2013

Hardware JPEG Decompression

Dan MacDonald
University of Windsor

Hardware JPEG Decompression

by

Dan MacDonald

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
2013

Hardware JPEG Decompression

by

Dan MacDonald

APPROVED BY:

Boubakeur Boufama
Computer Science

Huapeng Wu
Electrical and Computer Engineering

Roberto Muscedere, Advisor
Electrical and Computer Engineering

May 15, 2013

Declaration of Originality

Abstract

Due to the ever increasing popularity of mobile devices, and the growing number of pixels in digital photography, there becomes a strain on viewing one's own photos. Similar to Desktop PCs, a common trend occurring in the mobile market to compensate for the increased computational requirements is faster and multi-processor systems. The observation that the number of transistors in integrated circuits doubles approximately every 18-24 months is known as Moore's law. Some believe that this trend, Moore's law, is plateauing which enforces alternate methods to aid in computation.

This thesis explores supplementing the processor with a dedicated hardware module to reduce its workload. This provides a software-hardware combination that can be utilized when large and long computations are needed, such as in the decompression of high pixel count JPEG images. The results show that this proposed architecture decreases the viewing time of JPEG images significantly.

I would like to dedicate this work to my family and friends. I thank you for the support and motivation to tinker and find this path.

Acknowledgments

I will always be grateful to my supervisor, Dr. Muscedere, for his vast knowledge, advice, and for bringing this challenging project to my attention.

I'd also like to thank my other committee members, Dr. H. Wu and Dr. B. Boufama, for their support in this research. Lastly, I'd like to thank my dog Maxx for posing in a test image.

Contents

Declaration of Originality **iv**

Abstract **v**

Dedication **vi**

Acknowledgments **vii**

List of Figures **xii**

List of Tables **xiv**

List of Abbreviations **xvi**

1 Introduction **1**

1.1 History of JPEG Images 2

1.2 Overview of Research, Motivation 3

1.3 Organization of Thesis 4

2 The JPEG Standard **6**

2.1 Discrete Cosine Transform 7

2.2 Quantization 10
2.3 Huffman Encoding and Decoding 11
2.4 Field Programmable Gate Array 14
2.4.1 Soft Processor 14
2.5 Summary 15

3 Existing Work in Hardware JPEG Algorithms 16

3.1 libjpeg 17
3.2 Inverse Discrete Cosine Transform (IDCT) 18
3.2.1 Distributed Arithmetic DCT 19
3.2.2 Loeffler Algorithm 20
3.3 YCC to RGB Colour Space Conversion 22
3.3.1 Look Up Table 24
3.3.2 Shift and Add 25
3.3.3 Integer Based Transform 26
3.4 Summary 27

4 Proposed Hybrid Architecture 28

4.1 Development Board Specs 29
4.2 Hardware Software Communication 30
4.2.1 Communication through DDR2 RAM 31
4.2.2 Communication through Hardware Registers 32
4.3 Hardware Design 33
4.3.1 IDCT 33
4.3.2 Colour Conversion 36

4.4 Software Design 38

4.4.1 IDCT 38

4.4.2 Colour Conversion 40

4.5 Summary 41

5 Results **43**

5.1 Testing Setup and Procedure 43

5.2 Timing Results 46

5.2.1 Hardware Timing 49

5.3 Image Verification 50

5.4 Summary 55

6 Conclusion and Recommendations **56**

6.1 Recommendations 57

References **59**

A Source Images **61**

B Testing Scripts **63**

C C Code **66**

C.1 libjpeg modifications 66

C.2 ReadImage.c 74

D VHDL Code **80**

D.1 2D IDCT 80

D.2 Colour Converter 95

E Huffman Decoding Example 105

E.1 Source Image 105

E.2 Huffman Table Extraction 108

E.3 Image Decoding 109

E.3.1 Block 1 - Luminance 109

E.3.2 Block 1 - Chrominance 110

E.3.3 Block 2 - Luminance 111

E.3.4 Block 2 - Chrominance 112

E.4 Finalizing 113

E.5 Huffman Tables 114

Vita Auctoris 119

List of Figures

2.1 JPEG Compression Process [13] 7

2.2 JPEG Decompression Process [13] 7

2.3 Energy Compaction of the DCT 9

2.4 Zig Zag order of a DCT block [13] 9

2.5 Huffman Tree 13

3.1 Desktop JPEG Profiling 18

3.2 Loeffler Algorithm for 1D DCT 21

3.3 Loeffler Algorithm for 1D IDCT 22

3.4 Cb-Cr colour plane at a constant luma value 23

4.1 Layers of a GNU/Linux based embedded system 31

4.2 Dequantization and 2D-IDCT Hardware Design 35

4.3 YCC to RGB Hardware Design 38

4.4 IDCT array-register data compaction 39

4.5 3D image array 40

4.6 Colour Conversion array-register data compaction 41

5.1 Image Decompression Timing 48

5.2 Image Decompression Timing 49

5.3 Bookstore image byte offset histogram 51

5.4 Dog image byte offset histogram 52

5.5 beach image byte offset histogram 53

A.1 Sample 20MP images 62

E.1 Sample image 106

E.2 HEX dump from sample image 106

E.3 DCT Frequency domain of an 8x8 JPEG block 107

List of Tables

2.1 Example Huffman data 12

3.1 Butterfly[B], Rotator[R], and constant Multiplication Blocks for Loeffler's Algorithm 21

3.2 Colour Space Look Up Table example 24

4.1 YCC - RGB constants 37

5.1 Test Image Dimensions 46

5.2 Bookstore Image Results 47

5.3 Dog Image Results 47

5.4 Beach Image Results 47

5.5 Hardware Only Timing per test image 50

5.6 Test Image Verification 54

E.1 Huffman Decoding Results 113

E.2 Huffman Luminance (Y) DC table 114

E.3 Huffman Luminance (Y) AC table 115

E.4 Huffman Chrominance (Cb and Cr) DC table 116

E.5 Huffman Chrominance (Cb and Cr) AC table 117

E.6 Huffman DC Value Encoding 118

List of Abbreviations

1D	1 Dimensional
2D	2 Dimensional
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DSP	Digital Signal Processing
FDCT	Forward Discrete Cosine Transform
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HW	Hardware
IDCT	Inverse Discrete Cosine Transform
JPEG	Joint Photographic Experts Group
LUT	Lookup Table
MP	Megapixel
px	pixels
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SOC	System On Chip
SW	Software
RGB	Red Green Blue
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
YCC	Luminance Chrominance(Cb) Chrominance(Cr)

Chapter 1

Introduction

In recent years digital photography has taken leaps and bounds in the consumer market. Along with improved sensors, functions, and touch screens, the pixel count of the images are steadily increasing. At the time of writing this thesis, a consumer can purchase a digital camera capable of taking a JPEG image of, 20 million pixels (20 MP). Additionally there has been an explosion with mobile devices such as smart phones and tablets. The trend that the number of transistors in integrated circuits doubles approximately every 18-24 months is known as Moore's law [9]. Following this trend, mobile processors are approaching the speed and multi-core architectures of their desktop counterparts. A pocket sized computer has its advantages, but it does face challenges when viewing the growing size of multi-media, especially digital images.

By far the most popular form digital image in computing today is the JPEG (Joint Photographic Experts Group) image. JPEG images are found in every digital camera, photo editing suites, MPEG video, internet browsers, and video games to name a few. As all of the photo albums, and every photo taken become digitized into JPEGs, being able to comfortably view the images becomes necessary.

1.1 History of JPEG Images

Early in computing history a need for viewing digital images arose. However, with limited memory and storage space, raw image data was not a feasible solution. In 1986 a committee, the Joint Photographic Experts Group (JPEG) [4], was formed to create a standard for digitized images. The JPEG group worked to standardize a method of coding still pictures, known as JPEGs. The JPEG standard outlines a codec that defines how an image is compressed into a stream of bytes and decompressed back into an image.

The JPEG standard was first publicly released in 1992, which was approved as ITU-T Recommendation T.81 and in 1994 as ISO/IEC 10918-1. Two years later the standard was updated to include rules and checks for software conformance. Several parts of the standard have been added in the years since, which introduce features such as the JPEG File Interchange Format(JFIF), that outlines the structure of the raw data in the file.

The focus of the JPEG codec is to preserve as much data as possible while compressing it into a much smaller file size. Due to this fact, JPEG images are considered a lossy compression. By default they have a high degree of compression, with minimal perceptible loss in image quality but this can be adjusted. Adjustment of the compressions ratio results in a tradeoff between data losses and file size. Higher compression ratio have smaller file size, and vice versa. The JPEG standard provides a framework for image compression and decompression, but depending on the algorithms used determines the different types of JPEGs. A few types are progressive sequential, lossless and baseline sequential. By far the most common type is Baseline Sequential. The standard provides a simple and efficient algorithm which makes it suitable for all digital cameras.

1.2 Overview of Research, Motivation

The accelerating popularity of mobile devices, and exploding pixel counts found in consumer grade digital cameras have lead to an issue when attempting to view ones own images. Many software applications have employed techniques when displaying larger images, such as pre-computing display sized thumbnails. However precomputation of a digital photo album does not resolve the problem, nor are these techniques future proof against large device screen sizes, resolution, and increasing pixel count.

The focus of this thesis is to present an alternative architecture to relieve computations from the CPU and therefore improve upon the delay mobile devices undergo when accessing large JPEG images. Currently there are dedicated hardware modules for audio and video playback, but nothing has been done for still images. This work is implemented on an embedded FPGA Linux platform using a JPEG decoding software library found in the majority of mobile devices.

This thesis demonstrates the capability of dedicated hardware to assist the CPU in computational workloads. It primarily focuses on reducing delays from the mathematically challenging two-dimensional inverse discrete cosine transform (IDCT), and the colour conversion from the luminance, and chrominance channels to the red, green, blue colour system frequently used in digital displays. It achieves improvements from hardware and software optimizations.

1.3 Organization of Thesis

Chapter 1 begins with an introduction to JPEG images and the JPEG ISO Standard, then continues with a brief introduction to digital photography and trends in the mobile market. Chapter 2 elaborates on the JPEG standard, describing the necessary steps used to compress and decompress raw data to and from a viewable image. Chapter 3 discusses previous works which aid in specific algorithms used by the JPEG standard. Chapter 4 proposes a new software-hardware hybrid architecture used to improve upon the JPEG decoding time. Chapter 5 presents the testing methodology, timing results, and verification of the hybrid architecture. Chapter 6 concludes this thesis by making recommendations on future improvements for this work.

Chapter 2

The JPEG Standard

The JPEG standard for compression of still images [13], outlines the processes which must be utilized to complete the JPEG compression or decompression. The standard is comprised of 3 primary transformations which are reversible. In this chapter, the methods and guidelines of the JPEG standard are reviewed, followed by a brief introduction to Field Programmable Gate Arrays (FPGAs) and embedded systems.

The JPEG compression process consists of the Forward Discrete Cosine Transform(FDCT), quantization of the DCT coefficients, and encoding the remaining values into binary representations as shown in Figure 2.1.

The JPEG decompression process consists of the same components as compres-

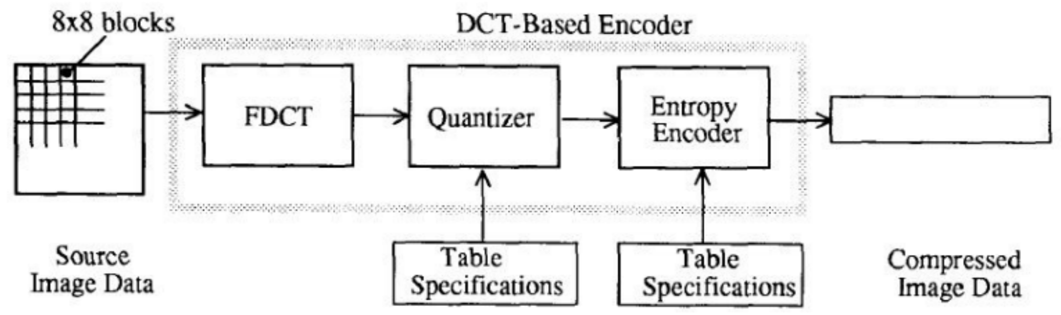


Figure 2.1: JPEG Compression Process [13]

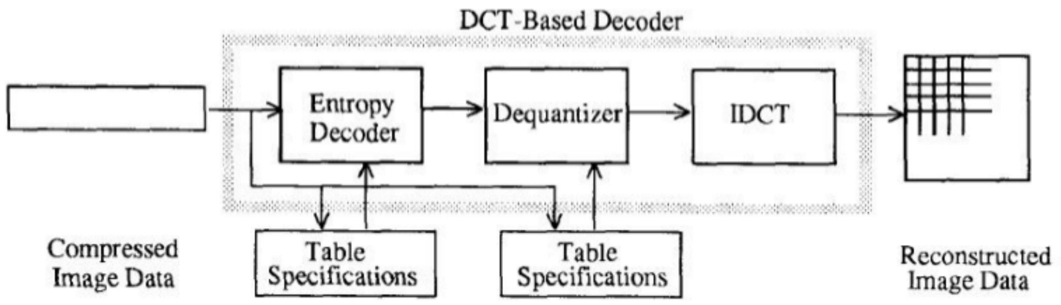


Figure 2.2: JPEG Decompression Process [13]

sion in the reverse and inverse order. Decoding the binary values, dequantization, and the Inverse Discrete Cosine Transform(IDCT) to obtain the original image content as shown in Figure 2.2.

