughout this project, using versions
4. PROPOSED SOLUTION

3 and 4, where IP Cores defined by the specification and custom AXI solutions are part of the design.

4.2.1 AXI3

The target device uses AXI3 as its communication protocol. The Xilinx software provides the option to use AXI4, but upon further investigation, if a design uses AXI4, the software inserts AXI bridges to convert the AXI4 transfer to AXI3, which adds unnecessary bulk to the FPGA design.

AXI3 is a burst-based handshake protocol that uses two-way VALID and READY signals. It allows up to 16 transfers per burst, and transfer sizes up to 1024 bits. In Figure 4.1, an example of an AXI3 read burst is shown. The master asserts ARVALID and places the address on the ARADDR bus, the slave asserts ARREADY and takes reads the address from the bus. After deasserting ARVALID, the master asserts RREADY to signal it is ready to take in data. The slave places the data on the bus and asserts RVALID. The master and slave know a transaction is complete when both RREADY and RVALID are asserted simultaneously on a rising clock edge. On the last transfer the slave also asserts RLAST and the transfer is complete.
4. PROPOSED SOLUTION

### 4.2.2 Control Interface

Controlling the SoC module is paramount to its operation, whether the commands are simple or a kernel driver is implemented, the fundamentals of communication are the same. AXI4-lite defines a set of software accessible hardware registers that have a relatively low bandwidth compared to its siblings, AXI4 and AXI4-stream. These registers are used in the SoC module to convey control information and receive status information.

The control registers are assigned a memory address when the kernel is booted, this address is hardcoded in the Linux systems device tree, which is a file that is used to designate hardware peripherals in some embedded systems. In C, a pointer is

![AXI3 Read Burst Diagram](image-url)
assigned to the address of the hardware registers, so they can be read from or written to by referencing the pointer.

### 4.2.3 Data Transfer

After the control registers have been set and the command to begin the decode is given, the image is pulled from a predetermined location in DDR3 RAM in chunks of a preset size. The SoC Module does all of this, there is no interference from software after the control registers have been set. The transfer happens over AXI3, with a burst size of 16 words per transfer.

The Zynq processor on the ZedBoard natively uses all AXI3. AXI4-lite is AXI3 with 1-word transfers so it is easier to implement. AXI4 is simply AXI3 but with transfers extended from 16 word to 256 words. The IP block is imported into Xilinx Platform Studio and it automatically creates a bridge from AXI3 to AXI4-lite for the command channel, whereas the main memory transfer does not require a bridge.

AXI is a master-slave bus protocol and the proposed solution takes advantage of this by being a master for memory transfers, and a slave for control registers. This allows the SoC module to be independent of the CPU when reading or writing data to and from RAM. Not having to wait for the CPU to push and pull new data is an enormous advantage over other solutions currently available.

### 4.3 Hardware Design

The hardware design is comprised of many submodules that separate the functionality of the JPEG codec into a sort of pipeline. Figure 4.1 shows the overall design pipeline, this subchapter will serve to thoroughly explain each submodule in the order that the data would flow through them.
4. PROPOSED SOLUTION

4.3.1 Top Level Module - user_logic.v

The top level module, user_logic.v, is responsible for implementing the software accessible registers for control of the system, interfacing with RAM to read and write data, and creating an instance of the decoder submodule with its First In First Out
(FIFO) buffers. This module acts as the go-between for software and hardware.

4.3.2 decode.v

The next module the data encounters is the decode module which is responsible for the flow of data into and out of the system at a rate that the other submodules can handle it. It creates the instances of all the other submodules and facilitates communication between them so as not to overload the system with data. Each submodule can report to the decode module whether they have too much data or not enough data, allowing the decode module to stall the pipeline or the input data until the system has caught up. This submodule also implements the ability to change the size of the input data based on the subsampling rates, and is responsible for making sure the output data is aligned when it is written.

4.3.3 blocker.v and header.v

The blocker module takes data from the input FIFO and splits it into 8-bit chunks so that the next module, the header module, can take those bytes and parse the header information. The header module reads this data and passes important information such as Huffman tables, Quantization tables, and image properties to the decode module. This information will be used throughout the rest of the image decode and is extremely important to the functionality of the SoC module.

4.3.4 stream.v and huff.v

The stream module takes the output of the header module and is designed to feed that data, bit by bit, to the huff module, which is responsible for the Huffman decoding of the data. Since the JPEG codec only allows Huffman lengths of up to 16 bits, the design can use 16 subtractors, one at each bit-length or tree depth, that use the
left-most value at that depth of the tree as an offset to quickly determine Huffman decoded values. Figure 4.3 shows an example of a tree, where storing the left-most elements at each level in a lookup table can greatly reduce the amount of hardware needed to lookup a value in the Huffman Table.

![Figure 4.3: JPEG DC Huffman Tree Example](image)

The Huffman module keeps track of the number of codes at each depth, as well as the code and the bit representation for that code. The hardware performs all 16 subtractions on the input in question and looks for positive result from the lowest level on the tree (or the highest bit length), which indicates that the code being searched for is on that level of the tree.

For example, using Figure 4.4, if the input was 1110b, the subtractions from bit-
lengths 5-16 would result in overflow. The lookup table containing the codes are in order, allowing the code to be extracted by knowing the position of the code for the first 4-bit word, plus an offset. In this case, $5 + (1110b - 1100b) = 7$, so the 7th element is extracted from the lookup table.

<table>
<thead>
<tr>
<th>Length</th>
<th>Bits</th>
<th>Code (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bits</td>
<td>00</td>
<td>05</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>06</td>
</tr>
<tr>
<td>3 bits</td>
<td>100</td>
<td>04</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>07</td>
</tr>
<tr>
<td>4 bits</td>
<td>1100</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td>02</td>
</tr>
<tr>
<td></td>
<td>1110</td>
<td>03</td>
</tr>
<tr>
<td>5 bits</td>
<td>11110</td>
<td>08</td>
</tr>
<tr>
<td>6 bits</td>
<td>111110</td>
<td>00</td>
</tr>
<tr>
<td>7 bits</td>
<td>1111110</td>
<td>09</td>
</tr>
</tbody>
</table>

Figure 4.4: Huffman Table corresponding to Figure 4.3

Another feature of the Huffman module is that it uses a predictor to indicate to the Stream module, which is feeding it data, how many bits should be skipped to start obtaining the next Huffman word. This is accomplished by feeding the length of the payload back to the Stream module and having it skip that number of bits. Allowing the Stream module to shift before the current operation is done allows the system to save 1 cycle per Huffman decode by having the next codeword ready before it is needed.

There are three concurrent processes in the Huff and Stream modules, one to decode the Huffman data, the second to process it, which involves reading the payload,
and the third to store it in a buffer for that block. The predictor helps by maximizing
the parallelism of the two modules, allowing the Stream module to have the next
Huffman code ready for decode in most cases.

4.3.5 idctcol.v and idctrow.v

The two IDCT modules are responsible for performing the 2D IDCT separated into a
row operation followed by a column operation. Loefflers IDCT is the implementation
used in these two modules, each taking 14 cycles from final input to final output.
The two modules are both split into separate stages because the system cannot feed
enough data to the 2D IDCT for it to be implemented in a single cycle.

![Figure 4.5: Loeffler’s 8-point IDCT](image)

4.3.6 colourmap.v

Following the 2D IDCT, the colourmap module takes the decoded data and converts
it from YCC or CMYK to BGRA. Blue-Green-Red-Alpha (BGRA) was chosen as the
output because it matches the framebuffer format for the ZedBoard, allowing for the decoded images to be displayed on an HDMI connected monitor.

4.4 Software Interface

A fully customized software interface was designed in C to interact with the hardware via its software accessible registers. The software has no role in the actual JPEG decode, and is present only to control and check the status of the hardware.

4.4.1 Memory Organization

The ZedBoard contains 512 MB of DDR3 RAM, of which most is used by the Linux kernel as main memory. By passing an argument to the kernel as it is booting, a portion of this memory is reserved and the kernel does not allocate it. This shared area of memory can be used to communicate large chunks of data between the hardware and the software. The shared memory area is split into two buffers, a read buffer and a write buffer, and because of the compressed nature of the JPEG, the write buffer is many times the size of its counterpart. Figure 4.6 shows how memory was allocated for this custom solution.
4.4.2 Software Responsibilities

The software starts by resetting the hardware so it is in a deterministic state. The read buffer in shared memory is then set to an arbitrary size. The size of the read buffer should be realistic and based on the amount of available memory and the size of the image. The image file is then opened so that the read buffer can be filled. The first read is important because it fills control registers with important values such as image size and subsampling factor, as well as allowing the Huffman tables and Quantization tables to be decoded by hardware.

During the initial read, the size of the write job is set to zero so that no information is written to the write buffer. The reason this is done is to allow the software to read
important information from the hardware that is used to determine the size of the write buffer and subsequent write jobs. Without doing this, the software would have no information on the size of the image and would not be able to control file writing to properly output the image.

Now that the software can properly set a write size, the program enters its main loop, in which it polls the software for completion of either a write job or a read job and starts the next corresponding job. Upon a write job finish, the information is read from memory and written to file, and upon read job completion, the next block of information is read from the JPEG file and written to the read buffer. When an image decode is finished, a bit is set in the status register and the software is designed to clean up and exit its execution.

### 4.5 Summary

Presented in this chapter was a fully custom hardware solution for decoding JPEG images. The next chapter will introduce the methods used to test the SoC Module’s performance in two aspects, speed and accuracy, as well as present the results of these tests.
Chapter 5

Results

This chapter presents results that were generated during the testing of the proposed Custom SoC Module. It also describes the methodology used to test the solution for accuracy and speed.

5.1 Test Structure

The goals of testing the module were to determine its characteristics such as speed and accuracy, while removing delays associated with I/O to obtain accurate test results. Removing I/O associated delays from the measurements was important because of the massive difference between the configuration of the ZedBoard, which uses a Network File System, and a smartphone, which uses high performance flash memory. In addition to this, the benchmark would far outperform the module if the images converted by the desktop workstation were stored on a SATA drive and the ZedBoard was limited to using its SD Card. This section describes the efforts made to mitigate
the different factors associated with testing two vastly different systems.

5.1.1 ZedBoard Configuration

The Linux kernel on the ZedBoard has a multitude of customization options, one of which is the ability to have the root filesystem, on which the Linux system is stored, be on a Network Filesystem (NFS). This option was a great opportunity to have the workstation filesystem double as the ZedBoard filesystem, greatly reducing the differences between the two test environments. The increased overhead of the NFS root filesystem running over Gigabit Ethernet (GbE) pales in comparison to the increase in filesystem speed and responsiveness over the SD Card interface at 10 MB/s.

The ZedBoard relies on a binary file to do a few things during its boot sequence and this file is stored on the SD Card. The boot binary is responsible for programming the FPGA portion of the board, as well as pointing the First Stage Bootloader (FSBL) to the kernel executable and the device tree file, which tells the kernel the types of hardware that are available to it and their respective addresses. When the kernel takes over the boot process, it continues with a standard Linux boot that presents the user with a command line interface. The user can then develop software that can operate in conjunction with the FPGA, without having to reboot the board every time a change in software is made. This creates a very user friendly development environment in hardware prototyping.

5.1.2 Test Image Database

The images used to test the SoC Module were obtained by scraping Flickr, a popular image hosting website, for images that were license-free and taken from an iPhone 6S. This ensures that the photos were taken from one of the most recent smartphones.
and allows for the images to be made available upon release of this thesis. The image
database totals 8760 images, giving a fairly good overall coverage of what the average
users image might be.

5.1.3 Testing Process

The testing process is made up of several sub-processes that were optimized to be as
efficient as possible with the hardware available. These processes are split between
running on the ZedBoard when necessary, and the workstation, which is a much more
powerful machine. Because the workstation and the ZedBoard share a filesystem,
file-based multiprocessing was implemented using BASH scripting on both machines.

The process starts with the workstation decoding a JPEG into what is referred to
as the golden image. This golden image is the result of the libjpeg decoder running
with the float option for IDCT, it will be used as a benchmark for comparison later in
the testing process. In order to save storage space on the shared filesystem, a hashing
algorithm, MD5, is used to create a signature of the decoded image and will be used
to compare the results of ZedBoard running the exact same configuration of libjpeg
only.

The ZedBoard decodes the image, performs the MD5 hash, and compares it to
the workstation generated MD5. This is a sanity check as the MD5 hashes should
always match between the ZedBoard and the workstation because the code bases
are the same. The ZedBoard is then responsible for creating six additional outputs
for comparison to the golden image. At the end of output generation for one input
image, outputs exist for libjpeg with three switches (float, fast, and slow), libjpeg-
turbo (float, fast, and slow), and a SoC Module output.

The following sub chapters will describe in detail the methodologies used to com-
pare these outputs, as well as describe how the time comparisons were performed,
5. RESULTS

and introduce results for both.

5.2 Testing for Accuracy

In order to test for accuracy between the golden image and the different decoders, a need exists for a way to measure the differences between two images that, to the naked eye, may look exactly the same. A simple eye test might work for images with significant differences in pixel values, but a more concrete method will provide a better understanding of the differences between different decoding methods.

5.2.1 Mean Squared Error

It is possible to use Mean Squared Error (MSE) to create values that represent the differences between the golden image and the output of the decoders. The MSE of a grayscale image is shown in the equation below. This formula is useful for grayscale images, but when the input is an RGB or CMYK image, the increase in colour components will artificially increase the output value of the MSE function. Normalizing this function according to the quantity of colour channels would eliminate bias associated to increasing numbers of colour channels, which is where Peak Signal to Noise Ratio becomes useful.

\[
MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2 \tag{5.1}
\]

5.2.2 Peak Signal to Noise Ratio

Peak Signal to Noise Ratio (PSNR) is a common way to measure the power of a signal, which in this case would be the output image from the set of decoders. PSNR is a fractional logarithm that relies on the maximum value of a sample to scale the
MSE. This allows for different colour schemes to be measured easily by adjusting the max value, shown in Equation 5.2.

\[
PSNR = 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE)
\]  \hspace{1cm} (5.2)

where

\[
MAX_I = 2^B - 1
\]  \hspace{1cm} (5.3)

where B is defined as the number of bits per sample

### 5.2.3 Accuracy Results

Using libjpeg-turbo as a benchmark, because of its widespread use in mobile application development, the hardware image output is measured using PSNR. PSNR is a measure of signal power, which means that if two images match exactly, the output of the function will be infinity, which is not able to be shown on a graph, so the output was modified from infinity to a value of 300dB.

A moderate grouping of "perfect" decodes by libjpeg-turbo with the float option for DCT can be seen at the top of the chart, as well as a grouping of images that were more accurate than hardware around the 200dB mark in Figure 5.1. But for the majority of images, the hardware was able to outperform the software because the software rounds values after every stage, whereas the hardware was built to round only in the late stages of the decode.
5.3 Testing for Speed

An important metric in hardware acceleration is the speedup that results on a given process. It is important to know what to measure and how to measure it so there are no false positives in the measurement process. This was difficult to ascertain for this project because of the hardware focused nature of the SoC Module. Measuring the hardware in clock cycles and the software in time, the most accurate way to make a comparison is to eliminate as many of the variables surrounding the two tests as possible.

When measuring the time software spent decoding a certain image, the measurement was made using C standard library function called getrusage. This allowed...
for the separation of user time and system time, where user time is the time spent actually running the software, excluding system events such as disk reads and writes. Removing system events from the measurement allowed for the direct comparison of user time to the number of clock cycles that hardware spent decoding that same image. On top of getrusage, a switch was implemented in the software to disable disk writes entirely so a speed run of each of the seven decoders could be done during processing.

The clock cycles are measured by hardware counters that are software accessible. Three measures of clock cycles occur: one when only the read portion of the hardware is active, one when only the write portion of the hardware is active, and one when both portions are active. Summing these three counters gives an accurate account of how much work was done because it discounts periods of time when neither system is active due to data stalls from software.

5.3.1 Speed Results

To present an argument that mobile CPUs have not yet caught up to Desktop CPUs, a comparison was done using libjpeg-turbo (float) of decode times on the ZedBoard and a workstation. The workstation is a mid-level machine from 2010 that features an Intel Xeon E5450 4-core CPU at a clock speed of 3.0GHz, and 8 GB of ECC DDR3. Figure 5.2 shows that the older workstation CPU far outperforms the mobile ARM Cortex-A9 on every image.
Again using libjpeg-turbo as a benchmark, decode times were compared. There is a large grouping, in Figure 5.3, where decode times for images under 10 megapixels are similar, but hardware still shows an improvement over software. Where the greatest improvement is seen is for images over 10 MB and especially on those over 15 MB. On average, hardware was 5.57 times faster than libjpeg-turbo with float DCT, and 4.14 times faster than libjpeg-turbo with fast DCT. Combine the speed increases with the results shown in Section 5.2, and the SoC Module vastly outperforms software even on entry level FPGAs.
5. RESULTS

Figure 5.3: Decode Time Results

To summarize the performance of the system, the average pixel processing speed was found to be 0.494 pixels/cycle, and the Huffman payload processing speed was 2.259 cycles/payload.

5.4 Hardware Reports

The overall design of the SoC Module uses roughly 18% of the available resources on the FPGA, as seen in Figure 5.4. This is possible because there is no need to implement a soft processor on the FPGA. Figure 5.4 shows the design was easily able to fit on the ZedBoard FPGA.
5. RESULTS

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Used</th>
<th>Available</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of RAMB18E1s</td>
<td>7</td>
<td>280</td>
<td>2%</td>
</tr>
<tr>
<td>Number of RAMB36E1s</td>
<td>1</td>
<td>140</td>
<td>1%</td>
</tr>
<tr>
<td>Number of Slices</td>
<td>2525</td>
<td>13300</td>
<td>18%</td>
</tr>
<tr>
<td>Number of Slice Registers</td>
<td>3508</td>
<td>106400</td>
<td>3%</td>
</tr>
<tr>
<td>Number of Slice LUTS</td>
<td>6917</td>
<td>53200</td>
<td>13%</td>
</tr>
<tr>
<td>Number of Slice LUT-FF pairs</td>
<td>7386</td>
<td>53200</td>
<td>13%</td>
</tr>
</tbody>
</table>

Figure 5.4: Resource Usage on the FPGA

There were limitations caused by the software used to develop the solution. The CPU/FPGA design could not be simulated because of the number of pins required to simulate a physical CPU, which caused significant delays in the design process. Physical debugging is extremely difficult compared to simulation debugging, but it was necessary because of the design of the ZedBoard.

Xilinx’s software has its limitations as well, providing vague information on the critical path of the design. This made it difficult to make incremental improvements on the design. The critical path was found between the Stream module and the Huff module, which was expected due to the bit-by-bit nature of the JPEG data stream. Although the FPGA software validated a successful implementation at 66MHz, in practice it was unstable. The next lowest option of 50MHz was used for all testing.

An option to combat all of these issues was to use a more expensive development kit, but the goal of the project was to implement it on hardware that was inexpensive. This was in an effort to show that consumer accessible hardware could run the design and provide a speedup for a very low cost.
5. RESULTS

5.5 Summary

It was shown in this chapter that full hardware acceleration of JPEG decoding has clear and distinct advantages over its software counterparts. A 5x speedup on FPGA combined with the increased accuracy for greater visual fidelity, if this SoC Module were to be implemented as an ASIC and put alongside a CPU as a coprocessor, it would greatly reduce the strain on the CPU and enhance user experience.
Chapter 6

Summary

6.1 Conclusions

The popularity of the JPEG codec makes it an excellent candidate for hardware acceleration. Furthering this candidacy is the rapid advancements in image sensor technology and smartphone display technology. Licensing issues have dictated the software market for many years, causing the image codec monopoly that is currently held by the JPEG. A need for faster image decompression exists where the vast majority of images rely on a 25-year-old codec that was not built with multiprocessing in mind.