2.1 Discrete Cosine Transform

The Forward Discrete Cosine Transform(FDCT) is commonly referred to as the Discrete Cosine Transform (DCT). It is a mathematical transform that achieves a high degree of compression with minimal losses. Equation 2.1 shows the 1 dimensional DCT (1D-DCT) of length 8. In JPEG image processing, Figure 2.1, the DCT is performed 2 dimensionally (2D-DCT) on 8x8 blocks as shown in Equation 2.2, in which the 1D-DCT is applied to each row, followed by each column.

$$\begin{aligned} F(k) &= \alpha(k) \sum_{x=0}^{7} f(x) \cos(\frac{(2x+1)k\pi}{16}) \\ \alpha(k) &= \sqrt{\frac{1}{8}}\, for\ k = 0 \\ \alpha(k) &= \frac{1}{2}\, otherwise \end{aligned} \tag{2.1}$$

$$\begin{aligned} F(u,v) &= \frac{1}{4} C(u) C(v) \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos(\frac{(2x+1)u\pi}{16}) \cos(\frac{(2y+1)v\pi}{16}) \\ C(u), C(v) &= \frac{1}{\sqrt{2}}\, for\ u, v = 0 \\ C(u), C(v) &= 1\ otherwise \end{aligned} \tag{2.2}$$

As a result the output of the 2D-DCT focuses its energy in the upper left corner of the 8x8 block, causing most of the 8x8 block to be zero. Figure 2.3 illustrates the energy compaction of one row of pixels. Encoding of the DCT block can then be optimized with the zig-zag pattern in Figure 2.4. Thus the trailing zeros can further assist in the high compression in JPEG images.

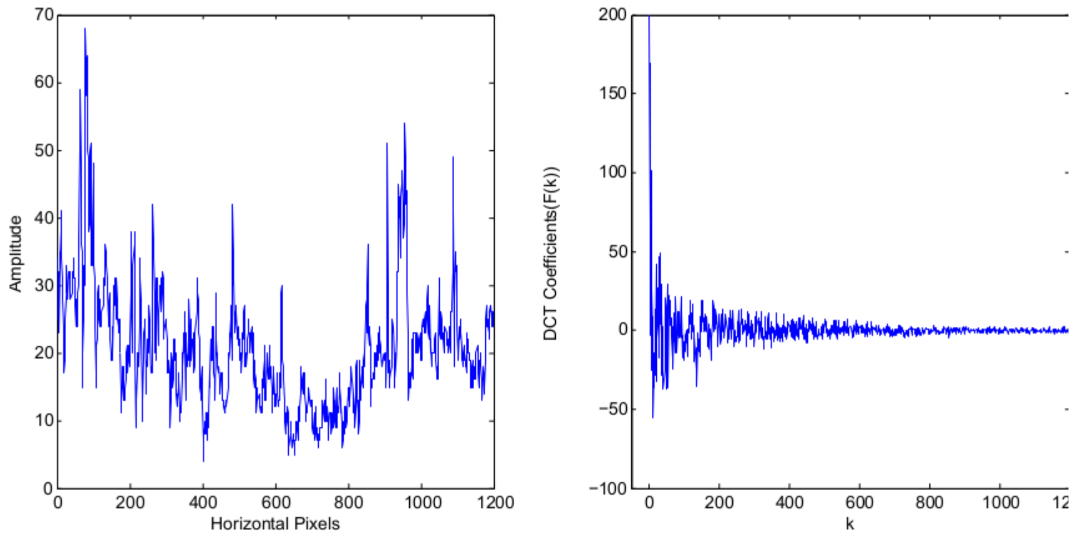


Figure 2.3: Energy Compaction of the DCT

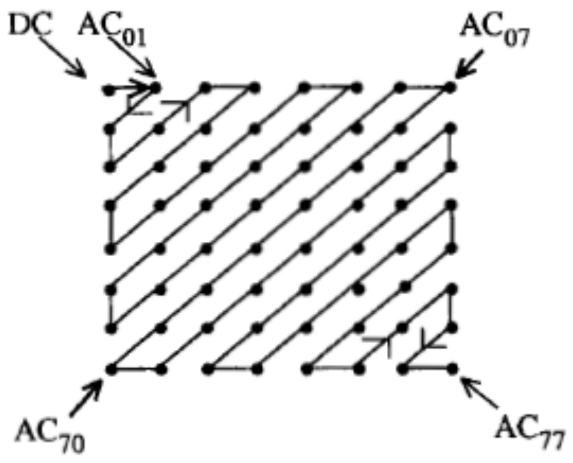


Figure 2.4: Zig Zag order of a DCT block [13]

In the JPEG decompression, Figure 2.2, process the Inverse Discrete Cosine Transform (IDCT) is utilized to transform the dequantized Huffman coefficients back into useable image data. Equation 2.3 shows the 2D-IDCT, which is the inverse of the

2D-DCT. Similar to the DCT, the IDCT is a row-column transformation.

$$\begin{aligned}
f(x,y) &= \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos(\frac{(2x+1)u\pi}{16})\cos(\frac{(2y+1)v\pi}{16}) \\
C(u),C(v) &= \frac{1}{\sqrt{2}}\; for\; u,v=0 \\
C(u),C(v) &= 1\; otherwise
\end{aligned} \tag{2.3}$$

2.2 Quantization

The quantization transformation shifts the output of the DCT down with integer rounded division to increase the compression shown in Equation 2.4. Smaller numbers use fewer bits which achieves higher compression. The factor at which the DCT coefficients are divided is determined by the quantization matrix. The quantization matrix is a statistically determined matrix where the higher valued coefficients at the top left of the matrix receive more scaling than the rest. This is due to the fact that the output of the DCT focuses its energy at this corner. In most JPEGs the quantization matrices are not statistically determined for that particular image, instead a generic set of tables based on the human vision system and trends from the DCT are used.

$$F^Q(u,v) = IntegerRound(\frac{F(u,v)}{Q(u,v)}) \tag{2.4}$$

Dequantization is the reverse of quantization where the inputs to the IDCT are multiplied by the same quantization matrix to scale up the coefficients to the original intended value as shown in Equation 2.5.

$$F^{Q'}(u,v) = F^Q(u,v) * Q(u,v) \tag{2.5}$$

2.3 Huffman Encoding and Decoding

The final component of the JPEG compression/decompression process is entropy encoding/decoding. Encoding is the process of converting data from one format to another for the purposes of speed, security or space saving. The purpose of entropy encoding/decoding used in JPEG images is to compress the data without any losses. The most common type of JPEG image is baseline sequential, which use Huffman encoding/decoding [6]. Huffman encoding achieves lossless compression by allocating the most frequently used symbols with the fewest number of bits. The example data set in Table 2.1 illustrates how this process functions.

Symbol	**Frequency**	**Code**	**Code Length**
A	24	0	1
B	12	100	3
C	10	101	3
D	8	110	3
E	8	111	3

Table 2.1: Example Huffman data

Based on the frequency of the symbol, a Huffman tree in Figure 2.5 is constructed. Most frequently used symbols are near the top of the tree, and the less frequent are at the bottom. A ”move” down the right branch of the tree represents a ”1”, and a ”move” down the left branch represents a ”0”

Eg: Symbol ’C’ is allocated the bits ”101”, while the most common symbol ’A’ uses only 2 bits.

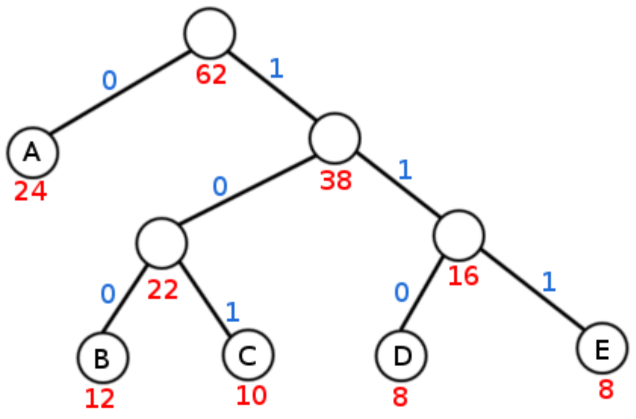


Figure 2.5: Huffman Tree

Using this method the Huffman encoding creates a much smaller encoded bit stream. Instead of building a Huffman tree each time, the JPEG image includes a set of tables giving the bit to number translations. For decoding a set of 4 tables are created. The tables are categorized by the colour channel, and frequency type: Luminance DC, Luminance AC, Chrominance DC, and Chrominance AC. The two Chrominance channels(Cb, Cr) share the AC and DC tables.

Huffman decoding is the reverse process of encoding. With the provided tables and compressed bitstream the values can be perfectly reconstructed during JPEG decompression. In many digital imaging devices such as digital cameras, the Huffman symbol calculations for that image are not calculated and a set of statistically determined tables are used instead. Utilizing the default set of tables allows for faster image compression, but can create a larger file size. In programs such as image editors, these settings can be changed to calculate the Huffman tables for a given image, which result in a smaller file size, but more computation time.

2.4 Field Programmable Gate Array

A Field Programmable Gate Array (FPGA) is an integrated circuit designed to be configurable after manufacture. Using design tools, an FPGA can be programmed to generate the logic for virtually any hardware configuration. Unlike common CPU architectures, since FPGAs are effectively programmable hardware it has the capability to perform computations in parallel. This attribute greatly enhances speed over the conventional methodology.

Many embedded systems, specifically those in mobile devices consist of a complete system on chip (SOC). Additional hardware such as peripherals, are added either internally or externally. The use of FPGAs provide a flexible development platform without the cost of a conventional SOC design. This work utilizes an FPGA which models a SOC and its peripherals. The FPGA platform makes this work a suitable platform and allows testing in real time.

2.4.1 Soft Processor

A soft processor is a microprocessor core that can be entirely implemented in using logic synthesis. For FPGAs a soft processor design can mimic the Reduced Instruction Set Computer(RISC) architectures found in many embedded systems. The MicroBlaze processor [1] used in this work is a soft processor designed for Xilinx FPGAs. The MicroBlaze is implemented entirely in the general-purpose memory and logic fabric of Xilinx FPGAs.

2.5 Summary

This chapter discussed the methods outlined in the JPEG standard used to compress and decompress JPEG image data, as well as gave an introduction to FPGAs in embedded systems.

Due to the fact that the JPEG is highly complex, this research focuses on optimizing individual components of the JPEG decompression process rather than the entire algorithm. Utilizing the configurability of FPGAs a hardware architecture can be designed to improve the software functions. This hybrid software-hardware architecture will be a beneficial replacement for the conventional software only approach.

Chapter 3

Existing Work in Hardware JPEG Algorithms

The JPEG ISO standard, described in Chapter 2, outlines the transforms that are needed in order to compress and decompress JPEG images. It does not describe the specific algorithms that need to be used to achieve the end product. There hasn't been much publicized work on the complete decompression of a JPEG, but there has been some work in the individual algorithms used to decompress a JPEG image. This chapter will begin with an introduction to libjpeg, the most commonly used JPEG software library used in mobile systems, followed by the state of the art in the algorithms applicable to JPEG images, as well as their advantages and disadvantages.

3.1 libjpeg

There has been much work done in the software compression and decompression of JPEG images. In 1986, shortly after the JPEG standard was created a group called the Independent JPEG Group [7] (IJG) created the free open-source C library called ”libjpeg”. The library is multi-platform configurable, and follows all of the JPEG ISO standards, and as such libjpeg is credited by the JPEG group to being a reliable source to use in software applications. Due to its longevity and popularity libjpeg has found itself as the primary JPEG encoder/decoder in mobile device applications, as well as desktop applications. For this reason, this work uses the libjpeg ported for the Xilinx MicroBlaze architecture as the starting point and control for testing.

Chapter 2 outlined the three primary steps used in compression and decompression, however these algorithms are not necessarily the most costly functions in the digital domain. Therefore, the libjpeg library was profiled (a process of timing the functions in software to find where the most significant bottlenecks reside) and examined.

To acquire an approximate function benchmark a 20 MP image was profiled on a desktop PC running a Pentium-4 3GHz processor, 32-bit GNU/Linux operating system, and 2GB of RAM. The image was profiled 100 times and averaged to achieve statistical accuracy. Figure 3.1 displays the profiling results.

The profiling results show that the most expensive functions are the IDCT, the

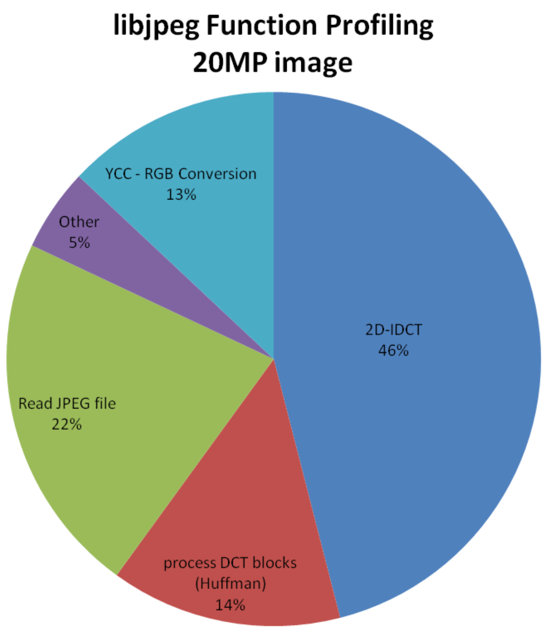


Figure 3.1: Desktop JPEG Profiling

colour space conversion, and the Huffman transform with 46%, 13%, and 14% respectively. The focus of this work centres on optimizing the IDCT and colour conversion to enhance the JPEG decompression time.

3.2 Inverse Discrete Cosine Transform (IDCT)

First introduced in Section 2.1 the IDCT is the Inverse of the Discrete Cosine Transform which used to transform the output of raw dequantized Huffman coefficients. For JPEG images the pixels are split into 8x8 blocks prior to applying the DCT. The 2D-IDCT consists of 16 1D-IDCTs over the 8 rows and 8 columns of the DCT blocks. Equation 3.1 demonstrates the formal definition of the 2D-IDCT as it applies to the

8x8 JPEG blocks.

$$\begin{aligned} f(x,y) &= \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos(\frac{(2x+1)u\pi}{16})\cos(\frac{(2y+1)v\pi}{16}) \\ C(u),C(v) &= \frac{1}{\sqrt{2}}\ for\ u,v=0 \\ C(u),C(v) &= 1\ otherwise \end{aligned} \tag{3.1}$$

The traditional IDCT in equation 3.1 shows the iterative multiplication process.

3.2.1 Distributed Arithmetic DCT

The Distributed Arithmetic DCT (DA-DCT) proposed by Pan [10] attempts to reduce the complexity of the DCT by exploiting the binary representations of the 2D-DCT matrix. The number of additions in this transform are reduced by a factor of 22. On average 1 multiplication, 40 additions, and 16 binary shifts are required for each DCT coefficient. Equations 3.2 - 3.6 illustrate how the transform is performed.

$$Y = A * X = \begin{pmatrix} A_1 & A_2 & \cdots & A_L \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_L \end{pmatrix} \tag{3.2}$$

$$A_k = -A_k^M 2^M + \sum_{i=N}^{M-1} A_k^i 2^i \tag{3.3}$$

$$\text{where } A_k^i \text{ is 0 or 1}$$

$$Y = \underbrace{2^N * \begin{pmatrix} -2^{M-N} & 2^{M-N-1} & \cdots & 2 & 1 \end{pmatrix}}_{\text{S}} * \underbrace{\begin{pmatrix} A_1^M & A_2^M & \cdots & A_L^M \\ A_1^{M-1} & A_2^{M-1} & \cdots & A_L^{M-1} \\ \vdots & \vdots & \cdots & \vdots \\ A_1^N & A_2^N & \cdots & A_L^N \end{pmatrix}}_{\text{B}} * \underbrace{\begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_L \end{pmatrix}}_{\text{X}} \tag{3.4}$$

$$\text{Let } C = B * X = \begin{pmatrix} C_M & C_{M-1} & \cdots & C_N \end{pmatrix}^T \tag{3.5}$$

$$Y = S * C \tag{3.6}$$

The multiplicative complexity is significantly reduced at the cost of additions and shift operations, however; the DA-DCT word length must be ≥ 9 in order to be considered indistinguishable from floating point, and it must be ≥ 16 to be numerically equivalent.

3.2.2 Loeffler Algorithm

By far the most popular algorithm for the DCT is the The Loeffler Algorithm [8] used in [12]. The Loeffler Algorithm takes a similar approach to the DCT as the Fast Fourier Transform (FFT) takes to the Discrete Fourier Transform(DFT). The Algorithm is mapped to a set of "Butterfly" and "Rotator" blocks. It reduces the DCT down to 29 additions and 11 multiplications, shown in Figure 3.2 and Table 3.1.

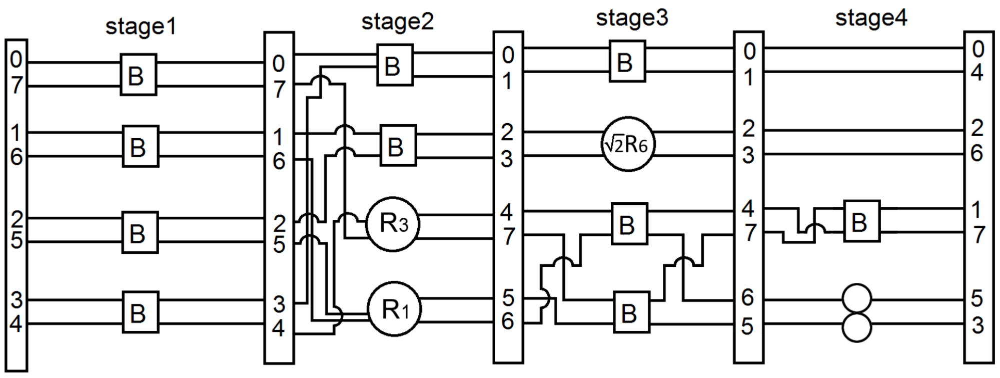


Figure 3.2: Loeffler Algorithm for 1D DCT

where

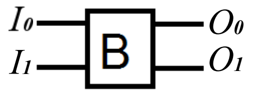
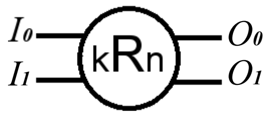
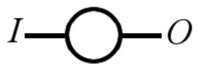
diagram	**equation**
	$O_0 = I_0 + I_1$
	$O_1 = I_0 - I_1$
	$O_0 = I_0 * k * \cos(\frac{n\pi}{16}) + I_1 * k * \sin(\frac{n\pi}{16})$
	$O_1 = I_1 * k * \cos(\frac{n\pi}{16}) - I_0 * k * \sin(\frac{n\pi}{16})$
	$O = I * \sqrt{\frac{1}{2}}$

Table 3.1: Butterfly[B], Rotator[R], and constant Multiplication Blocks for Loeffler's Algorithm

Due to the reversibility of the DCT, the IDCT Loeffler algorithm is mapped out as in Figure 3.3.

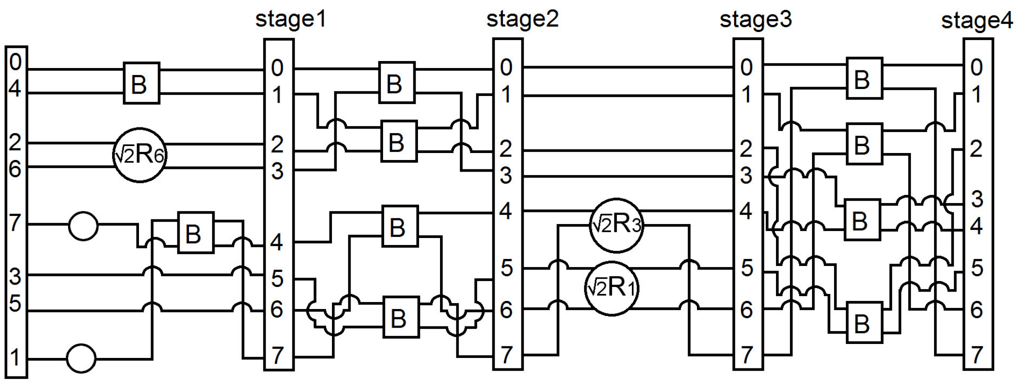


Figure 3.3: Loeffler Algorithm for 1D IDCT

Since the complexity of the algorithm significantly reduces the complexity of the IDCT to fewer multiplications and additions, it makes this algorithm the most suitable method for the JPEG process. Consequently, this is the current IDCT algorithm implemented in the libjpeg library.

3.3 YCC to RGB Colour Space Conversion

The colour scheme of the JPEG standard is based on the Luminance and Chrominance channels (YCC). The YCC colour scheme has minimal redundancy, unlike RGB, which makes it suitable for JPEG images. ”Y” is the Luminance component and ”Cb” and ”Cr” are the blue-difference, and red-difference chroma components. Figure 3.4 illustrates the chroma colour plane at a constant luma value.

Most displays use the Red-Green-Blue (RGB) colour scheme, so conversion between the two colour schemes is necessary. The colour space conversion is a simple process, Equation 3.7, but is an intensive operation due to the fact that each pixel

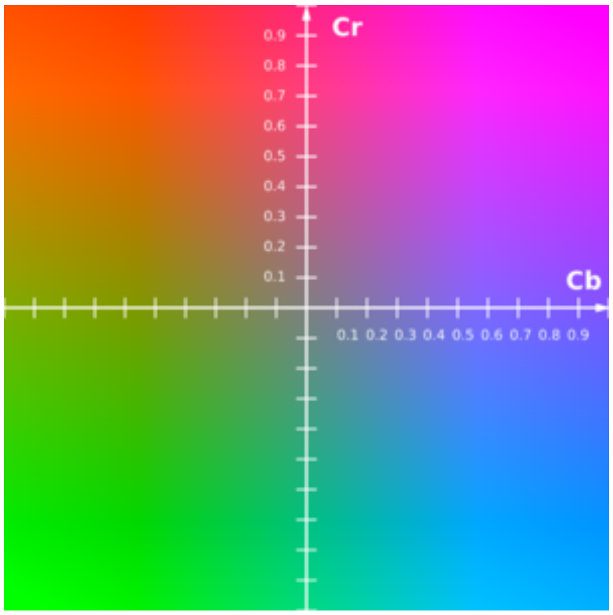


Figure 3.4: Cb-Cr colour plane at a constant luma value

has 3 colour channels, which have 4 multiplications and 4 additions. For example, a 20MP image has 60 million individual colour channels, which possibly has 80 million multiplications, and 80 million additions.

$$R = Y + 1.402 * (Cr - 128) \tag{3.7a}$$

$$B = Y + 1.772 * (Cb - 128) \tag{3.7b}$$

$$G = Y - 0.34414 * (Cb - 128) - 0.71414 * (Cr - 128) \tag{3.7c}$$

3.3.1 Look Up Table

The Look-up Table (LUT) method is one of the most efficient methods, especially for embedded systems, for computing a large number of calculations for limited multiplications. Since the colour conversion utilizes constant multiplications, and the chrominance colour channels are limited to 256 values many systems pre-compute every possible value. These values are stored in a table to be referenced when needed. Typical YCbCr LUT implementation calculates the 4 base multiplications, Cr * 1.402, Cr * 0.71414, Cb * 1.772 and Cb * 0.34414, and stores the values in tables. When calculation is needed the Cr and Cb values are used as indices in these tables, and additions are the only operation to calculate.

index	Cr-Red (Cr * 1.402)	Cr-Green (Cr * 0.71414)	Cb-Blue (Cb * 1.772)	Cb-Green (Cb * 0.34414)
1	1.402	0.714	1.772	0.344
2	2.804	1.428	3.544	0.688
⋮	⋮	⋮	⋮	⋮
255	357.51	182.106	451.86	87.756

Table 3.2: Colour Space Look Up Table example

libjpeg uses LUTs for the YCC to RGB conversion. The LUTs are computed once and stored upon startup. This allows a table look up and addition to calculate the colour conversion. The advantage of this method is speed, but the disadvantage is accuracy.

Indexing and accessing large tables occupies time, but the amount of time isn't necessarily consistent. Data caching problems, such as cache misses can arise with these large data sets. Most CPU caches use an n-way associative cache, which can cause replacement issues with using many table lookups, found in JPEG colour conversion. This can result in more cache misses and slower time, which is not ideal for this work.

3.3.2 Shift and Add

To reduce on the cumbersome floating point multiplications needed for colour space conversion Yang et al [14] proposed a Fast Algorithm for YCbCr to RGB. On Digital Signal Processors (DSPs), floating point multiplications, multiplications by a decimal number, are simplified with multiple shift and add operations. The more unique the decimal value of the multiplicand, the more shifts and adds are required, consuming processor resources and time. Colour conversion algorithms contain unique decimal values, and are typically carried out thousands to millions of times. Yang et al proposed to reduce the complexity of the multiplications in colour conversion with minimal quantification error while maintaining the image quality.

Their algorithm selects the most significant fixed-point shift and add operations to take place of the floating point multiplication. As such, the processor has fewer operations to compute and thus time consumption is reduced.

$$R = Y + Cr + Cr \gg 2 + Cr \gg 3 + Cr \gg 5 \tag{3.8a}$$

$$B = Y + Cb \gg 1 + Cb \gg 2 + Cb \gg 6 \tag{3.8b}$$

$$G = Y - (Cb \gg 2 + Cb \gg 4 + Cb \gg 5) - (Cr \gg 1 + Cr \gg 3 + Cr \gg 4 + Cr \gg 5) \tag{3.8c}$$

The YCC to RGB conversion in Equation 3.8 proposed were tested with video sequences comparing against floating point method. The maximum error observed was 0.02% of all the images in the video. This method is ideal for YCC to RGB approximation in JPEG applications, but the error in the colour may increase when data propagates through the DCT.

3.3.3 Integer Based Transform

Pei et al [11] proposed a Reversible Integer Color Transform With Bit-Constraints to optimize colour transforms. This method looks at redesigning the input coefficients to floating point calculations completely. As a result the colour conversion transform will be lossless, and can be implemented with only fixed-point operations.

The transformation process relies on the basis that most colour transformations consist of 3x3 matrices. YCbCr, RGB, Karhunen-Loeve Average (KLA), and Intensity Hue Saturation (IHS) are all examples of 3x3 matrix colour schemes. The process to generate the integer based transform involves normalizing the input matrix and extracting the binary valued matrix which depends on 8 fixed point-values that approximate floating-point values. After the transformation is generated, using the integer transform in forward and reverse applications becomes a simple set of matrix operations.

Pei et al, tested this transform for RGB to YCbCr achieving 0.288% normalized root mean square error. The advantage of this method eliminates all floating-point operations, but the pre-computation and encoding of the colours this way is necessary, and not applicable for JPEG applications.

3.4 Summary

This chapter presented several existing techniques that were developed to optimize and accelerate the individual components of the JPEG process. As discussed some of these techniques are more applicable to JPEG decompression than others. The Loeffler algorithm, colour conversion LUTs, and the shift and add methods for YCC to RGB conversion are the most advantageous procedures for JPEG implementation.

In the work presented herein, a novel hybrid architecture between software and hardware was designed to provide hardware acceleration of large JPEG images.

Chapter 4

Proposed Hybrid Architecture

This chapter illustrates the proposed hybrid software-hardware architecture used to accelerate the time it takes to decompress a JPEG image with libjpeg. This hybrid approach is applied to the IDCT and the colour conversion components of the JPEG decompression scheme. This chapter describes the architecture and the reasoning in each domain; hardware and software.

In the proposed design, the IDCT and colour conversion were implemented in hardware to aid the software library in its most expensive operations. This novel design incorporates the advantages of using parallelism in hardware along with a few software memory techniques to expedite the JPEG decompression time.