This work presented an all-encompassing solution for JPEG decoding using FPGA hardware. The benefits of hardware acceleration are clear, with the proposed solution outperforming software solutions at a fraction of the clock rate. The added benefit of offloading the work to a dedicated coprocessor would allow the CPU of a hetero-
geneous system to perform other tasks, providing a better overall user experience.

The architecture presented in this work is entirely novel at the time of writing. No other publicly available solution presents a hardware module that can decode a JPEG in its entirety, only relying on the CPU for memory transfers.

Taking into account the large number of IP cores being built into modern SoC designs, this module is small enough to be added to those designs with a minimal increase in size and cost, as shown in Figure 5.4.

Additionally, a full ASIC implementation would improve the speed of the circuit, with Kuon and Rose [17] showing that ASIC implementations provide speedups of about 4x against FPGA implementations of the same circuit in a 90nm process.

6.2 Recommendations for Future Work

The implementation presented in this work substantially improves the process of decoding a JPEG, however it is not a market ready solution. A Linux kernel driver, if properly designed and implemented, would greatly improve performance and allow for a more seamless experience on the side of the end-user. Proper context switching in the driver would allow for multiple images to be decoded at the same time. Two or more processes could submit decode jobs that would be switched based on time-slice scheduling so process B does not have to wait for process A to finish its decode.
References


Appendix A

Verilog Code

A.1 user_logic.v

module user_logic
(
    S_AXI_ACLK,
    S_AXI_ARSENSETN,
    S_AXI AWADDR,
    S_AXI AWVALID,
    S_AXI AWREADY,
    S_AXI_WRDATA,
    S_AXI_WSTRB,
    S_AXI WVALID,
    S_AXI WREADY,
    S_AXI_BRESP,
    S_AXI_BVALID,
    S_AXI_BREADY,
    S_AXI ARADDR,
    S_AXI ARVALID,
    S_AXI ARREADY,
    S_AXI RDATA,
    S_AXI RRESP,
);
A. VERILOG CODE

S_AXI_RVALID,
S_AXI_RREADY,
m_axi aclk,
m_axi aresetn,
m_axi arready,
m_axi arvalid,
m_axi araddr,
m_axi arlen,
m_axi arsize,
m_axi arburst,
m_axi arprot,
m_axi arcache,
m_axi rready,
m_axi rvalid,
m_axi rdata,
m_axi rresp,
m_axi rlast,
m_axi awready,
m_axi awvalid,
m_axi awaddr,
m_axi awlen,
m_axi awsize,
m_axi awburst,
m_axi awprot,
m_axi awcache,
m_axi wready,
m_axi wvalid,
m_axi wdata,
m_axi wstrb,
m_axi wlast,
m_axi bready,
m_axi bvalid,
m_axi bresp
);

// user_logic

input S_AXI_ACLK;
input S_AXI_ARESETE;
A. VERILOG CODE

output S_AXI_BVALID;
input S_AXI_BREADY;
input [31:0] S_AXI_ARADDR;
input S_AXI_ARVALID;
output S_AXI_ARREADY;
output [31:0] S_AXI_RDATA;
output S_AXI_RVALID;
output S_AXI_RREADY;
input m_axi_aclk;
input m_axi_arresetn;
inout m_axi_arready;
output m_axi_arvalid;
output [31:0] m_axi_araddr;
output 7:0 m_axi_arlen;
output 2:0 m_axi_arsize;
output 1:0 m_axi_arburst;
output 2:0 m_axi_arprot;
output 3:0 m_axi_arcache;
output m_axi_rready;
inout m_axi_rvalid;
inout [31:0] m_axi_rdata;
inout 1:0 m_axi_rresp;
inout m_axi_rlast;
inout m_axi_awready;
output m_axi_awvalid;
output [31:0] m_axi_awaddr;
output 7:0 m_axi_awlen;
output 2:0 m_axi_awsize;
output 1:0 m_axi_awburst;
output 2:0 m_axi_awprot;
output 3:0 m_axi_awcache;
inout m_axi_wready;
output m_axi_wvalid;
output [31:0] m_axi_wdata;
output [3:0] m_axi_wstrb;
output m_axi_wlast;
output m_axi_bready;
inout m_axi_bvalid;
inout [1:0] m_axi_bresp;

//--
//-- Implementation
//--
parameter writefifodepth = 10;
reg [writefifodepth-1:0] writefiforp, writefifowp;
wire [writefifodepth-1:0] writefiforpt, writefifocheck;
wire [31:0] writefifoout;
reg [31:0] writefifo;
reg writefifoready;
reg writefifovalid;
reg [writefifodepth-1-3:0] writeforpa;
wire [writefifodepth-1-3:0] writefowpa;
wire [31-3:0] writefifoouta;
reg [31-3:0] writefioa;

reg rS_AXI_ARREADY;
reg rS_AXI_AWREADY;
reg [1:0] rS_AXI_BRESP;
reg rS_AXI_BVALID;
reg rS_AXI_ARREADY;
reg [31:0] rS_AXI_RDATA;
reg [1:0] rS_AXI_RRESP;
reg rS_AXI_RVALID;

assign S_AXI_ARREADY = rS_AXI_ARREADY;
assign S_AXI_AWREADY = rS_AXI_AWREADY;
assign S_AXI_BRESP = rS_AXI_BRESP;
assign S_AXI_BVALID = rS_AXI_BVALID;
assign S_AXI_ARREADY = rS_AXI_ARREADY;
assign S_AXI_RDATA = rS_AXI_RDATA;
assign S_AXI_RRESP = rS_AXI_RRESP;
assign S_AXI_RVALID = rS_AXI_RVALID;

reg axi_arvalid;
reg [31:0] axi_araddr;
reg [7:0] axi_arlen;
reg [2:0] axi_arsize;
reg [1:0] axi_arsize;
reg [2:0] axi_arburst;
reg [0:0] axi_arprot;
reg [3:0] axi_arcache;
reg axi_awvalid;
reg [31:0] axi_awaddr;
reg [7:0] axi_awlen;
reg [2:0] axi_awsize;
reg [1:0] axi_awburst;
reg [2:0] axi_awprot;
reg [3:0] axi_awcache;
A. VERILOG CODE

```verilog
reg rm_axi_wvalid;
reg [3:0] rm_axi_wstrb;
reg rm_axi_wlast;
reg rm_axi_bready;

assign m_axi_arvalid = rm_axi_arvalid;
assign m_axi_araddr = rm_axi_araddr;
assign m_axi_arlen = rm_axi_arlen;
assign m_axi_arsize = rm_axi_arsize;
assign m_axi_arburst = rm_axi_arburst;
assign m_axi_arprot = rm_axi_arprot;
assign m_axi_arcache = rm_axi_arcache;
assign m_axi_awvalid = rm_axi_awvalid;
assign m_axi_awaddr = rm_axi_awaddr;
assign m_axi_awlen = rm_axi_awlen;
assign m_axi_awsize = rm_axi_awsize;
assign m_axi_awburst = rm_axi_awburst;
assign m_axi_awprot = rm_axi_awprot;
assign m_axi_awcache = rm_axi_awcache;
assign m_axi_wvalid = rm_axi_wvalid;
assign m_axi_wdata = writefifoout;
assign m_axi_wstrb = rm_axi_wstrb;
assign m_axi_wlast = rm_axi_wlast;
assign m_axi_bready = rm_axi_bready;

wire [31:0] binp;
wire binpvalid;

assign binp = { m_axi_rdata[0+:8], m_axi_rdata[8+:8], m_axi_rdata[16+:8], m_axi_rdata[24+:8] };
assign binpvalid = m_axi_rvalid;

assign writefifowpa = writefifowp[3+:writefifodepth-3];

dpsram #(writefifodepth,32) Uwritefifo ( m_axi_aclk, ( writefifovalid && writefifoready), writefifowp, writefifo, writefiforpt, writefifoout );
dpsram #(writefifodepth-3,32-3) UwritefifoA ( m_axi_aclk, (writefifovalid && writefifoready && writefifowp[0+:3]==0), writefifowpa, writefifoa, writefiforpa , writefifoouta );

wire [8*32−1:0] readregs;
```
wire startin;
wire [31:0] outcode;
wire [31–3:0] outaddr;
wire outvalid;
wire outlast;
wire burst16;

reg writefifofull;

decode10d Udecode(
    m_axi_aclk, readregs, startin, binp, binpvalid, m_axi_rready, burst16, outcode, outvalid,
    outlast, outaddr, writefifofull);

reg [3:0] job_read, job_write, writeresp;
reg job_done;
reg writeindone;

always @(posedge m_axi_aclk)
begin
    if (startin==1)
        begin
            writefifovvalid <= 0;
            writeindone <= 0;
            writefifofull <= 0;
            job_done <= 0;
        end
    else
        begin
            writefifofull <= (writefifocheck >= (512+256+128+64+32));
            if (outlast)
                begin
                    $display("GOT LAST! %1d",writefifocheck);
                    writeindone <= 1;
                end
            if (writeindone && writefifocheck==0)
                begin
                    job_done <= 1;
                end
            if (outvalid && !writeindone)
                begin
                    writefif0 <= outcode;
                    writefioa <= outaddr;
                    writefifovvalid <= 1;
                end
        end
end
else
begin
  writefifovalid <= 0;
end
end

reg [1:0] SRSTATE;

reg [31:0] measure_readaddr,measure_readdata,measure_writeaddr,measure_writedata,
          measure_writeresp,measure_burstread,measure_work;
reg [31:0] gk_wrcount,gk_rdcount,gk_bothcount;
reg gk_wractive, gk_ractive;

// Nets for user logic slave model s/w accessible register example
wire [31 : 0] myreadregs[0:15];
reg [31 : 0] mywriteregs[0:7];

assign myreadregs[0] = { 'h16'hbeef, job_done, 3'b000, job_read, 4'b0000, job_write };
assign myreadregs[1] = gk_wrcount://measure_writeaddr;
assign myreadregs[4] = measure_burstread;
assign myreadregs[5] = measure_readaddr;
assign myreadregs[6] = measure_readdata;
assign myreadregs[7] = measure_work;

assign myreadregs[8] = readregs[ 7*32+:32 ];
assign myreadregs[9] = readregs[ 6*32+:32 ];
assign myreadregs[10] = readregs[ 5*32+:32 ];
assign myreadregs[12] = readregs[ 3*32+:32 ];
assign myreadregs[14] = readregs[ 1*32+:32 ];
assign myreadregs[15] = readregs[ 0*32+:32 ];

assign startin = mywriteregs[0][4];
// measure work
always @(posedge m_axi_aclk)
begin
    if (m_axi_arresetn == 1'b0)
        begin
            gk_wrcount <= 0;
            gk_rdcount <= 0;
            gk_bothcount <= 0;
        end
    else
        begin
            if (gk_wractive && !gk_rdactive) gk_wrcount <= gk_wrcount + 1;
            else if (gk_rdactive && !gk_wractive) gk_rdcount <= gk_rdcount + 1;
            else if (gk_wractive && gk_rdactive) gk_bothcount <= gk_bothcount + 1;
            if (job_done && startin)
                begin
                    gk_wrcount <= 0;
                    gk_rdcount <= 0;
                    gk_bothcount <= 0;
                end
        end
end
end

// read lite
always @(posedge S_AXI_ACLK)
begin
    rS_AXI_RRESP <= 0;
    if (S_AXI_ARESETN == 1'b0)
        begin
            rS_AXI_ARREADY <= 0;
            rS_AXI_RVALID <= 0;
            SRSTATE <= 2;
        end
    else
        begin
            case (SRSTATE)
                0:
                    begin
                        if (S_AXI_ARVALID && rS_AXI_ARREADY)
                            begin
                                rS_AXI_RDATA <= myreadregs[S_AXI_ARADDR[2+:4]];
                                rS_AXI_ARREADY <= 0;
                            end
                    end
            default:
                begin
                    rS_AXI_ARREADY <= 0;
                end
            endcase
        end
end
A. VERILOG CODE

rS_AXI_RVALID <= 1;
SRSTATE <= 1;
end
else
begin
rS_AXI_RVALID <= 0;
end
end
1:
begin
if ( S_AXI_RREADY )
begin
rS_AXI_RVALID <= 0;
rS_AXI_ARREADY <= 1;
SRSTATE <= 0;
end
end
default:
begin
rS_AXI_ARREADY <= 1;
rS_AXI_RVALID <= 0;
SRSTATE <= 0;
end
endcase
end
end
derg [1:0] SWSTATE;
derg [31:0] SAWADDR;
derg [31:0] SWDATA;
derg [3:0] SWSTRB;
der SWA, SWD;

// write lite
always @(posedge S_AXI_ACLK )
begin
rS_AXI_BRESP <= 0;
if ( S_AXI_ARRESETN == 1'b0 )
begin
rS_AXI_ARREADY <= 0;
rS_AXI_WREADY <= 0;
rS_AXI_BVALID <= 0;
SWSTATE <= 2;
mywritereg[0] <= 32'h00000010;
A. VERILOG CODE

mywritergs[1] <= 32'h00000000; // read address
mywritergs[2] <= 32'h00000000; // size in 16 word blocks (or 64 byte chunks)
mywritergs[3] <= 32'h00000000; // write address
mywritergs[4] <= 32'h00000000; // size in 32 byte chunks
mywritergs[5] <= 0;
mywritergs[6] <= 0;
mywritergs[7] <= 0;
end
else
begin
  case (SWSTATE)
  0:
    begin
      SWA = ( S_AXI_AVALID && rS_AXI_AREADY );
      SWD = ( S_AXI_WVALID && rS_AXI_WREADY );
      if (SWA)
        begin
          SAWADDR <= S_AXI_AWADDR;
          rS_AXI_AWREADY <= 0;
        end
      if (SWD)
        begin
          SWDATA <= S_AXI_WDATA;
          SWSTRB <= S_AXIWSTRB;
          rS_AXI_WREADY <= 0;
        end
      if ( (SWA && SWD) || (SWA && !rS_AXI_WREADY) || (SWD &&
          !rS_AXI_AWREADY) )
        begin
          rS_AXI_BVALID <= 1;
          SWSTATE <= 1;
        end
    end
  1:
    begin
      if (SWSTRB[3]) mywritergs[SAWADDR[2+:3]][24+:8] <= SWDATA[24+:8];
      if (SWSTRB[2]) mywritergs[SAWADDR[2+:3]][16+:8] <= SWDATA[16+:8];
      if (SWSTRB[1]) mywritergs[SAWADDR[2+:3]][8+:8] <= SWDATA[8+:8];
      if (SWSTRB[0]) mywritergs[SAWADDR[2+:3]][0+:8] <= SWDATA[0+:8];
      if (S_AXI_BREADY)
        begin
          rS_AXI_AWREADY <= 1;
          rS_AXI_WREADY <= 1;
          rS_AXI_BVALID <= 0;
        end
  end

end
A. VERILOG CODE

```verilog
SWSTATE <= 0;
end
end
default:
begin
    rS_AXI_AWREADY <= 1;
    rS_AXI_WREADY <= 1;
    rS_AXI_BVALID <= 0;
    SWSTATE <= 0;
end
endcase
end
end

reg [2:0] MRSTATE;
reg [3:0] MRTAG;
reg [3:0] MRLENGTH;
reg [32-1:0] MRADDR;
reg [28-1:0] MRSIZE; // 16 word blocks

parameter mridle = 0, mraddrset = 1, mraddrwait = 2, mrdatapread = 3;

// read burst
always @(posedge m_axi_aclk )
begin

    rm_axi_arsize <= 2; // 4 byte transfers all the time
    rm_axi_arburst <= 1; // INCR transfers all the time
    rm_axi_arcache <= mywriteregs[0][24+:4]; // don’t know what these values should be right now
    rm_axi_arprot <= mywriteregs[0][28+:3]; // ditto
    if ( m_axi_arsetn == 1'b0 )
begin
        rm_axi_arvalid <= 0;
        MRTAG <= 0;
        MRSTATE <= mridle;
        gk_rdactive <= 0;
    end
else
begin
case (MRSTATE)
mridle: // wait for command to start read burst
```
begin
  rm_axi_arvalid <= 0;
gk_rdactive <= 0;
if (mywriterregs[0][8+:4]!=MRTAG)
begin
  if (job_done)
  begin
    job_read <= mywriterregs[0][8+:4];
  end
  else
  begin
    MRTAG <= mywriterregs[0][8+:4];
    MRADDR <= {mywriterregs[1][6+:26],6'b000000}; // address must have lower 6 bits as 0.
    MRSIZE <= mywriterregs[2][0+:28];
    MRSTATE <= mraddrset;
    measure_burstread <= 0;
    measure_readaddr <= 0;
    measure_readdata <= 0;
  end
end
mraddrset: // set initial address
begin
  gk_rdactive <= 1;
  measure_burstread <= measure_burstread + 1;
  if (MRSIZE==0 || writeindone || job_done || startin)
  begin
    rm_axi_arvalid <= 0;
    job_read <= MRTAG;
    MRSTATE <= mridle;
  end
else if (burst16) // wait for internal fifo to have atleast 16 empty entries on it
begin
  // start read
  rm_axi_arvalid <= 1;
  rm_axi_araddr <= MRADDR;
  rm_axi_arlen <= 15;
  MRLENGTH <= 15;
  MRSTATE <= mraddrwait;
  MRADDR <= MRADDR + (16*4); // new address is +16 words
  MRSIZE <= MRSIZE - 1'b1;
end
else
begin
    rm_axi_arvalid <= 0;
end
end

mraddrwait: // wait for address transaction to complete
begin
    gk_ractive <= 1;
    measure_readaddr <= measure_readaddr + 1;
    if ( rm_axi_arvalid && m_axi_arready )
    begin
        // they got it, move on to receiving data, clear address valid
        rm_axi_arvalid <= 0;
        MRSTATE <= mrdataread;
    end
    // keep waiting if they didn’t get it, probably should have a timeout here
end

mrdataread: // wait for data transaction to complete
begin
    gk_ractive <= 1;
    rm_axi_arvalid <= 0;
    measure_readdata <= measure_readdata + 1;
    if ( m_axi_rvalid && m_axi_rready )
    begin
        // we got it, are we done?
        if (MRLENGTH) // ignoring m_axi_rlast right now
        begin
            // move on to receiving next chunk of data
            MRLENGTH <= MRLENGTH - 1'b1;
        end
        else
        begin
            MRSTATE <= mraddrset;
        end
    end
    // keep waiting to get it, probably should have a timeout here
end
default:
begin
    gk_ractive <= 0;
    rm_axi_arvalid <= 0;
    MRSTATE <= mridle;
end
endcase
end
A. VERILOG CODE

end

assign writefifocheck = writefifowp - writefiforp;
assign writefiforpt = ( rm_axi_wvalid && m_axi_wready ) ? writefiforp + 1'b1 : writefiforp;

always @(posedge m_axi_aclk)
begin
  if (m_axi_aresetn == 1'b0)
    begin
      writefifowp <= 0;
      writefifoready <= 0;
    end
  else
    begin
      if (startin == 1)
        begin
          writefifowp <= 0;
          writefifoready <= 0;
        end
      else
        begin
          writefifoready <= 1;
          if (writefifovalid && writefifoready)
            begin
              writefifowp <= writefifowp + 1'b1;
            end
        end
    end
end

reg [2:0] MWSTATE;
reg [3:0] MWTAG;
reg [3:0] MWLENGTH, lengthtemp;
reg [31:0] MWADDR;
reg [29:1:0] MWSIZE; // 8 word blocks

parameter mwidle = 0, mwaddrset = 1, mwaddrwait = 2, mwdatawrite = 3, mwrespwait = 4;
reg [31:0] rm_axi_awaddr_temp;

// write burst
always @(posedge m_axi_aclk )
begin

  rm_axi_awsize  <= 2; // 4 byte transfers all the time
  rm_axi_awburst <= 1; // INCR transfers all the time
  rm_axi_awcache <= mywriterregs[0][16+:4]; // don’t know what these values should be right now
  rm_axi_awprot  <= mywriterregs[0][20+:3]; // ditto
  if ( m_axi_arresetn == 1’b0 )
  begin
    rm_axi_awvalid  <= 0;
    rm_axi_wvalid  <= 0;
    rm_axi_bready  <= 0;
    MWSTATE  <= mwidth;
    MWTAG  <= 0;
    MWLENGTH  <= 0;
    writefiforp  <= 0;
    writefiforpa  <= 0;
    measure_work  <= 0;
    gk_wractive  <= 0;
  end
  else
  begin
    measure_work  <= measure_work + 1’b1;
    rm_axi_bready  <= 1;
  end
  case (MWSTATE)
    mwidth: // wait for start command
    begin
      gk_wractive <= 0;
      rm_axi_wvalid <= 0;
      rm_axi_awvalid <= 0;
      if ( startin )
      begin
        writefiforp <= 0;
        writefiforpa <= 0;
      end
      if ( mywriterregs[0][0+:4]!=MWTAG )
      begin
        if ( job_done )
        begin
          job_write <= mywriterregs[0][0+:4];
        end
      end
  end
end
else
begin
    // setup write
    MWTAG <= mywriteregs[0][0+:4];
    MWADDR <= {mywriteregs[3][5+:27],5'b00000}; // address must have
    lower 5 bits as 0.
    MWSIZE <= mywriteregs[4][0+:27];
    MWSTATE <= mwaddrset;
    measure_writeaddr <= 0;
    measure_writedata <= 0;
    measure_writeresp <= 0;
end
end
mwaddrset: // wait for write command
begin
    gk_wractive <= 1;
    rm_axi_wvalid <= 0;

    measure_writeresp <= measure_writeresp + 1;

    if ((writeindone && writefifocheck==0) || startin )
begin
    // got last signal and all writes are flushed
    rm_axi_awvalid <= 0;
    job_write <= MWTAG;
    MWSTATE <= mwidle;
end
if (writefifocheck>=8)
begin
    // check if write area has been exceeded
    if (writefifoouta<MWSIZE)
begin
    // start write
    rm_axi_awvalid <= 1;
    rm_axi_awaddr_temp = MWADDR + (writefifoouta << 5);

    rm_axi_awaddr <= rm_axi_awaddr_temp;

    writefiforpa <= writefiforpa + 1'b1;
    rm_axi_awlen <= 7;
A. VERILOG CODE

MWLENGTH <= 7;
MWSTATE <= mwaddrwait;

end
else
begin
  // Leave, this writ chunk is done
  rm_axi_awvalid <= 0;
  job_write <= MWTAG;
  MWSTATE <= mwidle;
end
end
else
begin
  rm_axi_awvalid <= 0;
end
end

mwaddrwait: // wait for address transaction to complete
begin
  gk_wractive <= 1;
  measure_writeaddr <= measure_writeaddr + 1;
  if ( rm_axi_awvalid && m_axi_awready )
    begin
      // they got it, move on to sending data, clear address valid
      rm_axi_awvalid <= 0;
      rm_axi_wvalid <= 1;
      rm_axi_wstrb <= 4'b1111; // all bytes going
      rm_axi_wlast <= ( MWLENGTH == 0 ) ? 1'b1 : 1'b0; // set last accordingly
      MWSTATE <= mwdatawrite;
    end
  // keep waiting if they didn’t get it, probably should have a timeout here
end

mwdatawrite: // wait for data transaction to complete
begin
  gk_wractive <= 1;
  measure_writedata <= measure_writedata + 1;
  if ( rm_axi_wvalid && m_axi_wready )
    begin
      writefiforp <= writefiforp + 1'b1;
      // they got it, are we done?
      if ( MWLENGTH )
        begin
          // move on to sending next chunk of data
A. VERILOG CODE

lengthtemp = MWLENGTH - 1'b1;
rm_axi_wvalid <= 1;
rm_axi_wstrb <= 4'b1111; // all bytes going
rm_axi_wlast <= ( lengthtemp == 0 ) ? 1'b1 : 1'b0; // set last accordingly
MWLENGTH <= lengthtemp;
end
else
begin
  rm_axi_wvalid <= 0;
  rm_axi_wlast <= 0;

  MWSTATE <= mwaddrset;
end
end
// keep waiting if they didn’t get it, probably should have a timeout here
end
case
  end
endcase
end
endmodule

user_logic.v

A.2  decode.v

// 'define DEBUGDECODE
module decode10d(CK,readregs,startin,binp,binpvalid,binpready,burst16,
  finalout , finalvalid , finallast , finaladdr , writefifofull );

parameter oprec=3;
parameter bsize=9; // should be larger, output from column dct can produce larger
  values
parameter startsync=400;
parameter lastsync=130 + 64*4 − 1;

input  CK;
output [8*32−1:0] readregs;
input  startin;
input [31:0] binp;
A. VERILOG CODE

input binpvalid;
output binpready;
output burst16;
output [31:0] finalout;
output finalvalid;
output finallast;
reg finallast;
output [31–3:0] finaladdr;

input writefifofull;
wire startheader, startstream, starthuff;
wire [4+4+26+16+10+2*10+2*10−1:0] huffparams;

// huffparams
wire [3:0] blkmax;
wire [25:0] totalmcu;
wire [15:0] restart;
wire blkindex [0:9];
wire [1:0] blkquant [0:9];
wire [1:0] blkcomp [0:9];

wire [14*2+16*2+3+5+2+2+3+2*4+2*4+2*4+4*4−1:0] imageparams;

// image parameters
wire [14−1:0] sizex, sizey;
wire [16−1:0] dispx, dispy;
wire [2:0] compmax;
wire [4:0] mcusize;
wire [1:0] ssmaxx;
wire [1:0] ssmaxy;
wire [2:0] ssmaxxmask;
wire [1:0] subscalex [0:3];
wire [1:0] subscaley [0:3];
wire [1:0] subsampx [0:3];
wire [3:0] subsize [0:3];
wire [3:0] blocksm1;
wire errorheader, errorhuff;
wire outlast;

reg [31:0] stall_blocker, stall_stream, stall_write;
assign { blkmax, blocksm1, totalmcu, restart, blkindex [0], blkindex [1], blkindex [2],
A. VERILOG CODE

```verilog
blkindex [3], blkindex [4], blkindex [5], blkindex [6], blkindex [7], blkindex [8],
blkindex [9], blkquant [0], blkquant [1], blkquant [2], blkquant [3], blkquant [4],
blkquant [5], blkquant [6], blkquant [7], blkquant [8], blkquant [9], blkcomp[0],
blkcomp[1], blkcomp[2], blkcomp[3], blkcomp[4], blkcomp[5], blkcomp[6],
blkcomp[7], blkcomp[8], blkcomp[9] } = huffparams;

assign { sizex, sizey, dispx, dispy, compmax, mcusize, ssmaxx, ssmaxy, ssmaxxmask,
subsaclex[0], subsaclex[1], subsaclex[2], subsaclex[3], subscaley [0], subscaley [1],
subsacley [2], subsacley [3], subsampx[0], subsampx[1], subsampx[2], subsampx[3],
subsize[0], subsize [1], subsize [2], subsize [3] } = imageparams;

assign readregs = { 2'b00, sizex, 2'b00, sizey, dispx, dispy, ssmaxx, ssmaxy, 1'b0,
compmax, errorheader, errorhuff, 2'b00, startheader, startstream, starthuff,
ouletlast , 16'h0000, 6'b0000000, totalmcu, 32'h00000000, stall_blocker, stall_stream,
stall_write };

wire [2:0] progmode;
wire [9:0] progaddr;
wire [15:0] progdata;
wire [31:0] boutp;
wire boutpvalid;
wire [2:0] bshift ;
wire bshiftvalid ,bshiftready;

// merge

wire tready;
wire tvalid;
wire [31:0] inp;
wire [4:0] shift1 ,shift2 ;
wire [31:0] outp;
wire new;
wire stall ;
wire [2:0] remain;
wire signed [15:0] outcode;
wire signed [15:0] indata;
wire [3:0] outblk;
wire [3:0] outbank, outbank2;
wire [3:0] outbankenable;
wire [6:0] outindex;
wire [6:0] outaddr,inaddr,dctrowaddr,dctcoladdr;
wire [2:0] dctrowindex,dctcolindex;
wire signed [16+3+oprec−1:0] dctrowout,dctcolin;
wire [bsize−1:0] dctcolout;
reg [4+10−1:0] bankraddr;
```

wire [bsize*4−1:0] bankout;
wire [4*4−1:0] outbankslast;
wire [3:0] bankcount[0:3];
wire outcolourgo;
wire outvalid;

assign inaddr = outindex − 64 + 26;
assign dctrowindex = inaddr−1;
assign dctrowaddr = inaddr−13;
assign dctcolindex = inaddr−6;
reg nextoutvalid;
assign { bankcount[0], bankcount[1], bankcount[2], bankcount[3] } = outbankslast;

blocker10 Ublocker(CK,startin,startheader,binp,binpvalid,binpready,bshift, bshiftvalid , bshiftready, boutp, boutpvalid,burst16);
header10d Uheader(CK,huffparams,imageparams,startheader,startstream,boutp,boutpvalid, bshift , bshiftvalid , bshiftready,inp, tvalid ,tready,progmode,progaddr, proddata, errorheader);
stream10c Ustream( CK, startstream, inp, shift1, shift2 , starthuff , outp, tvalid , texture, remain, new, stalls);
huff10d Uhuff(CK,huffparams,progmode,progaddr,proddata,starthuff,outp,shift1,shift2, outcode,outindex, outaddr, outblk, outbank, outbankslast, outcolourgo, outvalid, outlast, remain, new, stall);
dpram #(7,16) Ubuf1(CK, outvalid, outaddr, outcode, inaddr, indata);
idctrowg #(16,11,oprec) Udctrowd(CK, outvalid, dctrowindex, indata, dctrowout);
dpram #(7,16+3+oprec) Ubuf2(CK, outvalid, dctrowaddr, dctrowout, dctcoladdr, dctcolin);
idctcolg #(16+3+oprec,11,oprec,bsize) Udctcol(CK, outvalid, dctcolindex, dctcolin, dctcolout);

wire [10−1:0] bankwaddr;

dpram #(4+6,bsize) Umem0(CK, outbankenable[0] & outvalid, bankwaddr, dctcolout, bankraddr[0*10+10], bankout[0*bsize+:bsize]);
dpram #(4+6,bsize) Umem1(CK, outbankenable[1] & outvalid, bankwaddr, dctcolout, bankraddr[1*10+10], bankout[1*bsize+:bsize]);
dpram #(4+6,bsize) Umem3(CK, outbankenable[3] & outvalid, bankwaddr, dctcolout, bankraddr[3*10+10], bankout[3*bsize+:bsize]);

reg colourgo, proccolour;
reg [3:0] banksel [0:3];
reg [3:0] procbanksel [0:3];
wire [31:0] combined;
wire combinedvalid;
reg [3:0] ix, iy;
reg [5:0] ia, ib;
reg [3:0] iw, iz;
reg [3:0] working2;
colourmap9 Ucolourmap(CK, proccolour, compmax, bankout, combinedvalid, combined);

reg [32−3:0] wbase; // 32−3 bits
reg [16−1:0] wyoffl; // 16 − 3 + 3 = 16 bits
reg [18−1:0] wyoffh; // max is 16 + 2 bits
reg [16−1:0] woxoffl,woxoffhtemp; // image width, 16 bits
reg [5:0] windex;
reg [3:0] wcount;
reg [1:0] wix, wiw, wyoffl;
reg wstart;

always @ (posedge CK)
begin
  if (startin)
  begin
    stall_blocker = 0;
    stall_stream = 0;
    stall_write = 0;
  end
  else
  begin
    if (bshiftready == 0) stall_blocker = stall_blocker + 1;
    if (stall == 1) stall_stream = stall_stream + 1;
    if (writefifofull == 1) stall_write = stall_write + 1;
  end
end

always @ (posedge CK)
begin
  wcount <= outblk;
  wix <= outblk & ssmaxxmask;
  wiw <= outblk >> ssmaxx;
  windex <= outindex;
  if (starthuff == 1)
  begin

A. VERILOG CODE

```
    wbase <= 0;
    wyoffl <= 0;
    woffh <= 0;
    wstart <= 0;
end
else
begin

    if (colourgo==1) wstart <= 1;
    if (proccolour==1 && windex[0+:3]==0)
        begin
            if ( windex[3+:3] == 0) wyoffl <= 0;
            else wyoffl <= wyoffl + sizex;
        end
    if (wstart==1 && outvalid==1 && windex==63 && wcount==blocksm1)
        begin
            if (ssmaxx == 0) wxoffhtemp = woffh + 1;
            else if (ssmaxx == 1) wxoffhtemp = woffh + 2;
            else wxoffhtemp = woffh + 4;

            if (wxoffhtemp>=sizex)
                begin
                    woffh <= 0;
                    if (ssmaxy == 0) wbase <= wbase + {sizex, 3'b000 };
                    else if (ssmaxy == 1) wbase <= wbase + {sizex, 4'b0000 };
                    else wbase <= wbase + {sizex, 5'b00000};
                end
            else
                begin
                    woffh <= wxoffhtemp;
                end
        end
    end

    woffl <= wix;

    wyoffh <= { sizex * wi, 3'b000};
end

assign finaladdr = wbase + wyoffh + wyoffl + woffh + woffl;

integer z;
```
always @(posedge CK) begin
  proccolour <= colourgo;
end

integer m;

always @(outindex, outblk, outvalid, bankcount[0], bankcount[1], bankcount[2],
        bankcount[3]) begin

  working2 = outblk;

  ix = working2 & ssmaxxmask;
  iy = working2 >> ssmaxx;

  for (m=0;m<4;m=m+1) begin
    ia = (ix<<2)>>subscalex[m];
    ib = (iy<<2)>>subscaley[m];
    iz = (ib>>2)<<subsampx[m] | (ia>>2);
    iw = bankcount[m] - subsize[m] + iz;
    bankraddr[m*10+:10] = { iw, outindex[0+:6] };
    if (subscalex[m]==1) bankraddr[m*10+3+:3] = { ia[1:1], outindex[2:1] };
    if (subscalex[m]==2) bankraddr[m*10+3+:3] = { ia[1:0], outindex[2:2] };
    if (subscaley[m]==1) bankraddr[m*10+3+:3] = { ib[1:1], outindex[5:4] };
    if (subscaley[m]==2) bankraddr[m*10+3+:3] = { ib[1:0], outindex[5:5] };
  end

  if (working2<mcusize) begin
    colourgo=outvalid & outcolourgo; // process if this is new data
  end
  else colourgo=0;

#ifdef DEBUGDECODE
  if (colourgo==1) $display(" CHECK %b ( %10b %10b %10b %10b )",working2,
           bankraddr[0*10+:10],bankraddr[1*10+:10],bankraddr[2*10+:10],bankraddr[3*10+:10]);
#endif

end

#ifdef DEBUGDECODE
always @(posedge CK )
end
begin
  if (combinedvalid) $display("OUT: %06x %1d %1d %1d", combined,combined[16+:8],combined[8+:8],combined[0+:8]);
end
‘endif

always @( posedge CK )
begin
  nextoutvalid <= outvalid;
end

reg [7:0] synccount;
assign finalvalid = combinedvalid;
assign finalout = combined;

always @( posedge CK )
begin
  finallast <= outlast | errorheader | errorhuff ;
end

endmodule

decode.v

A.3 blocker.v

`define DEBUGBLOCKER
module blocker10(CK,startin,startout,inp,tvalid,tready,shift ,
  shiftvalid , shiftready , outp,outpvalid,burst16);
input CK;
input startin;
input [31:0] inp;
input [2:0] shift ;
output [31:0] outp;
output startout;
reg startout;
input tvalid;
output tready;
reg tready;
input shiftvalid ;
output shiftready;
reg shiftready;
output outpvalid;
reg outpvalid;
output burst16;
reg burst16;

reg [63:0] top,temp;
reg [3:0] accum;

reg [4:0] rp,wp,rp,wpt;
reg [4:0] check16;

wire [31:0] inpz;

assign outp = top[63:32];

reg startin3;
reg startin2;

dparam #(5,32) TF ( CK, (tvalid && tready), wp, inp, rp, inpz );

always @( posedge CK )
begin

if ( startin )
begin
    startin3 <= 1;
    wp <= 0;
    tready <= 0;
    startin2 <= 1;
    burst16 <= 1;
end
else
begin

wpt = wp + 1'b1;

check16=wpt-rp;
burst16 <= (check16<15);

if ( tvalid && tready)
begin
 ifdef DEBUGBLOCKER
 $display("BLOCKERFIFO: W[%1d] = %08x %b",wp,inp,tready);

end
end

end
A. VERILOG CODE

`endif

wp <= wp;
if (wpt == rp) tready <= 1'b0; // full
else tready <= 1'b1; // not full

startin2 <= 0;

else if (startin2 == 1 && wp == rp) // only for reset
begin
  tready <= 1'b1; // not full on reset
end
else if (wp != rp)
begin
  tready <= 1'b1; // not full
end
// no else, keep past full status

startin3 <= startin2;

end

always @(posedge CK)
begin

  rpt = rp + 1'b1;

  if (startin)
begin
    startout <= 1;
    rp <= 0;
    outpvalid <= 0;
    shiftready <= 0;
  end
else
begin

  if (startin2)
begin
    accum = 0;
    outpvalid <= 0;
    shiftready <= 1;
    top = 0;
  end
else
begin
    if ( shiftready == 0 )
    begin
        ifdef DEBUGBLOCKER
            $display("BLOCKERSTALL!");
        endif
    end

    if ( shiftready && shiftvalid )
    begin
        accum = accum + shift;
        case ( accum )
            4: temp = { 32'h00000000, inpz },
            5: temp = { 24'h000000, inpz, 8'h00 },
            6: temp = { 16'h0000, inpz, 16'h0000 },
            7: temp = { 8'h00, inpz, 24'h000000 },
            8: temp = { inpz, 32'h00000000 },
            default: temp = 0;
        endcase
        case ( shift )
            0: top = { top },
            1: top = { top[0+:64−(1*8)], 8'h00 },
            2: top = { top[0+:64−(2*8)], 16'h0000 },
            3: top = { top[0+:64−(3*8)], 24'h000000 },
            4: top = { top[0+:64−(4*8)], 32'h00000000 },
            default: top = 0;
        endcase

        if ( accum>=4)
        begin
            ifdef DEBUGBLOCKER
                $display("BLOCKERFIFO: R[%1d] = %08x %b",rp,inpz,startin3);
            endif
            top = top | temp;
            accum = accum - 4'd4;
            rp <= rpt;
            if ( rpt == wp ) shiftready <= 0;
            else shiftready <= 1;
        end
    else
        shiftready <= 1;
    end
outpvalid <= 1;

`ifdef DEBUGBLOCKER
  $display("BLOCKER: %016x %1d %1d",top,accum,shift);
`endif

else if ( rp != wp )
  begin
    outpvalid <= 0;
    shiftready <= 1;
  end

end

startout <= startin3;

end

endmodule

blocker.v

A.4 header.v

// 'define DEBUGHEADER
// 'define STALLHEADER
module header10d(CK,huffparams,imageparams,startin,startout,inp,inpvalid,shift,
    shiftvalid,shiftready ,out,outvalid,outready,progmode,progaddr,progdata,error);
  input CK;
  output [4+4+26+16+10+2*10+2*10−1:0] huffparams;
  output [14+2+16+2+3+5+2+2+3+2*4+2*4+2*4+2*4+4+4−1:0] imageparams;
  input startin;
  output startout;
  reg startout;
  input [31:0] inp;
  input inpvalid;
A. VERILOG CODE

output [2:0] shift ;
reg [2:0] shift ;
input shiftready;
output shiftvalid;
reg shiftvalid ;
output [31:0] out;
reg [31:0] out;
output outvalid;
reg outvalid;
input outready;
output [2:0] progmode;
reg [2:0] progmode;
output [9:0] progaddr;
reg [9:0] progaddr;
output [15:0] progdata;
reg [15:0] progdata;
output error;
reg error;

ifdef STALLHEADER
reg [4:0] stalldelay ;  // for testing stalls to huff.v
endif

// progmode: 0=none, 1=base, 2=enable, 3=offset, 4=code, 5=quant
parameter pnone = 0, pbase = 1, penable = 2, poffset = 3, pcode = 4, pquant = 5;