4.1 Development Board Specs

In order to emulate the conditions of a real world mobile device this work used an embedded GNU/Linux variant on an FPGA. The FPGA development board used was a Xilinx XUPV5-LX110T development board [3], which has the following specifications:

- Xilinx Vertex®-5 XC5VLX110T FPGA
- 64-bit wide 256MB DDR2 small outline DIMM(SODIMM) module
- On-board 32-bit ZBT synchronous SRAM and Intel P30 StrataFlash
- 10/100/1000 tri-speed Ethernet PHY
- Programmable system clock generator
- RS-232 port
- JTAG programming interface

For this proposed architecture the development board is programmed with a 125 MHz MicroBlaze soft-core processor. The operating system is the GNU/Linux variant, Petalinux [2], which is targeted at FPGA-based embedded MicroBlaze and PowerPC systems. Petalinux allows C /C++ application development and debugging using its provided compilers. Included in the Petalinux kernel is a host of standard libraries and applications, one of which is a recent version of libjpeg.

4.2 Hardware Software Communication

The desired goal of the hybrid architecture is to optimize software by moving the expensive operations to dedicated hardware, but the communication link between hardware and software is a vital component in and of itself. There are a few methods available for interaction between the software and the hardware, each with their own pros and cons.

With respect to memory in a memory managed Linux kernel, the privileges can be envisioned as a set of layers as in Figure 4.1. The top most layer is the user-space. This layer is what the typical user interacts with, here the compiled applications are executed, and present the input and output. In this layer there exists a dynamic amount of memory available, which are virtually indexed by the kernel. This protection prevents the user from corrupting any valuable memory locations. The next layer down is the kernel. The kernel is the memory limited control centre. Device drivers exist in the kernel which communicate between the user-space and kernel, and between kernel and the physical memory or hardware. The bottom-most layer is the hardware. The hardware is the physical aspect such as DDR2 RAM, hardware registers, ports, and logic elements.

The kernel protection layer becomes an added bottleneck for the hardware acceleration of software, especially for multiple transactions. As a result the careful consideration of this communication is taken into account. There are two primary methods of inter-system interaction; through DDR2 RAM, or through hardware mapped registers, each of which utilize a memory controller driver found in "/dev/mem".

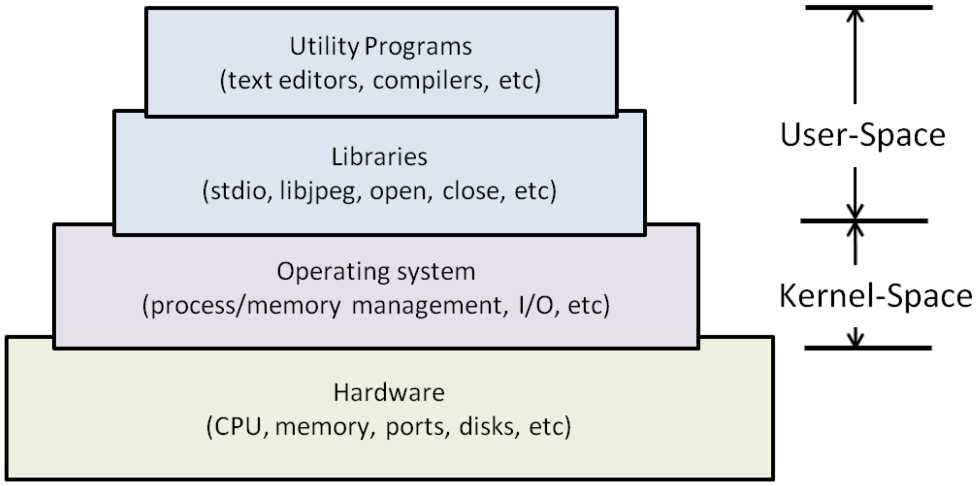

Utility Programs
(text editors, compilers, etc)
Libraries
(stdio, libjpeg, open, close, etc)
Operating system
(process/memory management, I/O, etc)
Hardware
(CPU, memory, ports, disks, etc)
User-Space
Kernel-Space

Figure 4.1: Layers of a GNU/Linux based embedded system

4.2.1 Communication through DDR2 RAM

The first method for software-hardware communication can be done through the system's available DDR2 RAM. The user-space application requests access to a sector of RAM from the memory driver, and receives a pointer that can be used to read and write to the RAM.

On the hardware side, the logic must provide the control signals at the correct timing to read and write to the RAM module. The control signals must be accurate to transmit the data correctly. Unlike software, the hardware has full control over how much data can be transmitted in one transaction. This advantage is through what is called Burst reading and writing. For this transaction a multi-port memory controller(MPMC) is available to the MicroBlaze processor. The MPMC supports Burst reads and writes up to 32 32-bit words per transaction. This greatly speeds up memory access time.

Utilizing Burst reading and writing in a design is beneficial, however there is too much data copying, which is too costly. Data is copied from software to RAM, from RAM to hardware logic elements to operate on, and then back.

4.2.2 Communication through Hardware Registers

The alternative method for communication is directly through the hardwares memory mapped registers, eliminating the need for additional RAM accesses. For software, the data transfer is similar to the process with RAM. The software reads and writes to a pointer allocated to it by the /dev/mem driver.

On the hardware side, the register which has been written to by software can be directly operated on without needing any additional copying. Additionally, the hardware has the ability to react immediately after a register has been written. This provides an ”interrupt on change” type of interface eliminating the need for a ”start” command.

Utilizing direct communication through registers is limited by the resources and the property that these software accessible registers cannot be bidirectional, however there is far less data copying than communication through RAM.

There are pros and cons for each method, but direct communication through registers provides faster throughput. The register only process is simpler to use for the hardware, faster, less copying of data, and it allows for concurrent processing between software and hardware. Concurrent processing is achieved when the software is writing to register ”n” while the hardware operates on registers ”n-1”, ”n-2”, ..., etc.

4.3 Hardware Design

In this work the software layout must be considered when designing the hardware for the 2D-IDCT and colour conversion. The most optimal hardware designs relieve as much processing from the Software as possible. The input and output data structures on the software level must remain the same, and any data manipulation must be done on the hardware level.

4.3.1 IDCT

The 2D-IDCT hardware design relieves the software from three operations found in libjpeg; dequantization, the 2D-IDCT, and range-limiting. Dequantization is the multiplication of the IDCT input coefficients by the quantization matrix, scaling up the coefficients back to the original post-DCT value. After dequantization, the 2D-IDCT is carried out. As mentioned in Chapter 2 and 3, the 2D-IDCT transform is separable into its rows and columns. In the JPEG architecture DCT blocks of 8x8 are used. Eight row 1D-IDCTs followed by eight column 1D-IDCTs are needed to complete a full 2D-IDCT. The last step, range-limiting is a simple process to suppress out of range corrupt values.

The separability of the rows and columns allows the hardware design of the 2D-IDCT to accept the input matrices, IDCT and quantization matrix, coefficients row-by-row. Upon receipt of an input and quantization row, the hardware IDCT module immediately computes the simple multiplication for the dequantization. The dequantized values are the input to the row 1D-IDCTs. The most efficient IDCT algorithm to date is the modified Loefller Algorithm [8], and is the most suitable for hardware implementation. Since this algorithm not only uses fewer multiplications than the standard IDCT algorithm, the multiplications are also in parallel which is ideal for hardware implementation. Natively libjpeg uses integer based multiplication for its IDCT. Floating point is possible, but is costly. To keep the accuracy found with floating point the integer based IDCT shifts all of its multiplication factors up, performs the calculation, and shifts the product back down. This process truncates its less significant digits, but retains accuracy and speed in integer form. The 1D-IDCT implementation in this hardware design utilizes the shift and multiply technique carried through the entire 1D-IDCT.

Once a complete row has been written by software to the hardware registers, the 1D-IDCTs compute the row IDCTs. After all of the 8 rows have been written to the hardware registers, the hardware module transposes the rows and computes the 8 column IDCTs for a full 2D-IDCT. The final step is range limiting the output. Range limiting shifts and truncates the output to an 8 bit range of 0 to 255, and is the data type libjpeg uses for these values. Data corruption in JPEG compression is possible, so range limiting prevents out of range pixel values from wrapping around to an invalid number. The design layout is illustrated in Figure 4.2.

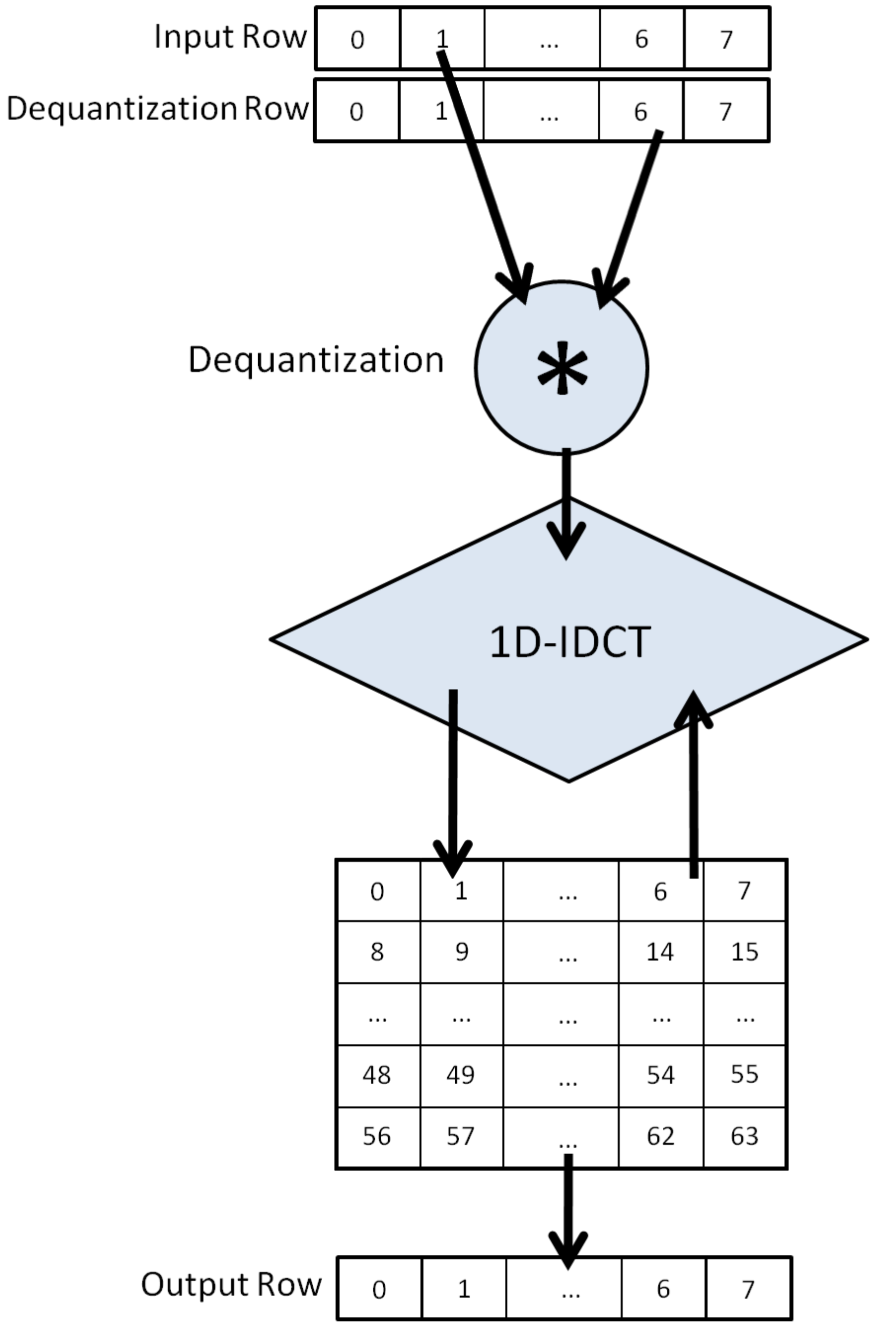


Figure 4.2: Dequantization and 2D-IDCT Hardware Design

4.3.2 Colour Conversion

The colour conversion hardware design in this work relieves the software from three operations found in libjpeg; YCC to RGB table pre-computation, YCC to RGB conversion, and range limiting.

The majority of digital displays use the RGB colour scheme. As mentioned in Chapter 3, libjpeg uses the YCC colour scheme to optimize storage capacity. The conversion from YCC to RGB libjpeg utilizes a set of lookup tables for the Chrominance components; Cr-red, Cb-blue, Cr-green, Cb-green, of each colour channel. LUTs are the most efficient method, but the access speed can be inconsistent due to the possibility of cache misses. As the data set increases, the probability of a cache miss increases as well. As a result, this is not a suitable method for this work.

The hardware design of the colour conversion in this work performs a straight YCC to RGB calculation, eliminating the need for pre-computed table. This is possible because the number of cycles hardware uses to perform a multiplication by the Cr-red, Cb-blue, Cr-green, and Cb-green constants is fewer than the number of cycles used in software to reference the look up tables. The multiplications by the constants in Table 4.1 are performed using fixed point multiplication while retaining the accuracy of floating point multiplication with decimal truncation.

Colour Channel	Decimal Value	Binary Representation
Cr-red	1.402	10110011011101001
Cr-green	0.71414	01011011011010010
Cb-green	0.34414	00101100000011010
Cb-blue	1.772	11100010110100010

Table 4.1: YCC - RGB constants

The last optimization the hardware colour conversion makes is multi pixel conversion. Since each colour channel, Y, Cb, Cr, R, G, and B, are 8 bits each and the software accessible registers are 32 bits the hardware converts 4 pixels at once, filling each register with 4 colour channels. The RGB output range-limited to 8 bit values of 0 to 255 and is rearranged to match the order in libjpeg as shown in Figure 4.3.