// huffparams
reg [3:0] blkmax;
reg [2:0] compmax;
reg blkindex[0:9];
reg [1:0] blkquant[0:9];
reg [1:0] blkcomp[0:9];
reg [4:0] mcusize;
reg [15:0] restart;
reg [1:0] ssmaxx;
reg [2:0] ssmaxxmask;
reg [1:0] subscalex [0:3];
reg [1:0] subscaley [0:3];
reg [1:0] subsampx[0:3];
reg [3:0] subsize [0:3];
reg [25:0] totalmcu;
reg [3:0] blocksm1;

assign huffparams = { blkmax, blocksm1, totalmcu, restart, blkindex[0], blkindex[1], blkindex [2], blkindex [3], blkindex [4], blkindex [5], blkindex [6], blkindex [7],...
A. VERILOG CODE

blkindex [8], blkindex [9], blkquant [0], blkquant [1], blkquant [2], blkquant [3], blkquant [4], blkquant [5], blkquant [6], blkquant [7], blkquant [8], blkquant [9], blkcomp[0], blkcomp[1], blkcomp[2], blkcomp[3], blkcomp[4], blkcomp[5], blkcomp[6], blkcomp[7], blkcomp[8], blkcomp[9] ;

// image parameters
reg [14-1:0] sizex, sizey ; // physical memory size /8 in both directions
reg [16-1:0] dispx, dispy;
reg [1:0] ssmaxy;
assign imageparams = { sizex, sizey, dispx, dispy, compmax, mcusize, ssmaxx, ssmaxy, ssmmaxxmask, subscalex[0], subscaley[1], subscaley[2], subscaley[3], subscaley[0], subscaley[1], subscaley[2], subscaley[3], subsampx[0], subsampx[1], subsampx[2], subsampx[3], subsize[0], subsize[1], subsize[2], subsize[3] };

// for now
reg [15:0] b [1:16];

// permanent
reg [1:0] subsampy[0:3];
reg [4:0] state, nextstate;
reg [15:0] skipbytes;
reg [1:0] number;
reg [7:0] index, index2, index3;
reg [7:0] huff [0:15];
done;
reg [15:0] base;
reg [3:0] ct [0:3]; // same as subsize
reg [1:0] cq [0:3];
reg [15:0] enable;
reg newval;

// should not become registers
reg [2:0] skip;
reg [7:0] work1;
reg [3:0] x,y;
reg throw;
reg nextshiftvalid ;
reg [15:0] w0, w1, w01;
reg [7:0] b0, b1, b2, b3;

reg found0xd8, found0xc0, found0xc4, found0xdb;

// 0
parameter sidle = 0, sgetword = sidle + 1, sskipchunks = sgetword + 1;
// 3
parameter sgetquant1 = sskipchunks + 1, sgetquant2 = sgetquant1 + 1;
// 5
parameter sgethuff1 = sgetquant2 + 1, sgethuff2 = sgethuff1 + 1, sgethuff3 = sgethuff2 + 1, sgethuff4 = sgethuff3 + 1, sgethuff5 = sgethuff4 + 1, sgethuff6 = sgethuff5 + 1;
// 11
parameter sgetsof1 = sgethuff6 + 1, sgetsof2 = sgetsof1 + 1, sgetsof3 = sgetsof2 + 1, sgetsof4 = sgetsof3 + 1, sgetsof5 = sgetsof4 + 1, sgetsof6 = sgetsof5 + 1, sgetsof7 = sgetsof6 + 1, sgetsof8 = sgetsof7 + 1;

// 17
parameter sgetsos1 = sgetsof8 + 1, sgetsos2 = sgetsos1 + 1, sgetsos3 = sgetsos2 + 1, sgetsos4 = sgetsos3 + 1, sgetsos5 = sgetsos4 + 1, sgetsos6 = sgetsos5 + 1;
// 23
parameter sgetdri = sgetsos6 + 1;
// 24
parameter serror = sgetdri + 1;
// 25
parameter srelay = serror + 1;

integer i;

always @(posedge CK)
begin
  'ifdef DEBUGHEADER
  $display("HEADER: startin=%1b shiftready=%1b shiftvalid=%1b inpvalid=%1b newval=%1b", startin, shiftready, shiftvalid, inpvalid, newval);
  'endif

  if ( startin == 1 )
  begin
    state <= sidle;
    found0xd8 <= 0;
    found0xc0 <= 0;
    found0xc4 <= 0;
    found0xdb <= 0;
    error <= 0;
    done <= 0;
    newval = 1;
    shift <= 0;
    shiftvalid <= 0;
  end
end
startout <= 1;
outvalid <= 0;
sizex <= 0;
sizey <= 0;
dispx <= 0;
dispy <= 0;
ifdef STALLHEADER
stalldelay <= -1;
endif

else
begin
skip = 0;
if ( error ) $finish;
if ( !newval ) newval = inpvalid;
if ( newval )
begin

w0 = inp[16+:16];
w1 = inp[0+:16];
w01 = inp[8+:16];
b0 = inp[24+:8];
b1 = inp[16+:8];
b2 = inp[8+:8];
b3 = inp[0+:8];

ifdef DEBUGHEADER
$display("HEADER2: inp=%08h inpvalid=%b newval=%b state=%1d",inp,inpvalid,newval,state);
endif

if ( done )
begi
$display("PARAMS");
$display("blkmax = %1d",blkmax);
$display("blocksm1 = %1d",blocksm1);
$display("compmax = %1d",compmax);

endif
$display("mcusize = %1d;",mcusize);
$display("totalmcu = %1d;",totalmcu);
$display("restart = %1d;",restart);
for( i = 0 ; i < 10 ; i = i + 1 ) $display("blkindex[%1d] = %1d;",i,blkindex[i]);
for( i = 0 ; i < 10 ; i = i + 1 ) $display("blkquant[%1d] = %1d;",i,blkquant[i]);
for( i = 0 ; i < 10 ; i = i + 1 ) $display("blkcomp[%1d] = %1d;",i,blkcomp[i]);
$display("ssmaxx = %1d;",ssmaxx);
$display("ssmaxy = %1d;",ssmaxy);
$display("ssmaxxmask = %1d","ssmaxxmask);
for( i = 0 ; i < 4 ; i = i + 1 ) $display("subscalex[%1d] = %1d;",i,subscalex[i]);
for( i = 0 ; i < 4 ; i = i + 1 ) $display("subscaley[%1d] = %1d;",i,subscaley[i]);
for( i = 0 ; i < 4 ; i = i + 1 ) $display("subsampx[%1d] = %1d;",i,subsampx[i]);
for( i = 0 ; i < 4 ; i = i + 1 ) $display("subsize[%1d] = %1d;",i,subsize[i]);
end

case ( state )
sidle :
    begin
        progmode <= pnone;
        if ( b0 == 255 )
            begin
                ifdef DEBUGHEADER
                    $display("HEADER3: Marker=%02x",b1);
                endif
                casex ( b1 )
                    8’hc0:
                        begin
                            found0xc0 <= 1;
                            state <= sgetword;
                            nextstate <= sgetsof1;
                            skip = 2;
                        end
                    8’hc4:
                        begin
                            found0xc4 <= 1;
                            state <= sgetword;
                            nextstate <= sgethuff1;
                            skip = 2;
                        end
                    8’hd8:
                        begin
                            found0xd8 <= 1;
                            restart <= 0;
                            skip = 2;
                        end
A. VERILOG CODE

```verilog
end

8'hda:
begin
if ( found0xd8 && found0xc0 && found0xc4 && found0xdb )
begin
    state <= sgetword;
    nextstate <= sgetsos1;
    skip = 2;
end
else
begin
    state <= serror;
end
end

8'hdb:
begin
    found0xdb <= 1;
    state <= sgetword;
    nextstate <= sgetquant1;
    skip = 2;
end

8'hdd:
begin
    state <= sgetword;
    nextstate <= sgetdri;
    skip = 2;
end

8'hex, 8'hfx: // stuff we don’t need, but can skip
begin
    state <= sgetword;
    nextstate <= sskipchunks;
    skip = 2;
end

default:
begin
    state <= serror;
end
endcase
end
else
begin
if ( found0xd8) // if junk data in image, generate error
begin
    state <= serror;
end
end
```
else
begin
    skip = 1;
end
end

sgetword:
begin
    skipbytes <= w0 – 2’d2;
    state <= nextstate;
    skip = 2;
end

sskipchunks:
begin
    if ( skipbytes )
        begin
            ifdef DEBUGHEADER
                $display("HEADER3: SKIPPING %1d",skipbytes);
            endif
            if ( skipbytes > 3 ) skip = 4;
            else if ( skipbytes > 2 ) skip = 3;
            else if ( skipbytes > 1 ) skip = 2;
            else skip = 1;
            skipbytes <= skipbytes – skip;
        end
    else
        begin
            state <= sidle;
            skip = 0;
        end
end

sgetquant1:
begin
    number <= b0[0+:2];
    index <= 0;
    skip = 1;
    skipbytes <= skipbytes – 1’b1;
    if ( b0[4+:4] != 0 || b0[2+:2] != 0 ) // only 8 bit tables supported, only tables 0–3 supported
        begin
            error <= 1;
            state <= sidle;
        end
end
A. VERILOG CODE

```verilog
end
else
begin
  state <= sgetquant2;
end
end

sgetquant2:
begin
  if ( index == 63 )
  begin
    if ( skipbytes > 63 ) state <= sgetquant1;
    else state <= sidle;
  end
  progmode <= pquant;
  progaddr <= (number<<6)|index; // 8 bits
  progdata <= b0;
  index <= index + 1'b1;
  skip = 1;
  skipbytes <= skipbytes - 1'b1;
end

sgethuff1:
begin
  progmode <= pnone;
  if ( skipbytes > 15 )
  begin
    if ( b0[5+:3] != 0 || b0[1+:3] != 0 ) // only destinations 0-1 supported,
      only classes 0-1 supported
    begin
      error <= 1;
      state <= sidle;
    end
  else
  begin
    number[0+:2] <= { b0[0+:1], b0[4+:1]};
    index <= 0;
    state <= sgethuff2;
  end
  skip = 1;
  skipbytes <= skipbytes - 1'b1;
end
else state <= sidle;
end
```
sgethuff2:
  begin
  index3 <= 0;
  base <= 0;
  if ( index == 15 )
  begin
    state <= sgethuff3;
    index <= 0;
  end
  else
  begin
    index <= index + 1'b1;
  end
  huff[index] <= b0;
  enable[index] <= ( b0 != 0 );
  skip = 1;
  skipbytes <= skipbytes - 1'b1;
end

sgethuff3:
  begin
    b[index+1] <= base; // for testing
    progmode <= pbase;
    progaddr <= {number[0+:2],index[0+:4]}; // 6 bits
    progdata <= base;
    base <= (base<<1);
    state <= sgethuff4;
  end

sgethuff4:
  begin
    progmode <= poffset;
    progaddr <= {number[0+:2],index[0+:4]}; // 6 bits
    progdata <= index3;
    index2 <= huff[index];
    if ( enable[index] ) state <= sgethuff5;
  else
  begin
    if ( index == 15 ) state <= sgethuff6;
  end
  end
A. VERILOG CODE

end

sgethuff5:
  begin
    progmode <= pcode;
    progaddr <= {number[0+:2],index3[0+:8]}; // 10 bits
    progdta <= b0;
    base <= base + 1’b1;
    index3 <= index3 + 1’b1;
    index2 <= index2 – 1’b1;
    skip = 1;
    skipbytes <= skipbytes – 1’b1;
    if ( index2 == 1 )
      begin
        index <= index + 1’b1;
        if ( index == 15 ) state <= sgethuff6;
        else state <= sgethuff3;
      end
  end

sgethuff6:
  begin
    progmode <= penable;
    progaddr <= number[0+:2]; // 2 bits
    progdta <= enable;
    state <= sgethuff1;
  end

sgetsof1 : // Check if baseline, and 8 bit precision, get display—y
  begin
    if ( b0 != 8 )
      begin
        error <= 1; // support only baseline, 8 bit precision
        state <= sidle;
      end
    else
      begin
        skip = 3;
        skipbytes <= skipbytes – 2’d3;
        state <= sgetsof4;
      end
dispy <= w01;
  end

sgetsof4 : // get display—x, number of components
begin
    dispx <= w0;
    if ( b2 > 4 )
        begin
            error <= 1; // support only upto 4 components
            state <= sidle;
        end
    else
        begin
            index <= 0;
            ssmaxy <= 0;
            ssmaxx <= 0;
            blkmax <= 0;
            compmax <= b2[0+:3];
            skip = 4; // should be 3, but we are skipping the first table id, we will assume they are in the same order in the SOF and SOS
            skipbytes <= skipbytes − 3’d4;
            state <= sgetsof5;
        end
    end
end

sgetsof5 : // read in subsamping of each component and huffman/quantization tables -- skip table IDs -- assume they are in order
begin
    throw = 0;
    x = b0[4+:3];
    y = b0[0+:3];
    work1 = x * y;
    subsize[index] <= work1[0+:4];
    blkmax <= blkmax + work1[0+:4];
    ct[index] <= work1[0+:4];
    cq[index] <= b1[0+:2];
    if ( b1[2+:6] ) throw = 1;

    if ( x == 1 ) x = 0; else if ( x == 2 ) x = 1; else if ( x == 4 ) x = 2;
        else throw = 1;
    if ( y == 1 ) y = 0; else if ( y == 2 ) y = 1; else if ( y == 4 ) y = 2;
        else throw = 1;

    if ( y > ssmaxy ) ssmaxy <= y;
    if ( x > ssmaxx ) ssmaxx <= x;
    subsampx[index] <= x;
    subsampy[index] <= y;

    if ( throw )
begin
  error <= 1; // incorrect subsampling or quant table range wrong
  state <= sidle;
end
else
begin
  if ( index == compmax - 1'b1 )
  begin
    index <= 0;
    state <= sgetsof6;
    skip = 2;
    skipbytes <= skipbytes - 2'd2;
  end
else
begin
  index <= index + 1'b1;
  skip = 3; // set to skip next table id; see above
  skipbytes <= skipbytes - 2'd3;
end
end
end

sgetso6: // adjust subsampling values
begin
  skip = 0;
  subscalex[index] <= ssmaxx - subsampx[index];
  subscaley[index] <= ssmaxy - subsampy[index];
  if ( index == compmax - 1'b1 )
  begin
    state <= sgetsof7;
  end
else
begin
  index <= index + 1'b1;
end
end

sgetso7: // generate proper memory dimensions for x and y
begin
  if ( ssmaxx == 0 & & disp[0+:3] != 0 ) sizex <= disp[3+:13] + 1'b1;
elself if ( ssmaxx == 1 & & disp[0+:4] != 0 ) sizex <= { disp[4+:12] + 1'b1, 1'b0 }
else if ( ssmaxx == 2 & & disp[0+:5] != 0 ) sizex <= { disp[5+:11] + 1'b1,
  2'b00 };
elself sizex <= disp[3+:13];
A. VERILOG CODE

if (ssmaxy==0 && dispy[0+:3]!=0) sizey <= dispy[3+:13]+1’b1;
else if (ssmaxy==1 && dispy[0+:4]!=0) sizey <= { dispy[4+:12]+1’b1, 1’b0 }
else if (ssmaxy==2 && dispy[0+:5]!=0) sizey <= { dispy[5+:11]+1’b1,
2’b00 }
else sizey <= dispy[3+:13];

work1 = ssmaxx + ssmaxy;
if ( work1[0+:3] == 0 ) mcusize <= 1;
else if ( work1[0+:3] == 1 ) mcusize <= 2;
else if ( work1[0+:3] == 2 ) mcusize <= 4;
else if ( work1[0+:3] == 3 ) mcusize <= 8;
else mcusize <= 16;

state <= sgetsof8;
end

getsof8: // calculate total MCUs based on size of x and y from above
begin

totalmcu <= sizex * sizey; // need to still divid by mcusize!!!
// determine the number of blocks which need to be processed for each mcu
if (blkmax<mcusize) blocksm1 <= mcusize - 1;
else blocksm1 <= blkmax - 1;
state <= sidle;
end

getsos1:
begin
if ( b0[0+:3] != compmax || b0[3+:5] != 0 )
begin
error <= 1; // support only upto 4 components or those specified in the SOF
state <= sidle;
end
else
begin
index <= 0;
index2 <= 0;
index3 <= 0;
skip = 1;
skipbytes <= skipbytes - 1’b1;
state <= sgetsos2;
end
end
sgetsos2:
begin
throw = 0; // skip table id; see above
if ( work1[1+:3] != 0 || work1[5+:3] != 0 || work1[0+:1] != work1[4+:1] )
throw = 1;
number[0] <= work1[0+:1];
index2 <= ct[index];

if ( throw )
begin
error <= 1; // unsupported table reference
state <= sidle;
end
else
begin
state <= sgetsos3;
skip = 2;
skipbytes <= skipbytes - 2'd2;
end
end

sgetsos3:
begin
blkindex[index3] <= number[0];
blkquant[index3] <= cq[index];
blkcomp[index3] <= index[0+:2];
index3 <= index3 + 1'b1;

if ( index2 == 1 )
begin
if ( index == compmax - 1'b1 )
begin
state <= sgetsos4;
end
else
begin
index <= index + 1'b1;
state <= sgetsos2;
end
end
else
begin
index2 <= index2 - 1'b1;
end
sgetsos4:
begin
if ( b0 != 0 & & b1 != 63 )
begin
error <= 1; // unsupported spectral selection
state <= sidle;
end
else
begin
state <= sgetsos5;
skip = 2;
nbytes <= skipbytes - 2'd2;
end
end

sgetsos5:
begin
if ( ssmaxx==0) ssmaxxmask <= 3'b000;
else if ( ssmaxx==1) ssmaxxmask <= 3'b001;
else ssmaxxmask <= 3'b011;
index <= 25;
index2 <= 0;

// need to get totalmcu/mcusize

if ( b0 != 0 )
begin
error <= 1; // unsupported successive approximation
state <= sidle;
end
else
begin
state <= sgetsos6;
skip = 1;
nbytes <= skipbytes - 1'b1;
end
end

// divide mcutotal by mcusize, result in mcutotal
sgetsos6:
begin
work1 = { index2[0:+7], totalmcu[index] };
if ( work1 >= mcusize )
begin
  work1 = work1 - mcusize;
  totalmcu[index] <= 1;
end
else
begin
  totalmcu[index] <= 0;
end
index2 <= work1;
if ( index == 0 )
begin
  state <= srelay;
  done <= 1;
end
else
begin
  index <= index - 1'b1;
end
end
sgetdri :
begin
  restart <= w0;
  skip = 2;
  skipbytes <= skipbytes - 2'd2;
  state <= sidle;
end
serror :
begin
  progmode <= pnone;
  error <= 1;
  state <= sidle;
end
srelay :
begin
  $display("HEADEROUT: %08x",inp);
  // place new value on streamer bus
  out <= inp;
  outvalid <= 1;
  newval = 0;
  done <= 0;
  startout <= 0;
A. VERILOG CODE

end

default:
    begin
        progmode <= pnone;
        state <= sidle;
    end

endcase

shift <= skip;
if ( skip )
begin
    shiftvalid <= 1;
    newval = 0;
end
else
begin
    shiftvalid <= 0;
end

end // newval
else
begin
    // when streamer gets value, have blocker shift 4 more bytes
    if ( outready && outvalid )
begin
        outvalid <= 0;
        ifdef STALLHEADER
        stalldelay <= -1;
        if ( shiftready && shiftvalid ) shiftvalid <= 0;
    end
    else if ( stalldelay == 0 )
begin
        stalldelay <= -1;
 wherein
    shift <= 4;
    shiftvalid <= 1;
    end
    else if ( shiftready && shiftvalid ) shiftvalid <= 0;
    ifdef STALLHEADER
    else if ( state == srelay && outvalid==0) stalldelay <= stalldelay - 1;
    endif
    end
}
end
end
endmodule