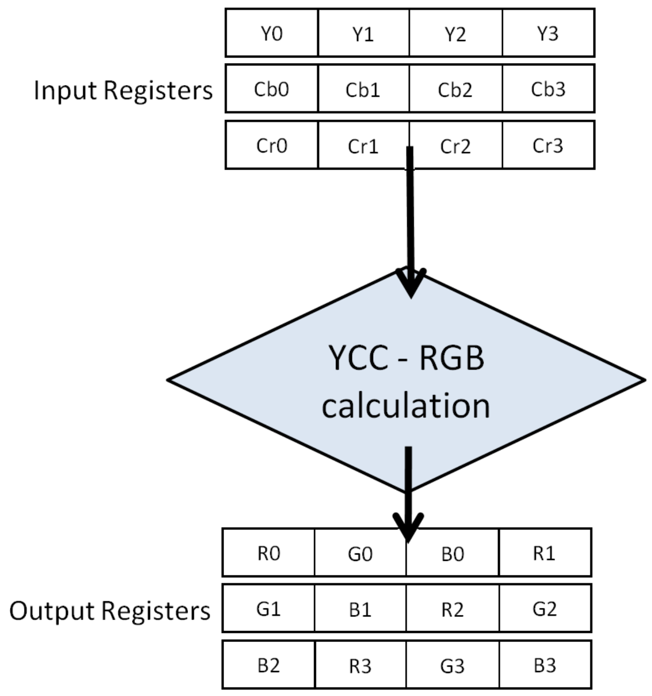


Figure 4.3: YCC to RGB Hardware Design

4.4 Software Design

In this work, libjpeg was modified to utilize the hardware components. Since most of the workload is being transferred into the hardware domain, all data copying was carefully thought out and minimized. Minimizations are performed by maintaining the layout of arrays, pointer manipulation and word size consistency.

4.4.1 IDCT

Libjpeg computes the IDCT using the Modified Loeffler algorithm. Pointers to the input coefficients are passed to the function along with a reference to the quantization matrix. The function dequantizes the coefficients by multiplication, performs the IDCT, and range limits the output to 8 bits. These three processes were moved to hardware.

For this architecture the software IDCT component's functions are limited to writing rows to the hardware registers, and reading rows from the registers. The software writes in the quantization matrix and IDCT coefficients into the hardware registers row by row. The same registers are used because the hardware can manipulate the data faster than the software can change it. To maximize the data transfer from software to hardware the rows are moved into the fewest amount of 32 bit registers as possible. Casting the rows into integer pointers permits using 2 hardware register for 8 entries of the IDCT input, and 4 registers for 8 entries of the quantization matrix. After computation in hardware, the software reads in the output registers row by row from 2 hardware registers. Figure 4.4 illustrates the data compaction.

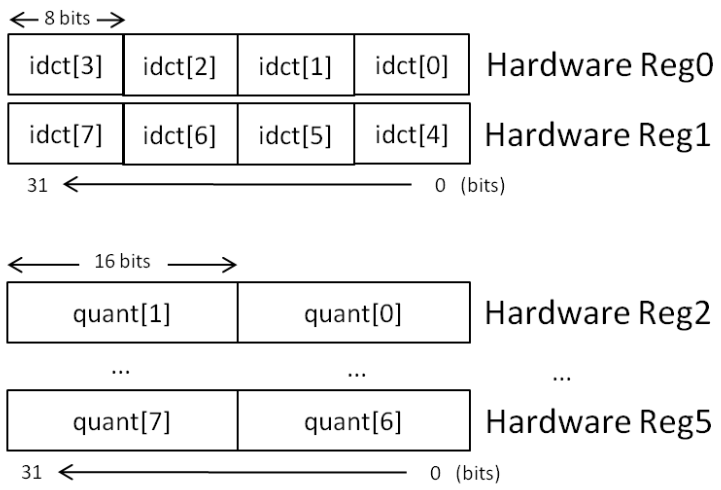


Figure 4.4: IDCT array-register data compaction

4.4.2 Colour Conversion

Upon starting, libjpeg calculates all the possible chrominance combinations for the conversion from YCC to RGB colour space. These calculations are stored in 4 tables: Cr-red, Cb-blue, Cr-green, Cb-green. When called, the YCC-RGB colour conversion function gets passed a 3 dimensional array which represents the deconstructed image in the YCC colour channels. Figure 4.5 shows a visual representation of this image. the x-y coordinates of the image represent the x-y coordinates of the source image, where each layer represents the different colour channels of each pixel.

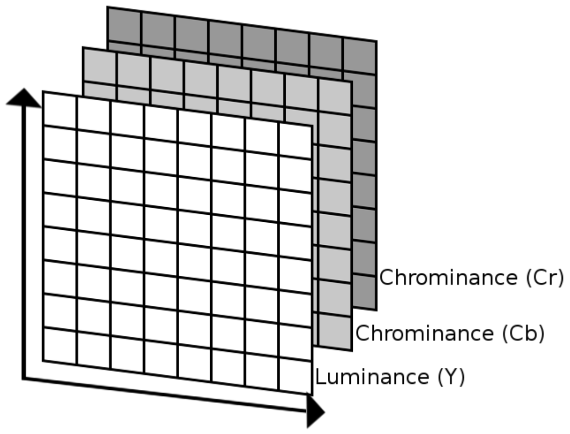


Figure 4.5: 3D image array

Each array of colour channels are passed in as 8 bit unsigned values. Natively, 32-bit CPU architectures function optimally with 32-bit words, additionally the software accessible hardware registers are 32 bits. As a result in this architecture for the colour conversion, a technique is used to cast a 32 bit integer pointer to the array of 8 bit bytes. This allows the software to access 4 values of Y, Cb, or Cr with one variable, and one write to the hardware register. Figure 4.6 illustrates this technique.

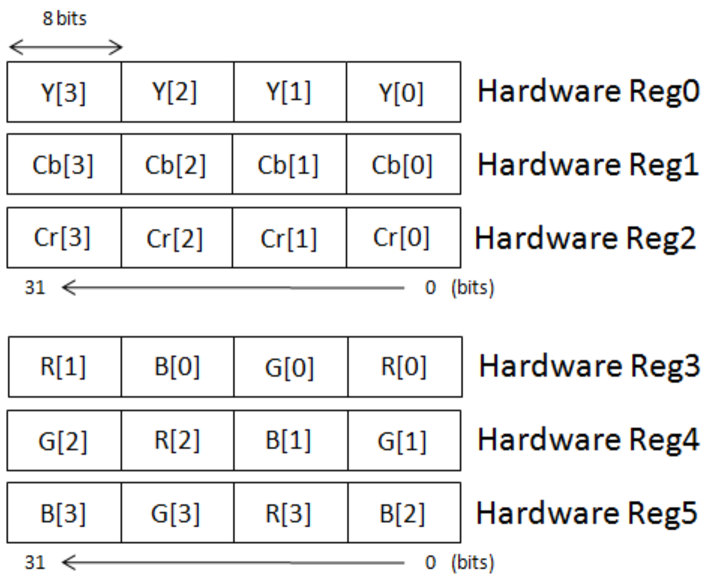


Figure 4.6: Colour Conversion array-register data compaction

Similarly, after conversion the RGB registers are read in the same manner. This data compaction utilizes 3 registers for input, and 3 registers for output to transmit 24 colour channels. Using this technique the software loops through all the pixels in the image, in 4 pixel increments, writing YCC and reading RGB without any need for control registers. No control registers are required, due to the fact that this conversion is very simple, and the hardware computes the conversion on 4 pixels before the software can read it.

4.5 Summary

This chapter presents the techniques, and structure of the proposed software-hardware hybrid architecture for the decompression of JPEG images. The JPEG software library is accelerated with hardware components for the 2D-IDCT and YCC to RGB colour conversion. Huffman decoding, the next logical step in JPEG decompression acceleration, is outlined in Appendix E.

In the following chapter, this work is timed, and tested. The results show how fast the decompression time has increased as well as the accuracy of the decompression with respect to a software only decompressed image.

Chapter 5

Results

This chapter presents the testing setup and procedures used to test the proposed hardware accelerated components of libjpeg, as well as the timing analysis. The conclusions that may be derived from the results are described, and then compared to the widely used software library, libjpeg.

5.1 Testing Setup and Procedure

Testing for this work was carried out on the Xilinx XUPV5-LX110T FPGA development board. The board was loaded with the GNU/Linux variant, Petalinux. The base design, hardware and software only included the most basic and crucial components. The hardware designed for the system includes the IDCT and colour conversion modules. These modules are linked into the MicroBlaze design, and accessed through input and output registers.

Petalinux includes the libjpeg source files which can be cross-compiled to run on the target. This library is the base in which this work is built on. The Petalinux kernel was configured for the Xilinx target hardware, MicroBlaze processor, and built for a Memory Management Unit (MMU) system. Once the Hardware is downloaded to the board, it is loaded with the Petalinux image. User built binaries can be loaded into the image, but for the purposes of this work the binaries and test JPEG images were executed through the network due to their size. Utilizing the network permits for faster development, testing, and eliminates the need to build the large test JPEG images from being built into the Petalinux flash image.

For this work, the necessary libjpeg files were modified and cross-compiled into a static library. Shared library compilation is possible, however it requires the Petalinux to be rebuilt and downloaded to the target which is not a productive solution. Upon generation of the static JPEG libray the main program files are cross-compiled with the library into a binary suitable for the target.

On the FPGA target, the binary is run by providing the path to JPEG image. Timing of the images is a difficult task. The work done in the areas of timing and profiling software applications on a soft processor, is underdeveloped and not readily available. As a result, this work depends on using the Petalinux ”time” function.

The time function provides the real, user, and system time execution of a binary with a resolution of 10 milliseconds. This work uses the user time which is presented in 5.2.

To verify the decompression of the images were accurate, a subjective test and objective test were performed. After decompression, the raw RGB data was written to the system's frame-buffer, which is displayed on an attached monitor. Visual, subjective inspection for artifacts, and similarity to the original was carried out on the image. Objective verification was implemented by comparing the decompressed and converted raw RGB values output from this work compared to the software only output. The verification results are presented in 5.3.

Not all images are the same, but will posses the same decompression scheme. For this work, 3 separate images with varying image metrics at 3 different pixel densities, 5 MP, 10 MP, and 20 MP were tested. To maintain statistical accuracy each image was tested 100 times and averaged using the scripts in Appendix B and a spreadsheet program. The three images, found in Appendix A, hereby referred to as the *bookstore, dog, and beach* images were the test images used in this work. The images began at an approximate 20MP, and were scaled down to 10MP, and 5MP to observe the versatility of this work. Each image was saved with no colour sub-sampling, and no Huffman optimizations to emulate the ideal settings observed in digital cameras. Table 5.1 gives the dimensional details of each image used.

Image	Width(px)	Height(px)	Size(MP)	Size(MB)
bookstore	2686	1862	5.0	1.5
	3799	2634	10.0	2.6
	5400	3744	20.2	4.1
dog	2560	1920	4.9	1.8
	3653	2740	10.01	3
	5164	3873	20.0	4.8
beach	2503	1998	5.0	1.8
	3540	2826	10.0	3.1
	5040	4024	20.2	4.9

Table 5.1: Test Image Dimensions

5.2 Timing Results

After each image was tested and timed, it was compared to the software only benchmark. Comparing to the benchmark, the timing improvement (Δt), and percentage of the benchmark was calculated (% SW) as shown in Tables 5.2, 5.3, and 5.4.

$$\text{\% of Software} = \frac{\text{Hardware time(s)}}{\text{Software time(s)}} * 100 \tag{5.1}$$

	5MP			10MP			20MP		
	Time(s)	Δt(s)	% SW	Time(s)	Δt(s)	% SW	Time(s)	Δt(s)	% SW
Software Only	49.18	-	-	96.71	-	-	184.54	-	-
Colour HW	46.66	2.52	94.87	91.87	4.84	94.99	174.08	10.46	94.33
IDCT HW	35.97	13.21	73.14	71.16	25.55	73.58	139.78	44.76	75.75
IDCT + Colour HW	33.28	15.91	67.66	65.73	30.98	67.97	128.72	55.82	69.75

Table 5.2: Bookstore Image Results

	5MP			10MP			20MP		
	Time(s)	Δt(s)	% SW	Time(s)	Δt(s)	% SW	Time(s)	Δt(s)	% SW
Software Only	48.44	-	-	95.88	-	-	188.86	-	-
Colour HW	45.84	2.60	94.62	90.64	5.24	94.54	176.91	11.95	93.67
IDCT HW	33.95	14.49	70.08	71.16	24.72	74.22	141.29	47.57	74.81
IDCT + Colour HW	32.93	15.52	67.97	65.63	30.25	68.45	130.31	58.55	69

Table 5.3: Dog Image Results

	5MP			10MP			20MP		
	Time(s)	Δt(s)	% SW	Time(s)	Δt(s)	% SW	Time(s)	Δt(s)	% SW
Software Only	49.73	-	-	96.30	-	-	189.40	-	-
Colour HW	46.85	2.87	94.22	91.72	4.58	95.24	177.02	12.37	93.47
IDCT HW	36.84	12.88	74.1	71.58	24.73	74.33	142.21	47.18	75.09
IDCT + Colour HW	33.89	15.83	68.16	66.22	30.08	68.76	131.19	58.21	69.27

Table 5.4: Beach Image Results

Tables 5.2, 5.3, and 5.4 illustrate that the hardware accelerated colour conversion provided an approximate 4% increase in timing, hardware accelerated 2D-IDCT provided an approximate 25% increase in timing, and the combined hardware components provided an approximate 31% increase in the timing over the software only approach for decompression. Furthermore, the individual as well as combined hardware acceleration of these components exhibits a near linear timing improvement as image size grows as shown in Figure 5.1 and 5.2.