A.5 stream.v

module
  stream10c(CK,startin,inp,shift1,shift2,startout,outp,tvalid,tready,remain,new,stall);
  input CK;
  input startin;
  input [31:0] inp;
  input [4:0] shift1,shift2;
  output [31:0] outp;
  output startout;
  reg startout;
  input tvalid;
  output tready;
  reg tready;
  output [2:0] remain;
  reg [2:0] remain;
  output new;
  reg new;
  output stall;
  reg stall;

  reg [63:0] top,temp;
  reg [6:0] accum;
  reg lastmarker;
  reg lastmarkert;

  reg [4:0] check3,check2,check1; // I have to do this.

  reg [31:0] inp2,inp3;
  wire [31:0] inpz;
  reg [1:0] add2,add3;
  wire [1:0] addz;

  assign outp = top[63:32];
```verilog
reg startin3;
reg startin2;
reg [4:0] shift;

reg [0:0] rp,wp,rpt,wpt;
reg [33:0] fifo [0:1];

assign {inpz,addz} = fifo[rp];
always @(posedge CK)
begin
    if (tvalid && tready) fifo[wp] = {inp2,add2};
end
always @(inp or lastmarker)
begin
    inp2 = inp;
    add2 = 0;

    if (lastmarker && inp2[3*8+:8] == 0)
        begin
            inp2 = {inp2[23:0],8'b00000000};
            add2 = 1;
        end
    if (inp2[3*8+:8] == 255 && inp2[2*8+:8] == 0)
        begin
            inp2 = {inp2[31:24],inp2[15:0],8'b00000000};
            add2 = add2 + 1'b1;
        end
    if (add2 < 2 && inp2[2*8+:8] == 255 && inp2[1*8+:8] == 0)
        begin
            inp2 = {inp2[31:16],inp2[7:0],8'b00000000};
            add2 = add2 + 1'b1;
        end
    if (add2 == 0 && inp2[1*8+:8] == 255 && inp2[0*8+:8] == 0)
        begin
            inp2 = {inp2[31:8],8'b00000000};
            add2 = add2 + 1'b1;
        end
    lastmarkert = (inp[add2<<3+:8] == 255);
```
A. VERILOG CODE

```verilog
end

always @(posedge CK)
begin

wpt = wp + 1'b1;
if (startin)
begin
    startin3 <= 1;
    wp <= 0;
    lastmarker <= 0;
    tready <= 0;
    startin2 <= 1;
end
else
begin
    if (tvalid && tready)
    begin
        $display("STREAMFIFO: W[%1d] = %b %d %b", wp, inp2, add2, tready);
        lastmarker <= lastmarkert;
        wp <= wpt;
        if (wpt == rp) tready <= 1'b0; // full
    end
    else if (startin2 == 1 && wp == rp) // only for reset
    begin
        tready <= 1'b1; // not full on reset
    end
    else if (wp != rp)
    begin
        tready <= 1'b1; // not full
    end
    // no else, keep past full status
    startin3 <= startin2;
end

end

always @(posedge CK)
begin

```

---

102
A. VERILOG CODE

rpt = rp + 1'b1;

if (startin)
begin
    startout <= 1;
    rp <= 0;
    new <= 0;
    remain <= 0;
    stall <= 0;
end
else
begin
    shift = shift1 + shift2;

    if (startin3 && startin2==0)
begin
        $display("STREAMFIFO: R[%1d] = %b %d %b",rp,inpz,addz,startin3);

        accum= 7'd32 | (addz <= 3);
        top = { inpz , {32{1'b0}}};
        new <= 1;
        remain <= 0;
        rp <= rpt;
        if (rpt == wp) stall <= 1;
        else stall <= 0;
end
else if (startin3)
begin
    rp <= 0;
    new <= 0;
    remain <= 0;
    stall <= 0;
end
else
begin
    if ( stall ==1)
begin
        $display("STREAMSTALL!");
end

    if ( shift !=0 && stall==0)
begin

end
A. VERILOG CODE

accum = accum + shift;
temp = inpz << (accum & 5'd31);
top = top << (shift & 5'd31);
if (accum >= 32)
begin
    $display("STREAMFIFO: R[%1d] = %b %d %b", rp, inpz, addz, startin3);
top = top | temp;
accum = accum - 6’d32;
accum = accum + (addz << 3);
rp <= rpt;
if (rpt == wp) stall <= 1;
else
    begin
        if (accum >= 33) // if not enough data, stall again
            begin
                $display("BADSTUFF!");
stall <= 1;
            end
        else
            stall <= 0;
    end
else
    begin
        stall <= 0;
    end
end
end
else if (rp != wp)
begin
    if (accum >= 33) // fix for low data on output
        begin
            $display("STREAMFIFO: R[%1d] = %b %d %b", rp, inpz, addz, startin3);
temp = inpz << (accum & 5’d31);
top = top | temp;
accum = accum - 6’d32;
accum = accum + (addz << 3);
rp <= rpt;
if (rpt == wp) stall <= 1;
else
    stall <= 0;
new <= 1;
end
else
begin
A. VERILOG CODE

```verilog
stall <= 0;
new <= 0;
end
end
else
begin
new <= 0;
end

remain <= -accum[2:0];

end

startout <= startin3;
end
end
endmodule

stream.v

A.6 huff.v

`define SYNTHESIS

`define PREDICTOR

module huff10d(  
    CK,huffparams,progmode,progaddr,progdata,startin,inp,outshift1,outshift2,
    outscaled,outindex,outaddr,outblk,outbank,outbankenable,outbankslast,outcolourgo,
    outvalid,outlast,remain,new,stall, writefifofull , error);

parameter imax=31;
input CK;
input startin;
input [2:0] progmode;
input [9:0] progaddr;
input [15:0] progdata;
input [imax:0] inp;
output [4:0] outshift1, outshift2;
reg [4:0] outshift1, outshift2;
output signed [15:0] outscaled;
```
output [6:0] outindex;
output [6:0] outaddr;
output [3:0] outblk;
reg [3:0] outblk;
output [3:0] outbank;
reg [3:0] outbank;
output [3:0] outbankenable;
reg [3:0] outbankenable;
output [16−1:0] outbankslast;
reg [16−1:0] outbankslast;
output outcolourgo;
reg outcolourgo;
output outvalid;
reg outvalid;
output outlast;
reg outlast;
input [2:0] remain;
input new;
input stall;
input writefifofull;
input [4+4+26+10+2*10+2*10−1:0] huffparams;
output error;
reg error;
reg new2;
reg pastnew2;
reg [1:0] decodestall;
reg advance;
reg flush;
reg foundmarker;
reg pastfoundmarker;
reg foundmarkert;
reg [2:0] marker,lastmarker;
reg [4:0] outshiftt;
reg [31:0] countercheck;
reg [31:0] pixels,clocks;
reg [25:0] mcucount;
reg [3:0] bankcount[0:3];
wire [5:0] zz [0:63];
reg [0:0] b02;
reg [1:0] b03,d01;
A. VERILOG CODE

reg [2:0] b04,d02;
reg [3:0] b05,d03;
reg [4:0] b06,d04;
reg [5:0] b07,d05;
reg [6:0] b08,d06;
reg [7:0] b09,d07;
reg [8:0] b10,d08;
reg [9:0] b11,d09;
reg [10:0] b12,d10;
reg [11:0] b13,d11;
reg [12:0] b14,d12;
reg [13:0] b15,d13;
reg [14:0] b16,d14;
reg [15:0] c16,d15;
reg [16:0] d16;
reg [16:1] c;

reg [1:0] index;
reg [3:0] blkcount,nextblkcount;
reg nextstartin,nextstartin2;
reg nextvalid;
reg nextlast;
wire [3:0] blkmax;
wire [15:0] restart;
wire blkindex[0:9];
wire [1:0] blkquant[0:9];
wire [1:0] blkcomp[0:9];
reg [1:0] nextblkquant;
wire [25:0] totalmcu;
wire [3:0] blocksm1;

reg [15:0] restartcount;

assign { blkmax, blocksm1, totalmcu, restart, blkindex[0], blkindex[1], blkindex[2], blkindex[3], blkindex[4], blkindex[5], blkindex[6], blkindex[7], blkindex[8], blkindex[9], blkquant[0], blkquant[1], blkquant[2], blkquant[3], blkquant[4], blkquant[5], blkquant[6], blkquant[7], blkquant[8], blkquant[9], blkcomp[0], blkcomp[1], blkcomp[2], blkcomp[3], blkcomp[4], blkcomp[5], blkcomp[6], blkcomp[7], blkcomp[8], blkcomp[9] } = huffparams;

reg signed [15:0] intcode, outcode, lastcode, nextcode, int2code;
reg [6−1:0] coefindex;
reg [4+6−1:0] coefindexfull, nextcoefindexfull;
reg [7:0] quantindex;
A. VERILOG CODE

reg [3:0] codelength;
reg [5:0] runcount;
reg [31:0] codetemp;
reg signed [15:0] codeval;
reg tempsign;
reg [3:0] sz,szb;
reg [7:0] ofs;
reg [3:0] newsz;
reg [7:0] newofs;
reg signed [15:0] dc [0:3];
reg [3:0] dcaccum;
reg [7:0] huffcode;

ifdef PREDICTOR
reg predicton1;
endif

wire [0:0] tb02_out;
wire [1:0] tb03_out;
wire [2:0] tb04_out;
wire [3:0] tb05_out;
wire [4:0] tb06_out;
wire [5:0] tb07_out;
wire [6:0] tb08_out;
wire [7:0] tb09_out;
wire [8:0] tb10_out;
wire [9:0] tb11_out;
wire [10:0] tb12_out;
wire [11:0] tb13_out;
wire [12:0] tb14_out;
wire [13:0] tb15_out;
wire [14:0] tb16_out;
wire [16−1:0] te_out;
wire [8−1:0] to_out;
wire [8−1:0] tc_out;
wire [8−1:0] tq_out;
wire [8−1:0] tc_addr2;
assign tc_addr2 = to_out+ofs;
‘ifdef SYNTHESIS

// progmode: 0=none, 1=base, 2=enable, 3=offset, 4=code, 5=quant
parameter pnone = 0, pbase = 1, penable = 2, poffset = 3, pcode = 4, pquant = 5;

wire [15:1] tb_we;
wire [2:1] tb_addr;
assign tb_addr = (progmode == 1) ? prog_addr[4+:2] : index;
assign tb_we = (progmode == 1) ? 1'b1 : 1'b0;
asyncmem #(2,1) TB02 (CK, tb_we[1], progdata[0+:1], tb_addr, tb02_out);
asyncmem #(2,2) TB03 (CK, tb_we[2], progdata[0+:2], tb_addr, tb03_out);
asyncmem #(2,3) TB04 (CK, tb_we[3], progdata[0+:3], tb_addr, tb04_out);
asyncmem #(2,4) TB05 (CK, tb_we[4], progdata[0+:4], tb_addr, tb05_out);
asyncmem #(2,5) TB06 (CK, tb_we[5], progdata[0+:5], tb_addr, tb06_out);
asyncmem #(2,6) TB07 (CK, tb_we[6], progdata[0+:6], tb_addr, tb07_out);
asyncmem #(2,7) TB08 (CK, tb_we[7], progdata[0+:7], tb_addr, tb08_out);
asyncmem #(2,8) TB09 (CK, tb_we[8], progdata[0+:8], tb_addr, tb09_out);
asyncmem #(2,9) TB10 (CK, tb_we[9], progdata[0+:9], tb_addr, tb10_out);
asyncmem #(2,10) TB11 (CK, tb_we[10], progdata[0+:10], tb_addr, tb11_out);
asyncmem #(2,11) TB12 (CK, tb_we[11], progdata[0+:11], tb_addr, tb12_out);
asyncmem #(2,12) TB13 (CK, tb_we[12], progdata[0+:12], tb_addr, tb13_out);
asyncmem #(2,13) TB14 (CK, tb_we[13], progdata[0+:13], tb_addr, tb14_out);
asyncmem #(2,14) TB15 (CK, tb_we[14], progdata[0+:14], tb_addr, tb15_out);
asyncmem #(2,15) TB16 (CK, tb_we[15], progdata[0+:15], tb_addr, tb16_out);

wire te_we;
wire [2:1] te_addr;
assign te_addr = (progmode == 2) ? prog_addr[0+:2] : index;
assign te_we = (progmode == 2) ? 1'b1 : 1'b0;
asyncmem #(2,16) TE (CK, te_we, progdata[0+:16], te_addr, te_out);
wire to_we;
wire [6:1:0] to_addr;
assign to_addr = (progmode == 3) ? progaddr[0+:6] : { index, sz };
assign to_we = (progmode == 3) ? 1'b1 : 1'b0;
asyncmem #(6,8) TO (CK, to_we, progdata[0+:8], to_addr, to_out);

wire tc_we;
wire [10−1:0] tc_addr;
assign tc_addr = (progmode == 4) ? progaddr[0+:10] : { index, tc_addr2 };
assign tc_we = (progmode == 4) ? 1'b1 : 1'b0;
asyncmem #(10,8) TC (CK, tc_we, progdata[0+:8], tc_addr, tc_out);

wire tq_we;
wire [8−1:0] tq_addr;
assign tq_addr = (progmode == 5) ? progaddr[0+:8] : quantindex;
assign tq_we = (progmode == 5) ? 1'b1 : 1'b0;
asyncmem #(8,8) TQ (CK, tq_we, progdata[0+:8], tq_addr, tq_out);

initial
begin
  pixels=0;
  clocks = 0;
  countercheck=1;
end

‘else

reg [0:0] TB02[0:3];
reg [1:0] TB03[0:3];
reg [2:0] TB04[0:3];
reg [3:0] TB05[0:3];
reg [4:0] TB06[0:3];
reg [5:0] TB07[0:3];
reg [6:0] TB08[0:3];
reg [7:0] TB09[0:3];
reg [8:0] TB10[0:3];
reg [9:0] TB11[0:3];
reg [10:0] TB12[0:3];
reg [11:0] TB13[0:3];
reg [12:0] TB14[0:3];
reg [13:0] TB15[0:3];
reg [14:0] TB16[0:3];
reg [15:0] TE[0:3];
reg [7:0] TO[0:63];
reg [7:0] TC[0:1023];
A. VERILOG CODE

reg [7:0] TQ[0:255];
assign tb02_out = TB02[index];
assign tb03_out = TB03[index];
assign tb04_out = TB04[index];
assign tb05_out = TB05[index];
assign tb06_out = TB06[index];
assign tb07_out = TB07[index];
assign tb08_out = TB08[index];
assign tb09_out = TB09[index];
assign tb10_out = TB10[index];
assign tb11_out = TB11[index];
assign tb12_out = TB12[index];
assign tb13_out = TB13[index];
assign tb14_out = TB14[index];
assign tb15_out = TB15[index];
assign tb16_out = TB16[index];
assign te_out = TE[index];
assign to_out = TO[{index,sz}];
assign tc_out = TC[{index,tc_addr2}];
assign tq_out = TQ[quantindex];

initial
begin
  pixels = 0;
  clocks = 0;
  countercheck = 1;
end

// progmode: 0=none, 1=base, 2=enable, 3=offset, 4=code, 5=quant
parameter pnone = 0, pbase = 1, penable = 2, poffset = 3, pcode = 4, pquant = 5;
reg [1:0] progttemp;
always @(posedge CK)
begin
  if (progmode == 1)
  begin
    progttemp = progaddr[4+:2];
    case (progaddr[0+:4])
      1: TB02[progttemp] <= progdata[0+:1];
      2: TB03[progttemp] <= progdata[0+:2];
      3: TB04[progttemp] <= progdata[0+:3];
      4: TB05[progttemp] <= progdata[0+:4];
    endcase
  end
end
A. VERILOG CODE

5: TB06[progtemp] <= progsdata[0:+5];
6: TB07[progtemp] <= progsdata[0:+6];
7: TB08[progtemp] <= progsdata[0:+7];
8: TB09[progtemp] <= progsdata[0:+8];
9: TB10[progtemp] <= progsdata[0:+9];
10: TB11[progtemp] <= progsdata[0:+10];
11: TB12[progtemp] <= progsdata[0:+11];
12: TB13[progtemp] <= progsdata[0:+12];
13: TB14[progtemp] <= progsdata[0:+13];
14: TB15[progtemp] <= progsdata[0:+14];
15: TB16[progtemp] <= progsdata[0:+15];
endcase
end
if (progmode==2) TE[progaddr[0:+2]] <= progsdata[0:+16];
if (progmode==3) TO[progaddr[0:+6]] <= progsdata[0:+8];
if (progmode==4) TC[progaddr[0:+10]] <= progsdata[0:+8];
if (progmode==5) TQ[progaddr[0:+8]] <= progsdata[0:+8];
end
'endif

always @(posedge CK )
begin

b02 = tb02_out;
b03 = tb03_out;
b04 = tb04_out;
b05 = tb05_out;
b06 = tb06_out;
b07 = tb07_out;
b08 = tb08_out;
b09 = tb09_out;
b10 = tb10_out;
b11 = tb11_out;
b12 = tb12_out;
b13 = tb13_out;
b14 = tb14_out;
b15 = tb15_out;
b16 = tb16_out;

c [16:1] = te_out;
end

always @(inp or c or b02 or b03 or b04 or b05 or b06 or b07 or b08 or b09 or b10 or b11 or b12 or b13 or b14 or b15 or b16 )
begin
A. VERILOG CODE

d01 = { 1'b0, inp[imax] };
d02 = { 1'b0, inp[imax:imax−1] } – { 1'b0, b02, 1'b0 };
d03 = { 1'b0, inp[imax:imax−2] } – { 1'b0, b03, 1'b0 };
d04 = { 1'b0, inp[imax:imax−3] } – { 1'b0, b04, 1'b0 };
d05 = { 1'b0, inp[imax:imax−4] } – { 1'b0, b05, 1'b0 };
d06 = { 1'b0, inp[imax:imax−5] } – { 1'b0, b06, 1'b0 };
d07 = { 1'b0, inp[imax:imax−6] } – { 1'b0, b07, 1'b0 };
d08 = { 1'b0, inp[imax:imax−7] } – { 1'b0, b08, 1'b0 };
d09 = { 1'b0, inp[imax:imax−8] } – { 1'b0, b09, 1'b0 };
d10 = { 1'b0, inp[imax:imax−9] } – { 1'b0, b10, 1'b0 };
d11 = { 1'b0, inp[imax:imax−10] } – { 1'b0, b11, 1'b0 };
d12 = { 1'b0, inp[imax:imax−11] } – { 1'b0, b12, 1'b0 };
d13 = { 1'b0, inp[imax:imax−12] } – { 1'b0, b13, 1'b0 };
d14 = { 1'b0, inp[imax:imax−13] } – { 1'b0, b14, 1'b0 };
d15 = { 1'b0, inp[imax:imax−14] } – { 1'b0, b15, 1'b0 };
d16 = { 1'b0, inp[imax:imax−15] } – { 1'b0, b16, 1'b0 };

if (c[16] && !d16[16]) begin sz=15; ofs=d16[0:+8]; end
else if (c[15] && !d15[15]) begin sz=14; ofs=d15[0:+8]; end
else if (c[14] && !d14[14]) begin sz=13; ofs=d14[0:+8]; end
else if (c[13] && !d13[13]) begin sz=12; ofs=d13[0:+8]; end
else if (c[12] && !d12[12]) begin sz=11; ofs=d12[0:+8]; end
else if (c[11] && !d11[11]) begin sz=10; ofs=d11[0:+8]; end
else if (c[10] && !d10[10]) begin sz=9; ofs=d10[0:+8]; end
else if (c[9] && !d09[9]) begin sz=8; ofs=d09[0:+8]; end
else if (c[8] && !d08[8]) begin sz=7; ofs=d08[0:+8]; end
else if (c[7] && !d07[7]) begin sz=6; ofs=d07; end
else if (c[6] && !d06[6]) begin sz=5; ofs=d06; end
else if (c[5] && !d05[5]) begin sz=4; ofs=d05; end
else if (c[4] && !d04[4]) begin sz=3; ofs=d04; end
else if (c[3] && !d03[3]) begin sz=2; ofs=d03; end
else if (c[2] && !d02[2]) begin sz=1; ofs=d02; end
else begin sz=0; ofs=d01; end

end

always @(posedge CK)
begin

$display("HD: new=%1b index=%1d sz=%1d, to=%1d, ofs=%1d, 
tc=%2x", new, index, sz, to_out, ofs, tc_out);

$display("HUFFSTREAM: %b %b", new, inp);

}
huffcode <= tc_out;
szb <= sz;

nextstartin <= startin;
if (startin)
begin
    new2 <= new;
    foundmarkert = 0;
    outshiftt = 0;
end
else
begin
    if (decodestall == 1 && flush != 0 && inp[imax:imax-11] == 12'b11111111101)
    begin
        marker <= inp[imax-15+:3];
        foundmarkert = 1;
    end
    else
    begin
        foundmarkert = 0;
    end

    new2 <= new;

'ifdef PREDICTOR

    outshiftt = 0;

    if (decodestall!=2)
    begin

        if ( writefifofull ) // have to stop all processing if write fifo to RAM is near full
        begin
            outshiftt = 0;
        end
    else if (nextlast || blkcount>=blkmax) // flush out data at end of image or inbetween mcus
    begin
        outshiftt = 0;
    end
    else if ( stall || ( new==0 && coefindex==63)) // have to stall at end of block to wait for restart markers
    begin
        outshiftt = 0;
    end

'endif PREDICTOR
end
else
begin
  if (runcount)
  begin
    if (runcount==1 && (predicton1 || new) && coefindexfull[5:0]!=63)
      begin
        outshiftt = tc_out[3:0] + sz + 1'b1;
      end
    else
      begin
        outshiftt = 0;
      end
  end
else
begin
  if (foundmarker!=0)
    begin
      outshiftt = 16;
    end
else if (decodestall==1 || new)
begin
  if (tc_out)
    begin
      outshiftt = tc_out[3:0] + sz + 1'b1;
    end
  else
    begin
      outshiftt = sz + 1'b1;
    end
end
else
begin
  begin
    outshiftt = 0;
  end
end
end
end
`$display("outshiftt=%d",outshiftt);

`else

    outshiftt = 0;

`endif

end

outshift1 <= outshiftt;

foundmarker <= foundmarkert;

if ( stall && outshift1) `$display("LOOKHERE1");

end

reg [2:0] shifttemp;

always @(posedge CK)
begin

nextstartin2 <= nextstartin;
if (nextstartin)
begin
    coefindexfull <= 0;
    blkcount <= 0;
    runcount = 0;
    lastcode = 0;
    outshift2 = 0;
    flush = 0;
    restartcount = restart;
    pastnew2 = 1;
    pastfoundmarker = 0;
    mcucount = 0;
    index <= { blkindex[0], 1’b0 };
    decodestall <= 0;
    nextlast <= 0;
    error <= 0;
    lastmarker <= 0;
end
else
begin


advance = 1;
coefindex = coefindexfull [5:0];
codelength = huffcode [3:0];
codetemp = inp << szb;
codeval [14:0] = codetemp [30:16];
codeval [15] = !codeval [14];
codeval = codeval >>> (15 – codelength);
if (codeval [15]) // signed number, correct it
begin
   codeval = codeval+1’b1;
end
if (stall && outshift2) $display("LOOKHERE2");
if (error) $display("ERROR TRIGGERED");
if (!pastnew2) pastnew2=new2;
if (!pastfoundmarker) pastfoundmarker=foundmarker;
$display("HUFFDECODE: decodestall=%2b new2=%1b pastnew2=%1b stall=%1b
runcount=%1d pastfoundmarker=%1b
marker=%1d", decodestall, new2, pastnew2, stall, runcount, pastfoundmarker, marker);
if (writefifofull && outshift1==0) // have to stop all processing if write fifo to
RAM is near full
begin
   $display("HUFFWRITEFIFOSTALL");
   intcode = 0;
   advance = 0;
   if (!stall) outshift2 = 0;
end
else if (nextlast || blkcount>=blkmax) // flush out data at end of image or
inbetween mcus
begin
   intcode = 0;
   if (!stall) outshift2 = 0;
end
else if (stall || (pastnew2==0 && coefindex==63)) // have to stall at end of
block to wait for restart markers
begin
   intcode = 0;
   advance = 0;
   if (!stall) outshift2 = 0;
end
else
begin

if ( runcount>1 )
begin
intcode = 0;
runcount = runcount−1’b1;
outshift2 = 0;
end
else
begin
if ( runcount==1 )
begin
intcode = lastcode;
runcount = 0;
outshift2 = 0;
end
else
begin
if ( decodestall==0 && pastfoundmarker!=0)
begin
pastfoundmarker = 0;
lastcode = 0;
runcount = 0;
intcode = 0;
outshift2 = 16;
advance = 0;
flush = 0;
if (marker!=lastmarker) error < 1;
lastmarker <= marker+1;
end
else if (decodestall==0 && pastnew2 )
begin
$display("HUFF: %02x %02x %4d",huffcode,szb+1,countercheck);
countercheck=countercheck+1’b1;
if (huffcode)
begin
runcount = huffcode[7:4];
‘ifdef PREDICTOR
if (runcount>1) prediction1=1; else prediction1=0;
‘endif
lastcode = codeval;
if (runcount) intcode = 0; else intcode = codeval;
outshift2 = codelength + szb + 1’b1;
end
else
begin
...
```verilog
A. VERILOG CODE

else
begin
  if (coefindex == 0)
begin
    runcount = 0;
  end
else if (coefindex == 63)
begin
    runcount = 0;
end
else
begin
  runcount = ~coefindex;
end
lastcode = 0;
intcode = 0;
outshift2 = szb + 1'b1;
end
end
else
begin
  lastcode = 0;
runcount = 0;
intcode = 0;
outshift2 = 0;
advance = 0;
end
end
end

ifdef PREDICTOR
  if (outshift2 != outshift1)
begin
  $display("+++++ %d %d %d",outshift2,outshift1,outshift2-outshift1);
end

  if (outshift1 > outshift2)
begin
  $display("BAD %d %d %d",outshift2,outshift1,outshift2-outshift1);
  $finish;
end

  outshift2 = outshift2 - outshift1;
endif
```
end // stall
nextvalid <= advance;

if (advance)
begin

nextcode <= intcode;
nextcoefindexfull <= coefindexfull;
nextblkquant <= blkquant[blkcount];
nextblkcount <= blkcount;

pixels = pixels + 1;
flush=0;

// only change index if we advance to next coef
coefindexfull <= coefindexfull + 1'b1;

if (coefindex == 63)
begin
$display("END BLOCK %1d",restartcount);
if (blkcount == blocksm1)
begin
blkcount <= 0;
index <= { blkindex[0], 1'b0 };
mcucount = mcucount + 1;

$display("END MCU %1d",mcucount);
if (mcucount>=totalmcu) nextlast <= 1;

if (restartcount == 1)
begin
$display("RESTART %1d %1d",mcucount,remain);
flush = 1;
restartcount = restart;

ifdef PREDICTOR

shifttemp = -(outshift2[2:0] + outshift1[2:0] - remain);
outshift2 = outshift2 + shifttemp;

else

shifttemp = -(outshift2[2:0] - remain);
outshift2 = outshift2 + shifttemp;

endif
end
else if (restartcount == 0)
begin
  // Nothing
end
else
begin
  restartcount = restartcount - 1'b1;
end
end
else
begin
  blkcount <= blkcount + 1'b1;
  index <= { blkindex[blkcount+1], 1'b0 };
end
$display("INDEXCHANGE");
decodestall <= 2;
end
else if (coefindex == 0)
begin
  index <= { blkindex[blkcount], 1'b1 };  
  $display("INDEXCHANGE");
  decodestall <= 2;
end
else
begin
  if (decodestall>0) decodestall <= decodestall - 1'b1;
end
end
else
begin
  if (! stall && decodestall>0) decodestall <= decodestall - 1'b1;
end // advance

if (outshift1 || outshift2) pastnew2=0;
end // nextstartin

$display(”huffcode=%b codeval=%b intcode=%b %6d outshift2=%1d coefindex=%1d runcount=%1d
flush=%b”,huffcode,codeval,intcode,intcode,outshift2,coefindex,runcount,flush);

end
A. VERILOG CODE

```verilog
always @(negedge CK )
begin
  if (nextstartin==0)
  begin
    clocks = clocks + 1;
  end
  $display("*************** CLOCK
          *******************");
end

reg [1:0] tempcomp;
reg [3:0] outblkdelay,outblkdelay2;
reg [3:0] outbankdelay,outbankdelay2;
reg [3:0] outbankenabledelay,outbankenabledelay2;
reg [15:0] outbankslastdelay,outbankslastdelay2;
reg outupper;
reg intcolourgo;
reg outlastdelay,outlastdelay2,outlastdelay3;

always @(posedge CK )
begin
  if (nextstartin2)
  begin
    dcaccum = 0;
    outblk <= 15;
    outblkdelay <= 15;
    outblkdelay2 <= 15;
    outbank <= 15;
    outbankdelay <= 15;
    outbankdelay2 <= 15;
    outbankenabledelay <= 0;
    outbankenabledelay2 <= 0;
    outbankenable <= 0;
    outbankslastdelay <= 0;
    outbankslast <= 0;
    bankcount[0] <= 0;
    bankcount[1] <= 0;
    bankcount[2] <= 0;
    bankcount[3] <= 0;
    outvalid <= 0;
    outlast <= 0;
    outlastdelay <= 0;
    outlastdelay2 <= 0;
    outlastdelay3 <= 0;
    outbankslast <= 0;
  
```

```
A. VERILOG CODE

```verilog
outcolourgo <= 0;
intcolourgo <= 0;
end
else
begin

  tempcomp = blkcomp[nextblkcount];
  int2code = nextcode;

  if (flush) dcaccum = 0;
  if (nextvalid && nextcoefindexfull[5:0] == 0)
  begin
    if (dcaccum[tempcomp]) int2code = int2code + dc[tempcomp];
    dc[tempcomp] <= int2code;
    dcaccum[tempcomp] = 1;

    // doesn't work for greyscale

    // on the beginning of a new mcu, save out the last bank references
    if (nextblkcount==0)
    begin
      outbankslastdelay =
      {
        bankcount[0], bankcount[1], bankcount[2], bankcount[3]
      };
    end

    if (outblkdelay==0)
    begin
      if (blkmax==1) // for greyscale
      begin
        outbankslast <= { outbankdelay, 12'b000000000000 };
        outlastdelay3 <= nextlast;
        outlastdelay2 <= outlastdelay3;
        outlastdelay <= outlastdelay2;
      end
      else
      begin
        outbankslast <= outbankslastdelay;
        outlastdelay <= nextlast;
      end

      intcolourgo <= 1;
      outcolourgo <= intcolourgo;
      outlast <= outlastdelay;
    end
  end
```

123
$display("YES %1b %1d %1d %1d %1d %1d", nextlast, nextblkcount, bankcount[0], bankcount[1], bankcount[2], bankcount[3]);

// consider checking for last here so you just pad the output
if (nextblkcount < blkmax) // update banks only if it is real MCU data, otherwise, just pad the IDCTs
begin
    bankcount[tempcomp] <= bankcount[tempcomp] + 1;
    outbankdelay2 <= bankcount[tempcomp];
    outbankenabledelay2 <= (1 << tempcomp);
end
else
begin
    outbankdelay2 <= 3; // it shouldn’t write here anyways
    outbankenabledelay2 <= 0; // no writes
end
outbankdelay <= outbankdelay2;
outbank <= outbankdelay;
outbankenabledelay <= outbankenabledelay2;
outbankenable <= outbankenabledelay;
outblkdelay2 <= nextblkcount;
outblkdelay <= outblkdelay2;
outblk <= outblkdelay;
end

tocode <= int2code;
quantindex <= {nextblkquant, nextcoefindexfull[5:0]};
outupper <= nextcoefindexfull[6+:1];
outvalid <= nextvalid;
end
end

#include "zigzagcont.v"

assign outscaled = outcode * tq_out;
assign outindex = {outupper, quantindex[5:0]};
assign outaddr = {outupper, zz[quantindex[5:0]]};

endmodule
A.7 dpram.v, dparam.v, dpsram.v, asyncmem.v

// 'define DEBUGDPRAM
module dpram( ck, wr_en, wr_addr, wr_data, rd_addr, rd_data );
parameter ADDR = 6;
parameter DATA = 32;
input ck;
input wr_en;
input [ADDR−1:0] wr_addr;
input signed [DATA−1:0] wr_data;
input [ADDR−1:0] rd_addr;
output signed [DATA−1:0] rd_data;
reg signed [DATA−1:0] mem[(2**ADDR)−1:0];

integer i, j;
always @(posedge ck)
begin
    rd_data <= mem[rd_addr];
    if (wr_en) mem[wr_addr] <= wr_data;
    ifdef DEBUGDPRAM
        if (wr_en) $display("RAM %m write to %1d with %1d",wr_addr,wr_data);
        if ( (wr_addr & 63) == 0 )
            begin
                $display("RAM %m");
                for (i=0;i<(2**ADDR)/8;i=i+1)
                    begin
                        $write("+ ");
                        for (j=0;j<8;j=j+1)
                            begin
                                $write("%d ",mem[i*8+j]);
                            end
                    $display("\n");
                end
            endif
        end
    endif
endmodule

dpram.v

module dparam( ck, wr_en, wr_addr, wr_data, rd_addr, rd_data );
A. VERILOG CODE

```verilog
parameter ADDR = 6;
parameter DATA = 32;
input ck;
input wr_en;
input [ADDR-1:0] wr_addr;
input signed [DATA-1:0] wr_data;
input [ADDR-1:0] rd_addr;
output [DATA-1:0] rd_data;

reg signed [DATA-1:0] mem[(2**ADDR)-1:0];
assign rd_data = mem[rd_addr];

integer i, j;

always @(posedge ck)
begin
    if (wr_en) mem[wr_addr] <= wr_data;
end
endmodule

dparam.v

module dpsram( ck, wr_en, wr_addr, wr_data, rd_addr, rd_data );
parameter ADDR = 10;
parameter DATA = 32;
input ck;
input wr_en;
input [ADDR-1:0] wr_addr;
input [DATA-1:0] wr_data;
input [ADDR-1:0] rd_addr;
output [DATA-1:0] rd_data;
reg [DATA-1:0] rd_data;

reg [DATA-1:0] mem[(2**ADDR)-1:0];

always @(posedge ck)
begin
    rd_data <= mem[rd_addr];
    if (wr_en) mem[wr_addr] <= wr_data;
end
endmodule
```
**A. VERILOG CODE**

dpsram.v

```verilog
module asyncmem( ck, wr_en, wr_data, rd_addr, rd_data );
    parameter ADDR = 6;
    parameter DATA = 8;
    input ck;
    input wr_en;
    input [DATA-1:0] wr_data;
    input [ADDR-1:0] rd_addr;
    output [DATA-1:0] rd_data;

    reg [DATA-1:0] mem[0:(2**ADDR)-1];

    assign rd_data = mem[rd_addr];

    always @(posedge ck)
    begin
        if (wr_en) begin
            mem[rd_addr] = wr_data;
        end
    end

endmodule
```

asyncmem.v

**A.8 idctrow.v and idctcol.v**

```verilog
module idctrowg(
    input clk,
    input valid_input,
    input [2:0] index,
    input signed [15:0] inputdata,
    output signed [21:0] outputdata
);

    parameter insize=16,iprec=11,oprec=3;

    integer W1 = 2841;
    integer W2 = 2676;
    integer W3 = 2408;
    integer W5 = 1609;
```
integer W6 = 1108;
integer W7 = 565;
reg signed [31:0] x [0:8];
reg signed [31:0] y [0:8];
reg signed [21:0] outbuf [0:7];

assign outputdata = outbuf[(index+4)&7];

// This implements the ROW IDCT from nanojpeg

integer i;
always @(posedge clk)
begin
    if (valid_input)
        begin
            case(index)
                0:
                    begin
                        x[0] <= (inputdata <<< 11) + 128; //pre first stage

y[2] <= x[0] + x[6];
y[0] <= x[0] - x[6];
                    end

                1:
                    begin
                        x[1] <= inputdata; //pre first stage

y[1] <= x[1] - x[5];
                    end

                2:
                    begin
                        x[2] <= inputdata; //pre first stage

                        y[6] <= ((181 * (y[1] + y[7]) + 128) >> 8);
y[1] <= ((181 * (y[1] - y[7]) + 128) >> 8);
                    end
        end
end
3:  
begin  
\begin{align*}  
x[3] & \leftarrow \text{inputdata}; \quad // \text{pre first stage} 
\end{align*}

\begin{align*}  
\text{outbuf}[0] & \leftarrow (y[3] + y[4]) \ggg 8; 
\text{outbuf}[1] & \leftarrow (y[2] + y[6]) \ggg 8; 
\text{outbuf}[2] & \leftarrow (y[0] + y[1]) \ggg 8; 
\text{outbuf}[5] & \leftarrow (y[0] - y[1]) \ggg 8; 
\text{outbuf}[3] & \leftarrow (y[8] + y[5]) \ggg 8; 
\text{outbuf}[4] & \leftarrow (y[8] - y[5]) \ggg 8; 
\text{outbuf}[6] & \leftarrow (y[2] - y[6]) \ggg 8; 
\text{outbuf}[7] & \leftarrow (y[3] - y[4]) \ggg 8; 
\end{align*}
end

4:  
begin  
\begin{align*}  
x[8] & \leftarrow x[0] + (\text{inputdata} \lll 11); \quad // \text{second stage line 1} 
x[0] & \leftarrow x[0] - (\text{inputdata} \lll 11); \quad // \text{second stage line 2} 
\end{align*}
end

5:  
begin  
\begin{align*}  
x[5] & \leftarrow W3 \ast x[3] + W5 \ast \text{inputdata}; \quad // \text{first stage line 5} 
x[3] & \leftarrow W3 \ast \text{inputdata} - W5 \ast x[3]; \quad // \text{first stage line 6} 
\end{align*}
end

6:  
begin  
\begin{align*}  
x[6] & \leftarrow W6 \ast x[2] - W2 \ast \text{inputdata}; \quad // \text{second stage line 5} 
x[2] & \leftarrow W6 \ast \text{inputdata} + W2 \ast x[2]; \quad // \text{second stage line 4} 
\end{align*}
end

7:  
begin  
\begin{align*}  
x[1] & \leftarrow W7 \ast \text{inputdata} + W1 \ast x[1]; \quad // \text{first stage line 2} 
x[7] & \leftarrow W7 \ast x[1] - W1 \ast \text{inputdata}; \quad // \text{first stage line 3} 
\end{align*}
end  
endcase  
end
endmodule

idctrow.v

module idctcolg(clk,valid,input,index,inputdata,outputdata);

\begin{align*}  
\text{parameter} \quad \text{insize}=22, \text{iprec}=11, \text{pprec}=3; 
\text{parameter} \quad \text{outsize} = 9; 
\end{align*}
A. VERILOG CODE

```verilog
parameter intsize = isize+3+(iprec–pprec);

input clk;
input valid_input;
input [2:0] index;
input signed [isize–1:0] inputdata;
output signed [ousize–1:0] outputdata;

integer W1 = 2841;
integer W2 = 2676;
integer W3 = 2408;
integer W5 = 1609;
integer W6 = 1108;
integer W7 = 565;

reg signed [32:0] x [0:8];
reg signed [32:0] y [0:8];
reg signed [ousize–1:0] outbuf [0:7];

function [ousize–1:0] clamp;
input signed [intsize–1:0] inp;
begin
  if ( inp[intsize–1:ousize–1] == {{{intsize–ousize+1}{1'b0}}}) || (inp[intsize–1:ousize–1] == {{{intsize–ousize+1}{1'b1}}}) // good range
    clamp = inp[0+:ousize]; // copy bits
  end
  else if ( inp[intsize–1+:1] == 1'b1 ) // negative
    clamp = { 1'b1, {{ousize–1}{1'b0}} };
  end
  else // positive
    clamp = { 1'b0, {{ousize–1}{1'b1}} };
  end
endfunction

assign outputdata = outbuf[(index+4)&7];