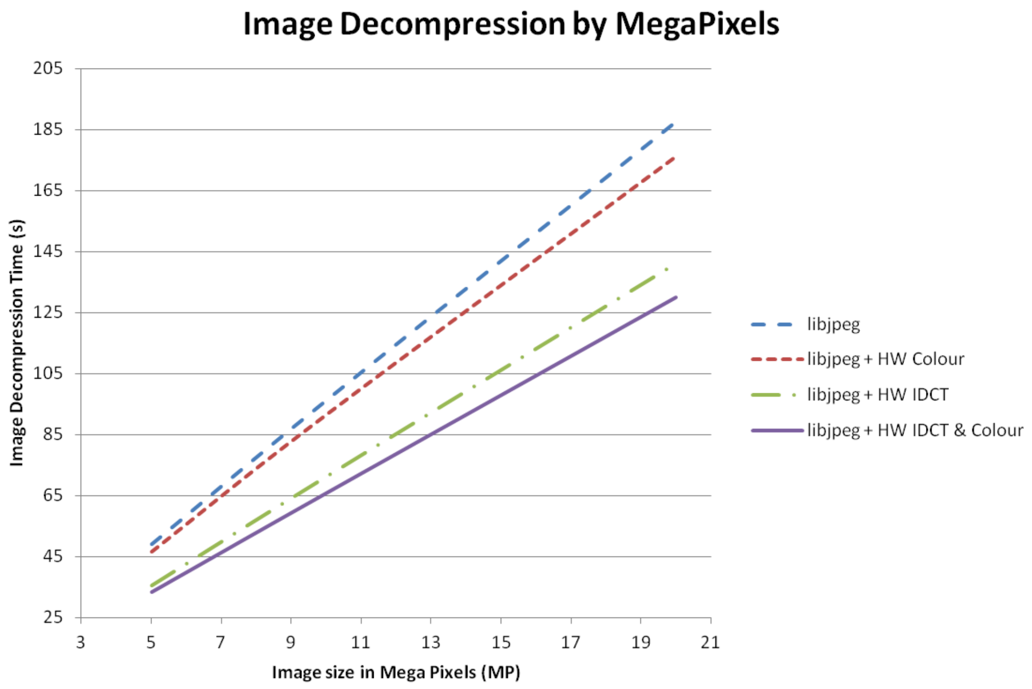


Figure 5.1: Image Decompression Timing

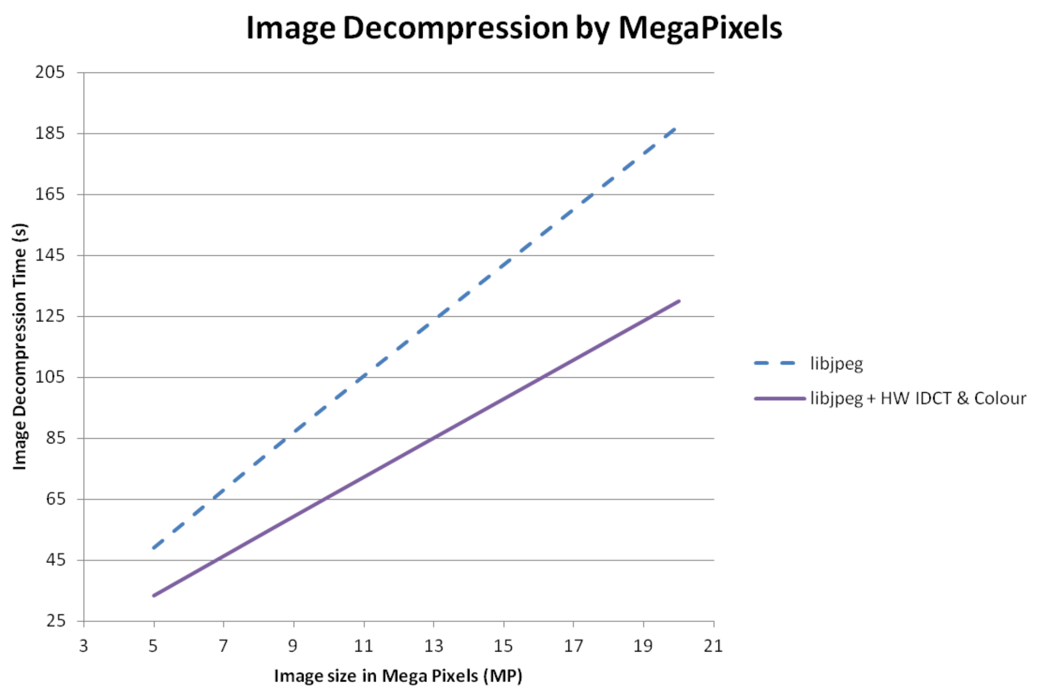


Figure 5.2: Image Decompression Timing

5.2.1 Hardware Timing

The primary source of improvement in this work is due to the speed of the hardware modules. Hardware has little to no overhead. Analyzing the hardware performance without the software overhead; the 2D-IDCT Hardware takes 186 clocks to achieve a full 8x8 transformation, and the YCC to RGB colour conversion takes 9 clocks to transform 4 pixels with 3 colour components. At 125 MHz, the IDCT takes 1.44 μs for one 8x8 block, and the colour conversion transforms 4 pixels in 7 2ns. Applying these calculations to each 20 MP test image can be found in Table 5.5.

Image	# 8x8 Blocks	# pixels/4	IDCT HW(s)	Colour HW(s)	% HW Time
bookstore	947700	15163200	1.36	1.09	1.90
dog	937509	15000129	1.35	1.08	1.86
beach	950670	15210720	1.37	1.10	1.88

Table 5.5: Hardware Only Timing per test image

5.3 Image Verification

Since both of the IDCT and Colour Conversion algorithms use fixed point arithmetic, some error will be introduced. This work aims to improve the timing of the entire JPEG decompression process while maintaining a valid output with minimal error. As a result the output of the image needs to be verified. The accuracy for this work was verified using subjective and objective comparisons against the original, software only, decompression.

For the subjective verification, the images were written to the system's frame buffer, a reserved region of memory that is processed with a video controller to be displayed on an attached monitor. The displayed images were then visually inspected for any image artifacts, and any differences in colour, shade, and orientation compared to the original.

For the objective verification the 5MP images were tested. The raw RGB bytes from the software only and software-hardware methods were saved to files. The approximate 15 million bytes of each image were then compared side by side to find the offsets. The byte offsets were calculated and plotted in the histograms in Figures 5.3, 5.4, and 5.5.

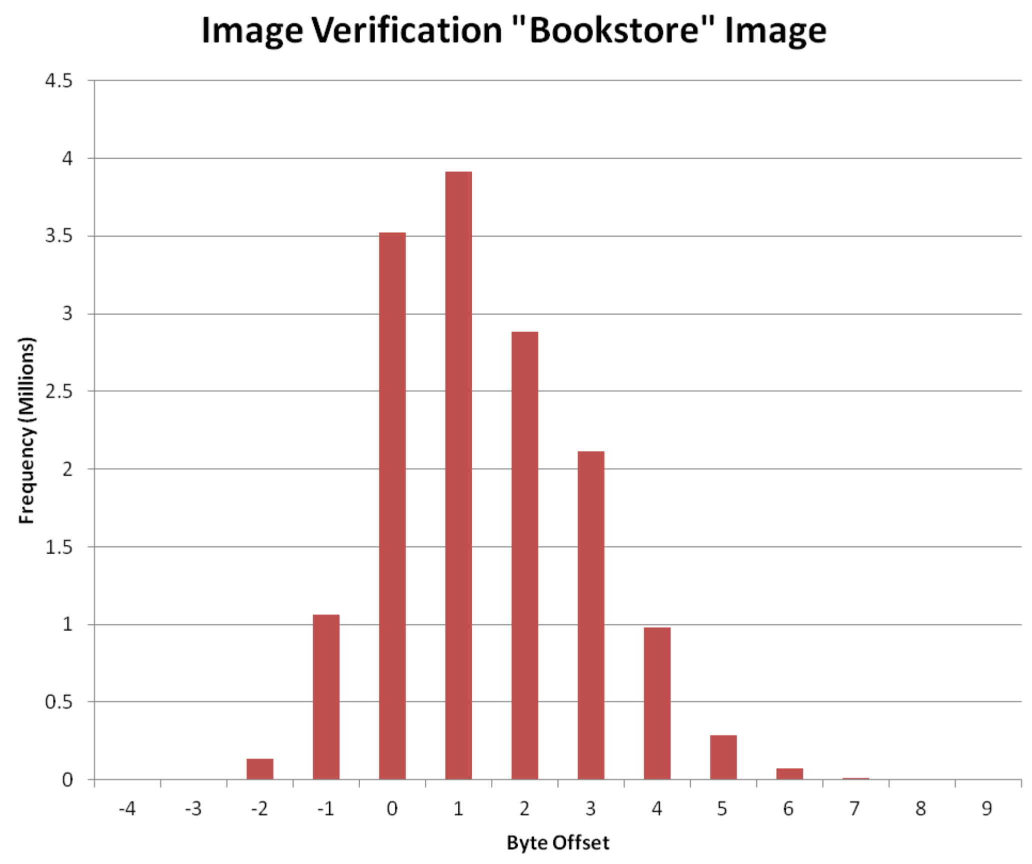


Figure 5.3: Bookstore image byte offset histogram

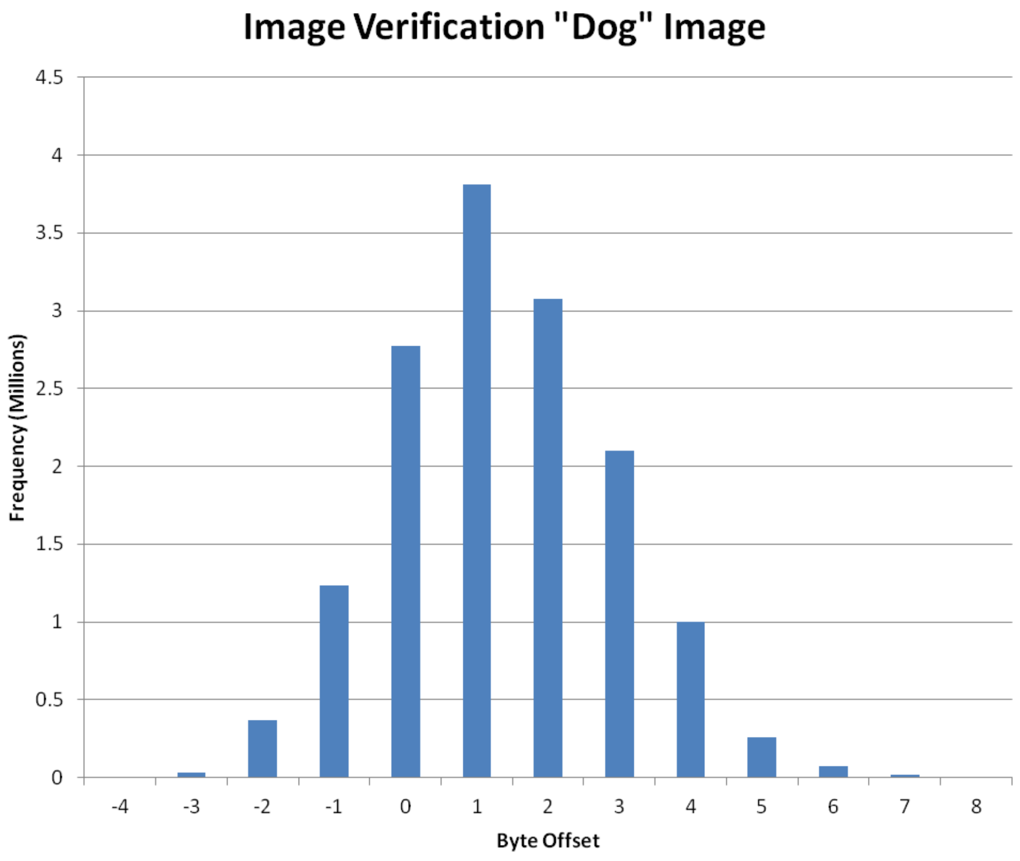


Figure 5.4: Dog image byte offset histogram

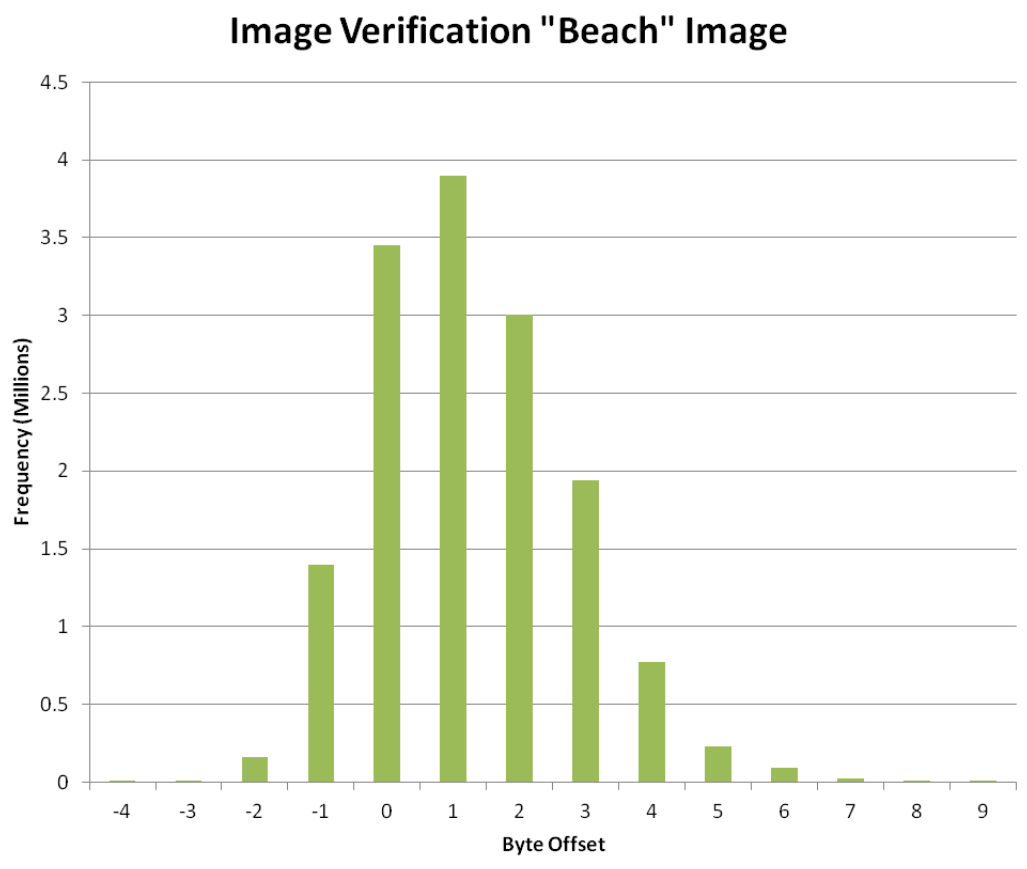


Figure 5.5: beach image byte offset histogram

A zero byte offset is ideal, meaning the decompressed version matches the software decompressed image perfectly. In the Figures 5.3, 5.4, and 5.5 the highest frequencies occur at the lowest offsets, and lowest frequencies occur at the highest offsets. Interestingly the highest frequency peaks exist at the offset of 1, likely from a rounding that appears during integer truncation. The worst case scenario in all cases is a byte offset of 9 at an extremely low frequency of 111. Quantifying the image data of the histograms can be performed using equations 5.2 and 5.3, which will find the similarity between the software decompressed image versus the proposed method.