// This implements the COL IDCT from nanojpeg

always @(posedge clk)
begin
```

---

// This implements the COL IDCT from nanojpeg
if (valid_input)
begin
    case(index)
    0:
    begin
        x[0] <= (inputdata <<< 8) + 8192; //pre first stage
        x[1] <= x[1] >>> 3;
        y[2] <= x[0] + x[6];
        y[0] <= x[0] − x[6];
    end
    1:
    begin
        x[1] <= inputdata; //pre first stage
        y[1] <= x[1] − x[5];
    end
    2:
    begin
        x[2] <= inputdata; //pre first stage
        y[6] <= ((181 * (y[1] + y[7]) + 128) >>> 8);
        y[1] <= ((181 * (y[1] − y[7]) + 128) >>> 8);
    end
    3:
    begin
        x[3] <= inputdata; //pre first stage
        outbuf[0] <= clamp( (y[3] + y[4]) >>> 14 );
        outbuf[1] <= clamp( (y[2] + y[6]) >>> 14 );
        outbuf[3] <= clamp( (y[8] + y[5]) >>> 14 );
        outbuf[4] <= clamp( (y[8] − y[5]) >>> 14 );
        outbuf[2] <= clamp( (y[0] + y[1]) >>> 14 );
        outbuf[5] <= clamp( (y[0] − y[1]) >>> 14 );
        outbuf[6] <= clamp( (y[2] − y[6]) >>> 14 );
        outbuf[7] <= clamp( (y[3] − y[4]) >>> 14 );
    end
end
4:
begin
  x[8] <= x[0] + (inputdata << 8); //second stage line 1
  x[0] <= x[0] - (inputdata << 8); //second stage line 2
end
5:
begin
end
6:
begin
end
7:
begin
  x[1] <= W7 * inputdata + W1 * x[1] + 4; //first stage line 2
end
endcase
end
endmodule

idctcol.v

A.9 colourmap.v and zigzagcont.v

module colourmap9( CK, enable, comps, bankin, valid, combined);
input CK;
input enable;
input [2:0] comps;
input [4*9–1:0] bankin;
output valid;
reg valid;
output [31:0] combined;
reg [31:0] combined;

reg signed [8:0] y, cb, cr;
reg signed [15:0] cb2, cr2;
reg signed [17:0] tempy;
reg signed [17:0] temp, tempg, tempb;
reg [7:0] temp4c, temp4m, temp4y, temp4k, tempa;
reg [17:0] tempm1, tempm2, tempm3;

always @ (posedge CK)
begin
  valid <= enable;
  if (enable)
  begin
    if (comps==1 || comps==3)
    begin
      y = bankin[0+:9];
      tempy = ( (y + 128) << 8 ) | 128;
      if (comps==1)
      begin
        cb = 0;
        cr = 0;
      end
      else
      begin
        cb = bankin[9+:9];
        cr = bankin[18+:9];
      end
      cb2 = cb;
      cr2 = cr;
      tempy = tempy + ( 359 * cr2 );
      tempg = tempy - ( 88 * cb2 ) - ( 183 * cr2 );
      tempb = tempy + ( 454 * cb2 );
      combined[24+:8] <= 0;
      combined[0+:8] <= ( tempb[17:16]==2'b00 ) ? tempb[15:8] : ( tempb[17:16]==2'b01 ) ? 255 : 0; // green
    end
  end
end
A. VERILOG CODE

```verilog
tempb[17:16]==2’b01 ) ? 255 : 0; // blue
end
else
begin
  temp4c = (128−bankin[0+:9]) ^ 255;
  temp4m = (128−bankin[9+:9]) ^ 255;
  temp4y = (128−bankin[18+:9]) ^ 255;
  temp4k = (128−bankin[27+:9]) ^ 255;

  tempm1 = { temp4c, 1’b1 } * { temp4k, 1’b1 };
  tempm2 = { temp4m, 1’b1 } * { temp4k, 1’b1 };
  tempm3 = { temp4y, 1’b1 } * { temp4k, 1’b1 };

  combined[24+:8] <= 0;
  combined[0+:8] <= tempm1[17:10]; // red
  combined[8+:8] <= tempm2[17:10]; // green
  combined[16+:8] <= tempm3[17:10]; // blue
end
else
begin
  combined <= 0;
end
end
endmodule

colourmap.v

assign zz[00]=0;
assign zz[01]=1;
assign zz[02]=8;
assign zz[03]=16;
assign zz[04]=9;
assign zz[05]=2;
assign zz[06]=3;
assign zz[07]=10;
assign zz[08]=17;
assign zz[09]=24;
assign zz[10]=32;
assign zz[12]=18;
assign zz[13]=11;
```
assign zz[14] = 4;
assign zz[15] = 5;
assign zz[16] = 12;
assign zz[17] = 19;
assign zz[18] = 26;
assign zz[19] = 33;
assign zz[20] = 40;
assign zz[21] = 48;
assign zz[22] = 41;
assign zz[23] = 34;
assign zz[24] = 27;
assign zz[25] = 20;
assign zz[26] = 13;
assign zz[27] = 6;
assign zz[28] = 7;
assign zz[29] = 14;
assign zz[30] = 21;
assign zz[31] = 28;
assign zz[32] = 35;
assign zz[33] = 42;
assign zz[34] = 49;
assign zz[35] = 56;
assign zz[36] = 57;
assign zz[37] = 50;
assign zz[38] = 43;
assign zz[39] = 36;
assign zz[40] = 29;
assign zz[41] = 22;
assign zz[42] = 15;
assign zz[43] = 23;
assign zz[44] = 30;
assign zz[45] = 37;
assign zz[46] = 44;
assign zz[47] = 51;
assign zz[48] = 58;
assign zz[49] = 59;
assign zz[50] = 52;
assign zz[51] = 45;
assign zz[52] = 38;
assign zz[53] = 31;
assign zz[54] = 39;
assign zz[55] = 46;
assign zz[56] = 53;
assign zz[57] = 60;
assign zz[58] = 61;
A. VERILOG CODE

assign zz[59]=54;
assign zz[60]=47;
assign zz[61]=55;
assign zz[62]=62;
assign zz[63]=63;

zigzagcont.v
Appendix B

C Code

B.1 hwjpeg.c

/∗
This program is the control logic for an FPGA JPEG Decoder.
It expects certain hardware to be present at certain addresses.
It also expects to be run on a kernel that allows access to /dev/mem in order to
control the FPGA.

If there’s something in here you don’t understand, email me and yell at me for writing
spaghetti code.
*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdint.h>
#include <time.h>
#include <unistd.h>
#include <inttypes.h>
#include <sys/time.h>
#include <sys/resource.h>
```c
#include "hwmap.h"

#define SHARED_MEM 0x1c000000
#define SHARED_MEM_SIZE 0x04000000

#define CTRL_ADDR 0x7E400000
#define CTRL_SIZE 0x0000ffff

void usage() {
    printf("USAGE:\n    n./hwjpeg.o file.jpg [0,1 -> nowrite,write]\n'\n");
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        usage();
        return 1;
    }

    struct rusage ru;
    struct timeval utime;

    int mem_dev = open("/dev/mem", O_RDWR | O_SYNC);
    if (mem_dev != -1) {
        unsigned volatile *p_ctrl = hwmap(mem_dev, CTRL_ADDR, CTRL_SIZE);

        if (p_ctrl == NULL) {
            printf("ctrl : hwmap failed\n");
            return 1;
        }

        unsigned volatile *p_shared = hwmap(mem_dev, SHARED_MEM, SHARED_MEM_SIZE);

        if (p_shared == NULL) {
            printf("sh: shared: hwmap failed\n");
            return 1;
        }

        else {
```
```
//open image file, fread in first x bytes to read header
//get size of canvas from h/w, multiply that by 32 to get buffer size
//write job is still off at this point, so the write buffer is full
//start the write job and do polling to see if r or w has finished
//if w is done and r is not, set the write addr and inc the write num
//if r is done and w is not, set the read addr and inc the read num
//if both are done, set both addrs and increment both nums
//if neither are done, sit there and twiddle thumbs
//don’t forget to fread more data if read is done
//the hardware will let us know when the image is done processing
//so there’s no need to worry about file read errors

unsigned const words_to_read = 0x4000; //number of 16 word blocks to read,
word is 4 bytes
unsigned read_size = words_to_read * 16 * 4;

FILE *image = fopen(argv[1], "rb");
fread((void*)p_shared, read_size, 1, image); //initial read

uint16_t p_ctrl0 = p_ctrl[0];
p_ctrl0 = p_ctrl0 ^ 0x0010;

unsigned imsize_x, imsize_y; //imsize is actually canvas size
unsigned write_addr;

p_ctrl[4] = 0;
p_ctrl[2] = 0;

p_ctrl[0] = p_ctrl0; //reset the h/w
p_ctrl0 = p_ctrl0 ^ 0x0010;
p_ctrl[0] = p_ctrl0; //pull the h/w out of reset

p_ctrl[1] = SHARED_MEM; //set the read address
p_ctrl[2] = words_to_read; //read buffer set
write_addr = SHARED_MEM + read_size;
p_ctrl[3] = write_addr; //set the write addr

p_ctrl0 = (p_ctrl0 + 0x101) & 0x00f0; //start the job
p_ctrl[0] = p_ctrl0;

while ((p_ctrl[0] & 0x00f) != (p_ctrl0 & 0x00f)); //printf("%08x\n", p_ctrl[0]);

if (((p_ctrl[10] & 0x00f00000) != 0)
{  printf("Cannot handle this file\n");

}
exit(-5);
}

imsize.x = (p_ctrl[8] >> 16) * 8; //read the reg for x_size
imsize.y = (p_ctrl[8] & 0xffff) * 8; //read the reg for y_size
unsigned canvas_fs = imsize.x * imsize.y * 4;

unsigned write_block_start = write_addr;
int ssmaxy = (p_ctrl[10] & 0x30000000) >> 28;
int ssfactor;
if (ssmaxy == 0) ssfactor = 1; //subsampling rates of the image affect the
  write buffer size
else if (ssmaxy == 1) ssfactor = 2;
else if (ssmaxy == 2) ssfactor = 4;
else exit(-1);
int ssmaxx = (p_ctrl[10] & 0xc0000000) >> 30;
int numcomp = (p_ctrl[10] & 0x07000000) >> 24;
unsigned write_block_increment = imsize.x * ssfactor;
unsigned write_block_increment_multiplier = (SHARED_MEM_SIZE - read_size) / (write_block_increment * 32);
write_block_increment *= write_block_increment_multiplier;
//The above lines base the write block size on how much available memory is left
  after allocating a read buffer

unsigned write_block_size = write_block_increment;
printf("%u,%u\n", read_size, write_block_size);
p_ctrl[4] = write_block_increment;//set the write size (# of 32 byte blocks)
p_ctrl0 = (p_ctrl0 + 1) & 0x0f0f;
p_ctrl[0] = p_ctrl0;

FILE *verifyhw;
if (atoi(argv[2])) verifyhw = fopen("/var/nfs/verifyhw.bin", "wb");

/ *

Because of the design of the hardware combined with the design of the software
and the intrinsics of JPEGs,
the following infinite loop has some very convoluted logic.

The control/status register, or p_ctrl[0], is used to start and monitor both the
read and write jobs,
as well as the overall status of the image decode.
READING p_ctrl[0] ->
  p_ctrl[0] & 0x000f = latest FINISHED write job
  p_ctrl[0] & 0x0f00 = latest FINISHED read job
  p_ctrl[0] & 0x8000 = is JPEG decode finished?

WRITING p_ctrl[0] ->
  p_ctrl[0] & 0x000f = next write job starts on write of this nibble
  p_ctrl[0] & 0x0f00 = next read job starts on write of this nibble

MEMORY ORGANIZATION

READ BUF
-----------------
WRITE BUF

-----------------
SYSTEM MEMORY

/*
 uint32_t old_job = p_ctrl0;
 unsigned image_width = (p_ctrl[9] & 0xFFFF0000) >> 16;
 unsigned image_height = (p_ctrl[9] & 0x0000FFFF);
 unsigned image_bytes = image_width * image_height * 4;
 unsigned bytes_to_write = image_bytes;
 unsigned bytes_written = 0;
 long long utimes = 0, utimeu = 0;

 getrusage(RUSAGE_SELF, &ru);
 utime = ru.ru_utime;
 utimes += utime.tv_sec;
 utimeu += utime.tv_usec;

 while (1)
 {
  getrusage(RUSAGE_SELF, &ru);
  utime = ru.ru_utime;
  utimes -= utime.tv_sec;
  utimeu -= utime.tv_usec;

  if ((p_ctrl[0] & 0xf00) == (old_job & 0xf00)) //if read is done
   { fread((void *)p_shared, read_size, 1, image); //read from file to read buffer

*/
old_job = (old_job + 0x100) & 0xf0f;  //set new read job
p_ctrl[0] = old_job;                    //start new read job

if ((p_ctrl[0] & 0x00f) == (old_job & 0x00f)) //if write is done
{
    /*
     ***** IMPORTANT *****
     The writes are tricky because the hardware has only a small region to write to.
     To get around this we move the "write" region back by the amount we’ve just written to allow
     it to write to the same place repeatedly. This also requires that we increase the size of the
     write by the amount we write every time. This way, the final write on each write job from hardware to buffer
     is always within the memory range allocated to the write buffer. This is a little counter-intuitive so make
     sure to understand this completely before altering the code.

     The best way to turn this into driver ready code is to rework how the hardware handles writing.
     */

    int j = 0;
    if (bytes_to_write > (write_block_increment * 32)) //if we’re not on the last write
    {
        for (int i = 0; i < write_block_increment * 32; i += imsize_x * 4)
        {
            if (atoi(argv[2])) fwrite((void *)&p_shared[(read_size/4) + (i/4)],
                image_width * 4, 1, verifyhw);
            /*
             (read_size/4) -> moves the pointer forward past the read buffer
             (i/4) -> i is incremented by canvas line (NOT image line)
             image_width*4 -> we want to write the bytes inside the image and
             discard the rest of the canvas
             write_block increment is in the number of 32 byte blocks to read,
             so multiplying it by 32 gives us a value in bytes
             */
        }
    }
}
j += image_width * 4;  
}
bytes_written += j; //tracks the number of bytes actually written to file  
for the final write in the else block
}
else
{
    int i = 0;
    while (bytes_written < image_bytes)
    {
        if (atoi(argv[2])) fwrite((void *)&p_shared[(read_size/4) + (i/4)],  
            image_width * 4, 1, verifyhw);
        i += imsize_x * 4;
        bytes_written += image_width * 4;
        // This is necessary for the final write because it may not be aligned to  
the preset boundaries from above
    }
}
bytes_to_write -= write_block_increment * 32;  //keep track of the bytes  
left to write
write_block_start -= write_block_increment * 32;  //move the start "address"  
back
write_block_size += write_block_increment;  //make the write size bigger to  
continue writing the image data in the same spot
p_ctrl[4] = write_block_size;  //update the values in the h/w regs
p_ctrl[3] = write_block_start;
old_job = (old_job + 1) & 0xf0f;

p_ctrl[0] = old_job;
if (((p_ctrl[0] & 0x8000) == 0x8000)  //if the job is complete, break out  
of the infinite while loop
{
    getrusage(RUSAGE_SELF, &ru);
    utime = ru.ru_utime;
    utimes += utime.tv_sec;
    utimeu += utime.tv_usec;
    break;
}
}
getrusage(RUSAGE_SELF, &ru);
B. C CODE

endtime = ru.ru_ute;  
etimes += utime.tv_seq;  
etimeu += utime.tv_usc;

}

printf ("%u,%u,%u,%u,%u", image_width, image_height, ssmaxx, ssfactor, 
umcomp);

for (int i = 1; i <= 3; i++) printf("%u", p_ctrl[i]);

printf("%lld,%lld", utimes, utimeu);

if (atoi(argv[2])) fflush (verifyhw);
fclose (image);
if (atoi(argv[2])) fclose (verifyhw);
}
munmap((void *)p_shared, SHARED_MEM_SIZE);
munmap((void *)p_ctrl, CTRL_SIZE);
}

else
{
    printf ("open failed
\n");
}

close(mem_dev);
return 0;

}

hwjpeg.c

B.2 hwmap.c and hwmap.h

// This function will return a pointer to a specified region in dev/mem
#include "hwmap.h"

unsigned volatile * hwmap (int mem_dev, uint32_t addr, uint32_t size)
{
    if (mem_dev != 1)
    {
        uint32_t page_mask, page_size, shared_alloc_size ;
        return NULL;
    }
    uint32_t


void *shared_pointer, *shared_virt_addr;

page_size = sysconf(_SC_PAGESIZE);
page_mask = (page_size - 1);

shared_alloc_size = (((size / page_size) + 1) * page_size);

//
unsigned volatile *p_ctrl;

shared_pointer = mmap(NULL,
    shared_alloc_size ,
    PROT_READ | PROT_WRITE,
    MAP_SHARED,
    mem_dev,
    (addr & ~page_mask)
);

if (shared_pointer == MAP_FAILED)
    printf("ctrl mmap failed\n");
else
{
    shared_virt_addr = (shared_pointer + (addr & page_mask));
    return (unsigned volatile *)shared_virt_addr;
}

return NULL;


#include <stdlib.h>
#include <inttypes.h>
#include <unistd.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdio.h>

unsigned volatile * hwmap (int, uint32_t, uint32_t);
B.3  psnr.c

//This program computes the PSNR given a source image and test image of the same
dimensions