$$\text{Byte Accuracy (\%)} = (1 - \left|\frac{\text{byte offset}}{255}\right|) * 100 \tag{5.2}$$

$$\begin{aligned}\text{Image Accuracy(\%)} = &\frac{\sum_{i=N}^{M}\left(1 - \left|\frac{\text{byte offset}_i}{255}\right|\right) * \text{byte offset frequency}_i}{\text{total \# of bytes}} * 100 \\ &where: \\ &N = \text{minimum byte offset} \\ &M = \text{maximum byte offset}\end{aligned} \tag{5.3}$$

Table 5.6 shows that the hardware decompressed test images are nearly 100% of their software decompressed counterpart.

Image	Percentage of SW Image (%)
bookstore	99.3914
dog	99.357
beach	99.4129

Table 5.6: Test Image Verification

5.4 Summary

This chapter presents the results of the proposed design. The hybrid software-hardware architecture was timed against a software only method providing 31% increase in timing in the decompression of a 20MP image, while maintaining over 99% accuracy. The timing was analyzed with 5MP, 10MP and 20MP images to observe that this design is superior to that of software as image size increases.

The following chapter concludes this work, and proposes design improvements for future work.

Chapter 6

Conclusion and Recommendations

The evolution of technology has greatly affected everyday consumers in the areas of mobile computing, and digital multimedia. The size of an image in digital photography, in terms of pixels, is constantly increasing as sensors, processors, and memory are improved. Mobile devices are approaching and meeting the processor speeds and architectures of their desktop counterparts. However, the methodology and algorithms used to decompress digital JPEG images on these powerful embedded systems is outdated and ineffective with the growing image sizes.

Since the invention of the JPEG image, software methods have been the sole answer for decompression. The JPEG decompression process contains many computations which increase with the image size. As computations increase so will the decompression time. To assist the software it is imperative to consider hardware acceleration in the mobile system to improve the decompression time of these images.

The research presented in this thesis focuses on the most cumbersome processing found in the software decompression of JPEG images. Hardware acceleration of the 2D-IDCT, YCC to RGB colour space conversion, and Huffman decoding processes were examined. The proposed solution to expedite slow JPEG decompression is hardware acceleration proven with the use of an FPGA. This novel design proves to combine the flexibility of software with the speed of hardware, resulting in a hybrid approach which accelerates the JPEG decompression significantly, while maintaining more than 99% of the original image.

The novelty of this architecture is the hybridization of the hardware and software to calculate the 2D-IDCT, and the YCC to RGB colour conversion. Multiplications in both components are calculated in parallel to optimize the throughput from software to hardware, preventing the need for the software to wait for the hardware to complete. The optimizations produced an average of 31% decrease in decompression time of a 20MP colour image, with no chroma-subsampling and no Huffman optimizations.

6.1 Recommendations

This design has greatly improved the JPEG decompression process, but bottlenecks in the system do remain. The JPEG process is natively sequential, which limits some parallelization optimizations from being possible. The most significant bottleneck in this design, proved to be the software design and layout. Specifically the linearity of the library as well as memory management and data transfer from software to hardware. A memory managed GNU/Linux kernel, which many embedded systems are based off, has limitations on the user-space from accessing its own physical memory addresses. This requires the need for kernel involvement, at the cost of time. Memory management is a concern in a typical software implementation to avoid lagging or crashing the system. Hardware would have the ability to act like a co-processor, but in this implementation the memory access is limited by the library layout. For these reasons, my recommendation for future improvement involves rewriting the libjpeg library with these considerations in mind. The library would need to implement a shared memory pool between software and hardware for direct access to the raw JPEG as well as the output buffer. This would allow less data copying, permit the hardware to relieve more workload from software, as well as access raw data for the Huffman decoding process.

References

[1] Microblaze soft processor core @ONLINE, January 2011. http://www.xilinx.com/tools/microblaze.htm.

[2] Petalinux software development kit @ONLINE, January 2011. http://www.xilinx.com/tools/petalinux-sdk.htm.

[3] Xilinx university program xupv-lx110t development platform @ONLINE, January 2011. http://www.xilinx.com/products/boards-and-kits/XUPV5-LX110T.htm.

[4] Joint photographic experts group @ONLINE, January 2012. http://www.jpeg.org.

[5] Calvin Hass. Impulse adventure - jpeg huffman coding tutorial @ONLINE, March 2011. http://www.impulseadventure.com/photo/jpeg-huffman-coding.html.

[6] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[7] Tom Lane. Independent jpeg group @ONLINE, October 2010. http://www.ijg.org/.

[8] C. Loeffler, A. Ligtenberg, and G.S. Moschytz. Practical fast 1-d dct algorithms with 11 multiplications. In *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*, pages 988 –991 vol.2, may 1989.

[9] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[10] Wendi Pan. A fast 2-d dct algorithm via distributed arithmetic optimization. In *Image Processing, 2000. Proceedings. 2000 International Conference on*, volume 3, pages 114 –117 vol.3, 2000.

[11] Soo-Chang Pei and Jian-Jiun Ding. Reversible integer color transform with bit-constraint. In *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, volume 3, pages III – 964–7, sept. 2005.

[12] R. Swamy, M. Khorasani, Yongjie Liu, D. Elliott, and S. Bates. A fast, pipelined implementation of a two-dimensional inverse discrete cosine transform. In *Electrical and Computer Engineering, 2005. Canadian Conference on*, pages 665 –668, may 2005.

[13] G.K. Wallace. The jpeg still picture compression standard. *Consumer Electronics, IEEE Transactions on*, 38(1):18 –19, feb 1992.

[14] Yang Yang, Peng Yuhua, and Liu Zhaoguang. A fast algorithm for ycbcr to rgb conversion. *Consumer Electronics, IEEE Transactions on*, 53(4):1490 –1493, nov. 2007.

Appendix A

Source Images

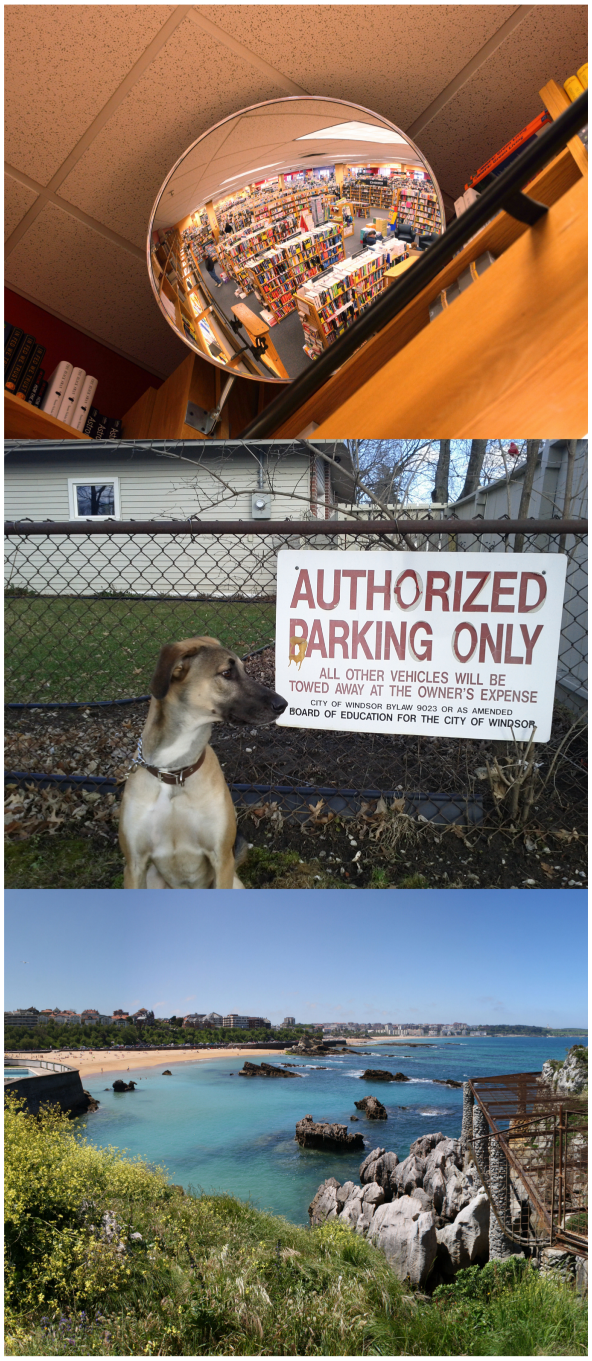

AUTHORIZED
PARKING ONLY
ALL OTHER VEHICLES WILL BE
TOWED AWAY AT THE OWNER'S EXPENSE
CITY OF WINDSOR BYLAW 9023 OR AS AMENDED
BOARD OF EDUCATION FOR THE CITY OF WINDSOR