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void usage();

int main (int argc, char *argv[])
{
    if (argc != 3) usage();
    else {
        FILE *f1, *f2;
        f1 = fopen(argv[1],"rb");
        f2 = fopen(argv[2],"rb");
        if (f1 == NULL || f2 == NULL)
            usage();
            return 1;
    }

    fseek(f1, 0, SEEK_END);
    long f1size = ftell(f1);
    fseek(f1, 0, SEEK_SET);

    fseek(f2, 0, SEEK_END);
    long f2size = ftell(f2);
    fseek(f2, 0, SEEK_SET);

    if (f1size != f2size)
    {
        printf("files should be the same size: exiting\n\n");
        fclose(f1);
        fclose(f2);
        return 1;
    }

    unsigned char *buf1, *buf2;
    int read_size = 1024*1024*sizeof(char);
    int bytes_to_read = f1size;
    buf1 = (unsigned char *) malloc (read_size);
buf2 = (unsigned char *) malloc (read_size);

if (buf1 == NULL || buf2 == NULL)
{
    printf("buffer malloc error\n");
    fclose (f1);
    fclose (f2);
    return 1;
}

unsigned long long mse = 0;
unsigned max_abs_diff = 0, abs_diff = 0;
unsigned int hist_r[256] = {0};
unsigned int hist_g[256] = {0};
unsigned int hist_b[256] = {0};

while (bytes_to_read > read_size)
{
    fread(buf1, read_size, 1, f1);
    fread(buf2, read_size, 1, f2);
    bytes_to_read -= read_size;
    for (int i = 0; i < read_size/(sizeof(char)); i++)
    {
        if (i % 4 == 3) continue;

        abs_diff = abs(buf1[i] - buf2[i]);

        if (i % 4 == 0) hist_r[abs_diff]++;
        else if (i % 4 == 1) hist_g[abs_diff]++;
        else if (i % 4 == 2) hist_b[abs_diff]++;

        mse += pow(abs_diff, 2);
        if (abs_diff > max_abs_diff)
        {
            max_abs_diff = abs_diff;
        }
    }
}

fread(buf1, bytes_to_read, 1, f1);
fread(buf2, bytes_to_read, 1, f2);
for (int i = 0; i < bytes_to_read/(sizeof(char)); i++)
{
    abs_diff = abs(buf1[i] - buf2[i]);

if (i % 4 == 3) continue;

abs_diff = abs(buf1[i] - buf2[i]);

if (i % 4 == 0) hist_b[abs_diff]++;
else if (i % 4 == 1) hist_g[abs_diff]++;
else if (i % 4 == 2) hist_r[abs_diff]++;

mse += pow(abs_diff, 2);
if (abs_diff > max_abs_diff)
    { max_abs_diff = abs_diff; }
printf("%llu,", mse);
printf("%u,", max_abs_diff);
for (int i = 0; i <= max_abs_diff; i++)
    printf("%u,%u,%u,", hist_r[i], hist_g[i], hist_b[i]);
printf("\n");

free(buf1);
free(buf2);
fclose(f1);
fclose(f2);

void usage()
{
    printf("usage: psnr [image1.bin] [image2.bin]\n");
    printf("image1 should be from libjpeg, image2 from h/w\n\n");
}

psnr.c
B.4 ljjpeg.c and ljpegt.c

These two files generate binary outputs in BGRA format using libjpeg (ljjpeg.c) and libjpeg-turbo (ljpegt.c) for use in the comparison of quality and speed against the SoC module.

```c
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include "libjpeg/jpeglib.h"
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <string.h>

//extern JSAMPLE *image_buffer; //RGB buffer

struct my_error_mgr {
    struct jpeg_error_mgr pub; /* "public" fields */

    jmp_buf setjmp_buffer; /* for return to caller */
};

typedef struct my_error_mgr *my_error_ptr;

METHODDEF(void)
my_error_exit (j_common_ptr cinfo)
{
    /* cinfo->err really points to a my_error_mgr struct, so coerce pointer */
    my_error_ptr myerr = (my_error_ptr) cinfo->err;

    /* Always display the message. */
    /* We could postpone this until after returning, if we chose. */
    (*cinfo->err->output_message) (cinfo);

    /* Return control to the setjmp point */
    longjmp(myerr->setjmp_buffer, 1);
}

int main(int argc, char *argv[])
{
}
B. C CODE

```c
long long utimes = 0, utimeu = 0;
if (argc != 4)
{
    printf("\nUSAGE\n\n./ljpeg file.jpg [0,1,2 -> slow, fast, float] [0,1 ->
/dev/null, /var/nfs/ljpeg.bin]\n\n");
    return 1;
}

int dct_type = atoi(argv[2]) ;
int of_loc = atoi(argv[3]) ;

if (dct_type > 2 || dct_type < 0)
{
    printf("ERROR: invalid dct_type\n");
    return 1;
}

struct rusage ru;
struct timeval utime;
struct jpeg_decompress_struct cinfo;
struct my_error_mgr jerr;
FILE *infile ;
FILE *verify;
JSAMPARRAY output_row_buffer;
int row_stride ;
infile = fopen(argv[1], "rb");
if (infile != NULL)
{
    char outputstr[100];
    strcpy(outputstr, argv[1]) ;
    strcat(outputstr, ". libj .bin");
    // printf("%s
", outputstr);
    if ( of_loc )
        verify = fopen(outputstr, "wb");
    else
        verify = fopen("/dev/null", "wb");
    cinfo.err = jpeg_std_error(&jerr.pub);
}

jerr.pub.error_exit = my_error_exit;

    if (setjmp(jerr.setjmp_buffer))
    {
```

150
fclose (infile);
return 0;
}
jpeg_create_decompress(&cinfo);
jpeg_stdio_src (&cinfo, infile);
(void) jpeg_read_header(&cinfo, TRUE);

/* JDCT_ISLOW: slow but accurate integer algorithm
JDCT_IFAST: faster, less accurate integer method
JDCT_FLOAT: floating-point method
JDCT_DEFAULT: default method (normally JDCT_ISLOW)
JDCT_FASTEST: fastest method (normally JDCT_IFAST) */

// cinfo.out_color_space = JCS_YCbCr;
if (dct_type == 2)
cinfo.dct_method = JDCT_FLOAT;
else if (dct_type == 1)
cinfo.dct_method = JDCT_IFAST;
else
cinfo.dct_method = JDCT_ISLOW;

(void) jpeg_start_decompress(&cinfo);
row_stride = cinfo.output_width * cinfo.output_components;
output_row_buffer = (*cinfo.mem->alloc_array)
       (((j_common_ptr) &cinfo, JPOOL_IMAGE, row_stride, 1);

//clock_gettime(CLOCK_REALTIME, &gettime_end);
//gettime_total += (gettime_end.tv_sec - gettime_start.tv_sec)
// + (gettime_end.tv_nsec - gettime_start.tv_nsec) / 1E9;

char alpha_char = 0;
char *row_temp;
//printf("%d
", row_stride/3*4);
unsigned char *orb_p;
orb_p = output_row_buffer[0];
int num_comp = cinfo.num_components;
if (num_comp == 1)
    row_temp = (char *)malloc(row_stride * 4);
else if (num_comp == 3)
row_temp = (char *)malloc(row_stride / 3 * 4);

// printf("numcomp: %d\n", num_comp);
int j;

getrusage(RUSAGE_SELF, &ru);
utsche = ru.ru_utime;
unises += utime.tv_sec;
unimeu += utime.tv_usec;

while (cinfo.output_scanline < cinfo.output_height)
{

getrusage(RUSAGE_SELF, &ru);
utsche = ru.ru_utime;
unises -= utime.tv_sec;
unimeu -= utime.tv_usec;

(void) jpeg_read_scanlines(&cinfo, output_row_buffer, 1);

j = 0;
if (num_comp == 1)
{
    for (int i = 0; i < row_stride; i++)
    {
        row_temp[j] = orb_p[i];
        row_temp[j + 1] = orb_p[i];
        row_temp[j + 2] = orb_p[i];
        row_temp[j + 3] = alpha_char;
        j += 4;
    }
}
else if (num_comp == 3)
{
    for (int i = 0; i < row_stride; i+=3)
    {
        row_temp[j] = orb_p[i + 2];
        row_temp[j + 1] = orb_p[i + 1];
        row_temp[j + 2] = orb_p[i];
        row_temp[j + 3] = alpha_char;
        j += 4;
    }
}

getrusage(RUSAGE_SELF, &ru);
utsche = ru.ru_utime;
utimes += utime.tv_sec;
utimeu += utime.tv_usec;

if (num_comp == 1)
    fwrite(row_temp, row_stride * 4, 1, verify);
else if (num_comp == 3)
    fwrite(row_temp, row_stride / 3 * 4, 1, verify);

// getrusage(RUSAGE_SELF, &ru);
// utime = ru.ru_utime;
// utimes += utime.tv_sec;
// utimeu += utime.tv_usec;

printf(“%lld,%lld,”, utimes, utimeu);

jpeg_finish_decompress(&cinfo);
jpeg_destroy_decompress(&cinfo);
fclose ( infile );
fclose ( verify );
}
return 0;

ljpeg.c

#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include ”libjpeg-turbo/jpeglib.h”
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <string.h>

//extern JSAMPLE *image_buffer; //RGB buffer

struct my_error_mngr {
    struct jpeg_error_mngr pub; /* ”public” fields */
jmp_buf setjmp_buffer; /* for return to caller */
}

typedef struct my_error_mgr *my_error_ptr;

METHODDEF(void)
my_error_exit (j_common_ptr cinfo)
{
    /* cinfo->err really points to a my_error_mgr struct, so coerce pointer */
    my_error_ptr myerr = (my_error_ptr) cinfo->err;

    /* Always display the message. */
    /* We could postpone this until after returning, if we chose. */
    (*cinfo->err->output_message) (cinfo);

    /* Return control to the setjmp point */
    longjmp(myerr->setjmp_buffer, 1);
}

int main(int argc, char *argv[])
{
    long long utimes = 0, utimeu = 0;
    int (argc != 4)
    {
        printf("\nUSAGE\n/ljpeg file.jpg [0,1,2 -> slow, fast, float] [0,1
        -> /dev/null, /var/nfs/ljpeg.bin]\n\n");
        return 1;
    }

    int dct_type = atoi(argv[2]) ;
    int of_loc = atoi(argv [3]) ;

    if (dct_type > 2 || dct_type < 0)
    {
        printf("ERROR: invalid dct_type\n");
        return 1;
    }

    struct rusage ru;
    struct timeval utime;

    struct jpeg_decompress_struct cinfo;
    struct my_error_mgr jerr;
    FILE *infile;
    FILE *verify;
JSAMPARRAY output_row_buffer;
int row_stride;
infile = fopen(argv[1], "rb");
if (infile != NULL) {
    char outputstr[100];
    strcpy(outputstr, argv[1]);
    strcat(outputstr, ".turbo.bin");
    if (of_loc)
        verify = fopen(outputstr, "wb");
    else
        verify = fopen("/dev/null", "wb");
cinfo.err = jpeg_std_error(&jerr.pub);
}

jerr.pub.error_exit = my_error_exit;

if (setjmp(jerr.setjmp_buffer)) {
    jpeg_destroy_decompress(&cinfo);
    fclose(infile);
    return 0;
}
jpeg_create_decompress(&cinfo);
jpeg_stdio_src(&cinfo, infile);
(void) jpeg_read_header(&cinfo, TRUE);

/*
JDCT_ISLOW: slow but accurate integer algorithm
JDCT_IFAST: faster, less accurate integer method
JDCT_FLOAT: floating-point method
JDCT_DEFAULT: default method (normally JDCT_ISLOW)
JDCT_FASTEST: fastest method (normally JDCT_IFAST)

In libjpeg-turbo, JDCT_IFAST is generally about 5–15% faster than
JDCT_ISLOW when using the x86/x86-64 SIMD extensions (results may vary
with other SIMD implementations, or when using libjpeg–turbo without
SIMD extensions.) For quality levels of 90 and below, there should be
little or no perceptible difference between the two algorithms. For
quality levels above 90, however, the difference between JDCT_IFAST and
JDCT_ISLOW becomes more pronounced. With quality=97, for instance,
JDCT_IFAST incurs generally about a 1–3 dB loss (in PSNR) relative to
JDCT_ISLOW, but this can be larger for some images. Do not use
JDCT_IFAST with quality levels above 97. The algorithm often

155
degenerates at quality = 98 and above and can actually produce a more lossy image than if lower quality levels had been used. Also, in libjpeg-turbo, JDCT_IFAST is not fully accelerated for quality levels above 97, so it will be slower than JDCT_ISLOW. JDCT_FLOAT is mainly a legacy feature. It does not produce significantly more accurate results than the ISLOW method, and it is much slower. The FLOAT method may also give different results on different machines due to varying roundoff behavior, whereas the integer methods should give the same results on all machines.

```c
if (dct_type == 2)
    cinfo.dct_method = JDCT_FLOAT;
else if (dct_type == 1)
    cinfo.dct_method = JDCT_IFAST;
else
    cinfo.dct_method = JDCT_ISLOW;

(void) jpeg_start_decompress(&cinfo);
row_stride = cinfo.output_width * cinfo.output_components;
output_row_buffer = (*cinfo.mem->alloc_sarray)
    ((j_common_ptr) &cinfo, JPOOL_IMAGE, row_stride, 1);

char alpha_char = 0;
char *row_temp;
    //printf("%d\n", row_stride/3*4);
int num_comp = cinfo.num_components;
if (num_comp == 1)
    row_temp = (char *)malloc(row_stride * 4);
else if (num_comp == 3)
    row_temp = (char *)malloc(row_stride / 3 * 4);
unsigned char *orb_p;
    orb_p = output_row_buffer[0];
int j;

getrusage(RUSAGE_SELF, &ru);
utime = ru.ru_utime;
utimes += utime.tv_sec;
utimeu += utime.tv_usec;
```
while (cinfo.output_scanline < cinfo.output_height) {
    getrusage(RUSAGE_SELF, &ru);
    utime = ru.ru_utime;
    utimes -= utime.tv_sec;
    utimeu -= utime.tv_usec;

    (void) jpeg_read_scanlines(&cinfo, output_row_buffer, 1);

    j = 0;
    if (num_comp == 1) {
        for (int i = 0; i < row_stride; i++) {
            row_temp[j] = orb_p[i];
            row_temp[j + 1] = orb_p[i];
            row_temp[j + 2] = orb_p[i];
            row_temp[j + 3] = alpha_char;
            j += 4;
        }
    }
    else if (num_comp == 3) {
        for (int i = 0; i < row_stride; i+=3) {
            row_temp[j] = orb_p[i + 2];
            row_temp[j + 1] = orb_p[i + 1];
            row_temp[j + 2] = orb_p[i];
            row_temp[j + 3] = alpha_char;
            j += 4;
        }
    }

    getrusage(RUSAGE_SELF, &ru);
    utime = ru.ru_utime;
    utimes += utime.tv_sec;
    utimeu += utime.tv_usec;

    if (num_comp == 1)
        fwrite(row_temp, row_stride * 4, 1, verify);
    else if (num_comp == 3)
        fwrite(row_temp, row_stride / 3 * 4, 1, verify);
B. C CODE

```c
getrusage(RUSAGE_SELF, &ru);
// utime = ru.ru_utime;
// utimes += utime.tv_sec;
// utimeu += utime.tv_usec;

printf("%lld,%lld,", utimes, utimeu);
jpeg_finish_decompress(&cinfo);
jpeg_destroy_decompress(&cinfo);
fclose (infile);
fclose (verify);
}
return 0;
}

ljpegt.c
```
Appendix C

Bash Scripts

C.1 iwhbyd.sh

# this is the testing script
# 1. Run turbo->float on board and compare with desktop lib float md5sum
# 2. Run lib->float,fast,slow and turbo->float,fast,slow to measure time
# 2a.Run psnr
# 3. Run hw decoder and measure time and cycles
# 4. PSNR between hw and other things
# 5. Save it all in csv
DIR=/root/hw_jpeg_accel

for f in /var/nfs/jpg/$1/∗.jpg; do
  if [ ! -e $f.csv.hw ]
    then
      continue
    fi

  a='wc -l $f.csv.hw | awk −F' ' '{ print $1 }'"

  if [[ "$a" != "1" ]]
    then
      echo $f.csv.hw

C. BASH SCRIPTS

```bash
if [ -e $f.csv.hw ]
then
  rm -f $f.csv.hw
fi
fi
done

for f in /var/nfs/jpg/$1/*jpg; do
  echo $f
  if [ -e $f.csv.speed ]
    then
      if [ -e $f.csv.hw ]
        then
          continue
        fi
  fi
  a='$DIR/ljpeg $f 2 1'
  floatmd5='md5sum $f.libj.bin | awk -F' ' '{print $1}''
  echo $floatmd5 > $f.b.md5
  goldmd5='cat $f.md5'
  if [ [ "$floatmd5" != "$goldmd5" ]]
    then
      echo $f >> /var/nfs/md5-mismatch.txt
      rm -f "$f".bin
      continue
    fi
  mv $f.libj.bin $f.gold.bin
  if [ ! -e $f.csv.hw ]
    then
      echo "Running compare for $f"
      a='$DIR/ljpeg $f 1 1'
      mv $f.libj.bin $f.libj.fa.bin
      a='$DIR/ljpeg $f 0 1'
      mv $f.libj.bin $f.libj.s.bin
      a='$DIR/ljpegt $f 2 1'
      mv $f.turbo.bin $f.turbo.fl.bin
```
C. BASH SCRIPTS

```bash
a='$DIR/ljpeg $f 1 1'
mv $f.turbo.bin $f.turbo.fa.bin

a='$DIR/ljpeg $f 0 1'
mv $f.turbo.bin $f.turbo.s.bin

a='$DIR/hwjpeg $f 1 | grep -o handle'
if [[ "$a" == "handle" ]] #detect progressive jpegs
  then
    touch verifyhw.bin #allow pccompanion to continue
    touch $f.csv.speed #skip speed run
    #prog=1
  fi
mv /var/nfs/verifyhw.bin $f.hw.bin
fi

#if [ prog == 1 ]
#then
#  sleep 20
#  mv $f /var/nfs/jpg/prog/
#  rm -f $f.*
#  continue
#fi
#prog=0

if [ ! -e $f.csv.speed ]
then
  echo "Running speed for $f"
  #now do speed run
  printf "%%s," "$f" >> $f.csv.speed
  b='ls -l $f | awk -F ' ' {print $5}''
  printf "%%s," "$b" >> $f.csv.speed
  $DIR/hwjpeg $f 0 >> $f.csv.speed
  $DIR/ljpeg $f 2 0 >> $f.csv.speed
  $DIR/ljpeg $f 1 0 >> $f.csv.speed
  $DIR/ljpeg $f 0 0 >> $f.csv.speed
  $DIR/ljpeg $f 2 0 >> $f.csv.speed
  $DIR/ljpeg $f 1 0 >> $f.csv.speed
  $DIR/ljpeg $f 0 0 >> $f.csv.speed
fi

done
```
C.2 pccompanion.sh

DIR=/home/george/Desktop/hw_jpeg_accel

for f in /var/nfs/jpg/$1/*.jpg; do
  if [ ! -e $f.csv.speed ]
    then
      if [ ! -e $f.csv.hw ]
        then
          continue
        fi
    fi
  echo $f
while [ ! -f $f.libj.s.bin ]
  do
    sleep 2
  done
echo " libj .fa"
$DIR/psnr $f.gold.bin $f.libj.fa.bin > $f.l.fa.csv.b

while [ ! -f $f.turbo.fl.bin ]
  do
    sleep 2
  done
echo " turbo. fl"
$DIR/psnr $f.gold.bin $f.turbo.fl.bin > $f.t.fl.csv.b

while [ ! -f $f.turbo.fa.bin ]
  do
    sleep 2
  done
echo " turbo. fa"
$DIR/psnr $f.gold.bin $f.turbo.fa.bin > $f.t.fa.csv.b

while [ ! -f $f.turbo.s.bin ]
  do
    sleep 2
  done
echo " turbo. s"
$DIR/psnr $f.gold.bin $f.turbo.s.bin > $f.t.s.csv.b
C. BASH SCRIPTS

```bash
done
echo " turbo.fa"
$DIR/psnr $f.gold.bin $f.turbo.fa.bin > $f.t.fa.csv.b

while [ ! -f $f.hw.bin ]
do
  sleep 2
done
echo " turbo.s"
$DIR/psnr $f.gold.bin $f.turbo.s.bin > $f.t.s.csv.b

while [ ! -f $f.csv.speed ]
do
  sleep 2
done

if [ ! -s $f.hw.bin ] #if file size is 0 (means progressive was detected)
then
  echo "PROGRESSIVE"
  mv $f /var/nfs/jpg/prog/
  continue
else
  echo " hw"
  $DIR/psnr $f.gold.bin $f.hw.bin > $f.csv.hw
fi

rm -f "$f"*.bin
done
```

pccompanion.sh

C.3  md5gen.sh

```bash
for f in /var/nfs/jpg/$1/*.jpg; do
  if [ ! -e $f.md5 ];
    then
      info=' file $f | grep JPEG'
      if [ [ "$info" == "" ] ];
        then
          mv $f /var/nfs/jpg/notJPEG/
```
C. BASH SCRIPTS

```bash
continue
fi

a='/home/george/Desktop/hw.jpeg_accel/ljpeg $f 2 1'  #run libjpeg float
b='md5sum $f.libj.bin | awk -F' ' '{print $1}' > $f.md5'  #save md5sum
rm -f "$f"*.bin
fi
echo $f

done

md5gen.sh
```
George Kyrtsakas was born in Windsor, Ontario in 1992. In 2014, he completed his Bachelor of Applied Science in Electrical and Computer Engineering as well as his Bachelor of Computer Science at the University of Windsor. He then began working towards his Master of Applied Science at the University of Windsor in Electrical and Computer Engineering with a focus on embedded systems design.