Figure A.1: Sample 20MP images

Appendix B

Testing Scripts

```
#!/bin/bash
#
# This script times and  runs the Secondary scripts with the given parameters
#
# COUNT = The number of times to run the secondary script
# RESULT_FILE = the files to store the timing results in
# PROG = precompiled program name
#      0 = Software only
#      1 = IDCT HW accelerated
#      2 = Colour Conversion HW accelerated
#      3 = IDCT + Colour Conversion HW accelerated
# ARG1 = the path to the image to be decoded
#      Available images:
#       5MP_1.jpg, 5MP_2.jpg, 5MP_3.jpg,
#       10MP_1.jpg, 10MP_2.jpg, 10MP_3.jpg,
#       20MP_1.jpg, 20MP_2.jpg, 20MP_3.jpg
# ARG2 = boolean to write the resultant image to the system's framebuffer
# ARG3 = factor to scale the image down

# Variables
COUNT=20 # user-defined
RESULT_FILE0=results_SW_timing.txt
RESULT_FILE1=results_idct_timing.txt
RESULT_FILE2=results_colour_timing.txt
RESULT_FILE3=results_total_timing.txt
PROG0=./Image_SW_control
PROG1=./ImageHw_idct
PROG2=./ImageHw_colour
PROG3=./ImageHw_total
ARG2=0                  # fb: 0 = no, 1 = yes
ARG3=1                  # scale factor: 1-8
ARG1=pics/pic.jpg    # path to image file
```

```
## list of available images
## 10MPimage.jpg  20MPimage.jpg  asdf.jpg  Jaggies_test.jpg  maxx2.jpg  soheil.jpg

# remove the old result files
echo "--- removing $RESULT_FILE0 ---"
rm $RESULT_FILE0
echo "--- removing $RESULT_FILE1 ---"
rm $RESULT_FILE1
echo "--- removing $RESULT_FILE2 ---"
rm $RESULT_FILE2
echo "--- removing $RESULT_FILE3 ---"
rm $RESULT_FILE3

############### 5MPimages.jpg ####################
ARG1=pics/5MP_1.jpg   # path to image file
echo "================ $ARG1 ================"

sh secondary_test.sh $PROG0 $ARG1 $ARG2 $ARG3 $RESULT_FILE0 $COUNT
sh secondary_test.sh $PROG1 $ARG1 $ARG2 $ARG3 $RESULT_FILE1 $COUNT
sh secondary_test.sh $PROG2 $ARG1 $ARG2 $ARG3 $RESULT_FILE2 $COUNT
sh secondary_test.sh $PROG3 $ARG1 $ARG2 $ARG3 $RESULT_FILE3 $COUNT

ARG1=pics/5MP_2.jpg   # path to image file
echo "================ $ARG1 ================"

sh secondary_test.sh $PROG0 $ARG1 $ARG2 $ARG3 $RESULT_FILE0 $COUNT
sh secondary_test.sh $PROG1 $ARG1 $ARG2 $ARG3 $RESULT_FILE1 $COUNT
sh secondary_test.sh $PROG2 $ARG1 $ARG2 $ARG3 $RESULT_FILE2 $COUNT
sh secondary_test.sh $PROG3 $ARG1 $ARG2 $ARG3 $RESULT_FILE3 $COUNT

ARG1=pics/5MP_3.jpg   # path to image file
echo "================ $ARG1 ================"

sh secondary_test.sh $PROG0 $ARG1 $ARG2 $ARG3 $RESULT_FILE0 $COUNT
sh secondary_test.sh $PROG1 $ARG1 $ARG2 $ARG3 $RESULT_FILE1 $COUNT
sh secondary_test.sh $PROG2 $ARG1 $ARG2 $ARG3 $RESULT_FILE2 $COUNT
sh secondary_test.sh $PROG3 $ARG1 $ARG2 $ARG3 $RESULT_FILE3 $COUNT

############### 10MPimages.jpg ####################
ARG1=pics/10MP_1.jpg   # path to image file
echo "================ $ARG1 ================"

sh secondary_test.sh $PROG0 $ARG1 $ARG2 $ARG3 $RESULT_FILE0 $COUNT
sh secondary_test.sh $PROG1 $ARG1 $ARG2 $ARG3 $RESULT_FILE1 $COUNT
sh secondary_test.sh $PROG2 $ARG1 $ARG2 $ARG3 $RESULT_FILE2 $COUNT
sh secondary_test.sh $PROG3 $ARG1 $ARG2 $ARG3 $RESULT_FILE3 $COUNT

ARG1=pics/10MP_2.jpg   # path to image file
echo "================ $ARG1 ================"

sh secondary_test.sh $PROG0 $ARG1 $ARG2 $ARG3 $RESULT_FILE0 $COUNT
sh secondary_test.sh $PROG1 $ARG1 $ARG2 $ARG3 $RESULT_FILE1 $COUNT
sh secondary_test.sh $PROG2 $ARG1 $ARG2 $ARG3 $RESULT_FILE2 $COUNT
sh secondary_test.sh $PROG3 $ARG1 $ARG2 $ARG3 $RESULT_FILE3 $COUNT

ARG1=pics/10MP_3.jpg   # path to image file
echo "================ $ARG1 ================"

sh secondary_test.sh $PROG0 $ARG1 $ARG2 $ARG3 $RESULT_FILE0 $COUNT
sh secondary_test.sh $PROG1 $ARG1 $ARG2 $ARG3 $RESULT_FILE1 $COUNT
sh secondary_test.sh $PROG2 $ARG1 $ARG2 $ARG3 $RESULT_FILE2 $COUNT
sh secondary_test.sh $PROG3 $ARG1 $ARG2 $ARG3 $RESULT_FILE3 $COUNT

############### 20MPimages.jpg ####################
ARG1=pics/20MP_1.jpg   # path to image file
echo "================ $ARG1 ================"

sh secondary_test.sh $PROG0 $ARG1 $ARG2 $ARG3 $RESULT_FILE0 $COUNT
sh secondary_test.sh $PROG1 $ARG1 $ARG2 $ARG3 $RESULT_FILE1 $COUNT
sh secondary_test.sh $PROG2 $ARG1 $ARG2 $ARG3 $RESULT_FILE2 $COUNT
sh secondary_test.sh $PROG3 $ARG1 $ARG2 $ARG3 $RESULT_FILE3 $COUNT

ARG1=pics/20MP_2.jpg   # path to image file
echo "================ $ARG1 ================"

sh secondary_test.sh $PROG0 $ARG1 $ARG2 $ARG3 $RESULT_FILE0 $COUNT
sh secondary_test.sh $PROG1 $ARG1 $ARG2 $ARG3 $RESULT_FILE1 $COUNT
sh secondary_test.sh $PROG2 $ARG1 $ARG2 $ARG3 $RESULT_FILE2 $COUNT
sh secondary_test.sh $PROG3 $ARG1 $ARG2 $ARG3 $RESULT_FILE3 $COUNT

ARG1=pics/20MP_3.jpg   # path to image file
echo "================ $ARG1 ================"

sh secondary_test.sh $PROG0 $ARG1 $ARG2 $ARG3 $RESULT_FILE0 $COUNT
sh secondary_test.sh $PROG1 $ARG1 $ARG2 $ARG3 $RESULT_FILE1 $COUNT
sh secondary_test.sh $PROG2 $ARG1 $ARG2 $ARG3 $RESULT_FILE2 $COUNT
sh secondary_test.sh $PROG3 $ARG1 $ARG2 $ARG3 $RESULT_FILE3 $COUNT
```

master_test.sh

```
#!/bin/bash
# This script decodes JPEG images with the 6 parameters passed to it.
#
# It times and runs the
# <PROGRAM> with <ARG#>
# <COUNT> number of times logging/appending the data to <RESULT_FILE>
#
# This script is called by master_test.sh with:
#  sh secondary_test $PROGNAME $PIC $FB $SCALE_FACTOR $RESULT_FILE $COUNT

PROGRAM=$1
ARG1=$2
ARG2=$3
ARG3=$4
RESULT_FILE=$5
COUNT=$6

echo "************$ARG1********">>$RESULT_FILE  # append the picture name to the
    result file

echo "************$PROGRAM********" #echo the progress to the terminal

i=1

while [ $i -le $COUNT ]
do
  echo "$i"
  i=`expr $i + 1`

 (time $PROGRAM $ARG1 $ARG2 $ARG3) 2>>$RESULT_FILE # append the timing information to
      the result file
 echo -e "\012" >>$RESULT_FILE            # add extra line to the result file

done
```

secondary_test.sh

Appendix C

C Code

C.1 libjpeg modifications

```
/*
 * jpeglib.h
 *
 * Copyright (C) 1991-1998, Thomas G. Lane.
 * Modified 2002-2009 by Guido Vollbeding.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file defines the application interface for the JPEG library.
 * Most applications using the library need only include this file,
 * and perhaps jerror.h if they want to know the exact error codes.
 *
 * Modified 2012 by Dan MacDonald
 * Added hardware register pointers for the jpeg_decompress_struct
 */

#ifndef JPEGLIB_H
#define JPEGLIB_H

...

/* Master record for a decompression instance */

struct jpeg_decompress_struct {
  jpeg_common_fields;      /* Fields shared with jpeg_compress_struct */

  /* Source of compressed data */
  struct jpeg_source_mgr * src;

   /* Hardware mmap'd variables
    * c_convert_reg = the pointer to the colour converter register space (use
        appropriate Register offsets)
    * idct_reg = the pointer to the idct register space (use appropriate Register
        offsets)
    */
   volatile void *c_convert_reg;
   volatile void *idct_reg;

  /* Basic description of image --- filled in by jpeg_read_header(). */
  /* Application may inspect these values to decide how to process image. */

  JDIMENSION image_width;  /* nominal image width (from SOF marker) */
  JDIMENSION image_height; /* nominal image height */
  int num_components;      /* # of color components in JPEG image */
  J_COLOR_SPACE jpeg_color_space; /* colorspace of JPEG image */
```

jpeglib.h

```
/*
 * jidctint.c
 *
 * Copyright (C) 1991-1998, Thomas G. Lane.
 * Modification developed 2002-2009 by Guido Vollbeding.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains a slow-but-accurate integer implementation of the
 * inverse DCT (Discrete Cosine Transform).  In the IJG code, this routine
 * must also perform dequantization of the input coefficients.
 *
 * ...
 *
 * Modified 2012 by Dan MacDonald.
 * Implemented the 8x8 2D-IDCT in Hardware
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdct.h"    /* Private declarations for DCT subsystem */

...

/*
 * Perform dequantization and inverse DCT on one block of coefficients.
 * IN HARDWARE
 */
GLOBAL(void)
jpeg_idct_islow (j_decompress_ptr cinfo, jpeg_component_info * compptr,
       JCOEFPTR coef_block,
       JSAMPARRAY output_buf, JDIMENSION output_col)
{
  JCOEFPTR inptr;
  ISLOW_MULT_TYPE * quantptr;
  volatile JSAMPROW outptr;
  // 32 bit pointers to improve throughput
  volatile unsigned int * int_outptr;
  unsigned int *int_inptr;
  unsigned int *reg_outptr;
  volatile unsigned int *inregptr;
  volatile unsigned int *outregptr;
  volatile unsigned int *doneptr;

  // allocate pointers
  inptr = coef_block;
```

```
int_inptr = (unsigned int *)inptr;         // integer version of the inptr
quantptr = (ISLOW_MULT_TYPE *) compptr->dct_table;

//register transfer pointers
inregptr = (unsigned int *)(cinfo->idct_reg);         // pointer for quantization
    and input matrices
outregptr = (unsigned int *)(cinfo->idct_reg + 52); // pointer for output matrix
doneptr = (unsigned int *)(cinfo->idct_reg + 84);   // pointer to see if the hw is
    done

// write in the input DCT coefficients and the quantization matrix
// into the hardware registers row by row
// loop is unrolled to prevent any unwanted compiler optimizations

 //------------------row0------------------
 inregptr[0] = *quantptr++;
 inregptr[1] = *quantptr++;
 inregptr[2] = *quantptr++;
 inregptr[3] = *quantptr++;
 inregptr[4] = *quantptr++;
 inregptr[5] = *quantptr++;
 inregptr[6] = *quantptr++;
 inregptr[7] = *quantptr++;

 inregptr[8] = *int_inptr++;
 inregptr[9] = *int_inptr++;
 inregptr[10] = *int_inptr++;
 inregptr[11] = *int_inptr++;

 //------------------row1------------------
 inregptr[0] = *quantptr++;
 inregptr[1] = *quantptr++;
 inregptr[2] = *quantptr++;
 inregptr[3] = *quantptr++;
 inregptr[4] = *quantptr++;
 inregptr[5] = *quantptr++;
 inregptr[6] = *quantptr++;
 inregptr[7] = *quantptr++;

 inregptr[8] = *int_inptr++;
 inregptr[9] = *int_inptr++;
 inregptr[10] = *int_inptr++;
 inregptr[11] = *int_inptr++;

 //------------------row2------------------
 inregptr[0] = *quantptr++;
 inregptr[1] = *quantptr++;
 inregptr[2] = *quantptr++;
 inregptr[3] = *quantptr++;
 inregptr[4] = *quantptr++;
 inregptr[5] = *quantptr++;
 inregptr[6] = *quantptr++;
 inregptr[7] = *quantptr++;

 inregptr[8] = *int_inptr++;
 inregptr[9] = *int_inptr++;
 inregptr[10] = *int_inptr++;
 inregptr[11] = *int_inptr++;
 //------------------row3------------------
 inregptr[0] = *quantptr++;
 inregptr[1] = *quantptr++;
 inregptr[2] = *quantptr++;
 inregptr[3] = *quantptr++;
 inregptr[4] = *quantptr++;
 inregptr[5] = *quantptr++;
 inregptr[6] = *quantptr++;
 inregptr[7] = *quantptr++;
```

```
inregptr[8] = *int_inptr++;
inregptr[9] = *int_inptr++;
inregptr[10] = *int_inptr++;
inregptr[11] = *int_inptr++;

//------------------row4------------------
inregptr[0] = *quantptr++;
inregptr[1] = *quantptr++;
inregptr[2] = *quantptr++;
inregptr[3] = *quantptr++;
inregptr[4] = *quantptr++;
inregptr[5] = *quantptr++;
inregptr[6] = *quantptr++;
inregptr[7] = *quantptr++;

inregptr[8] = *int_inptr++;
inregptr[9] = *int_inptr++;
inregptr[10] = *int_inptr++;
inregptr[11] = *int_inptr++;
//------------------row5------------------
inregptr[0] = *quantptr++;
inregptr[1] = *quantptr++;
inregptr[2] = *quantptr++;
inregptr[3] = *quantptr++;
inregptr[4] = *quantptr++;
inregptr[5] = *quantptr++;
inregptr[6] = *quantptr++;
inregptr[7] = *quantptr++;

inregptr[8] = *int_inptr++;
inregptr[9] = *int_inptr++;
inregptr[10] = *int_inptr++;
inregptr[11] = *int_inptr++;
//------------------row6------------------
inregptr[0] = *quantptr++;
inregptr[1] = *quantptr++;
inregptr[2] = *quantptr++;
inregptr[3] = *quantptr++;
inregptr[4] = *quantptr++;
inregptr[5] = *quantptr++;
inregptr[6] = *quantptr++;
inregptr[7] = *quantptr++;

inregptr[8] = *int_inptr++;
inregptr[9] = *int_inptr++;
inregptr[10] = *int_inptr++;
inregptr[11] = *int_inptr++;
//------------------row7------------------
inregptr[0] = *quantptr++;
inregptr[1] = *quantptr++;
inregptr[2] = *quantptr++;
inregptr[3] = *quantptr++;
inregptr[4] = *quantptr++;
inregptr[5] = *quantptr++;
inregptr[6] = *quantptr++;
inregptr[7] = *quantptr++;

inregptr[8] = *int_inptr++;
inregptr[9] = *int_inptr++;
inregptr[10] = *int_inptr++;
inregptr[11] = *int_inptr++;

     // wait for the hardware to finish
while(*doneptr !=1);

//unrolled reading of each row of the output registers
int_outptr = (int *)(output_buf[0] + output_col);
int_outptr[0] = outregptr[0];
```

```
   int_outptr[1] = outregptr[1];

   int_outptr = (int *)(output_buf[1] + output_col);
   int_outptr[0] = outregptr[0];
   int_outptr[1] = outregptr[1];

   int_outptr = (int *)(output_buf[2] + output_col);
   int_outptr[0] = outregptr[0];
   int_outptr[1] = outregptr[1];

   int_outptr = (int *)(output_buf[3] + output_col);
   int_outptr[0] = outregptr[0];
   int_outptr[1] = outregptr[1];

   int_outptr = (int *)(output_buf[4] + output_col);
   int_outptr[0] = outregptr[0];
   int_outptr[1] = outregptr[1];

   int_outptr = (int *)(output_buf[5] + output_col);
   int_outptr[0] = outregptr[0];
   int_outptr[1] = outregptr[1];

   int_outptr = (int *)(output_buf[6] + output_col);
   int_outptr[0] = outregptr[0];
   int_outptr[1] = outregptr[1];

   int_outptr = (int *)(output_buf[7] + output_col);
   int_outptr[0] = outregptr[0];
   int_outptr[1] = outregptr[1];
}
```

jidctint.c

```
/*
 * jdcolor.c
 *
 * Copyright (C) 1991-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains output colorspace conversion routines.
 * ...
 *
 * Modified 2012 by Dan MacDonald.
 * Implemented the YCC to RGB Conversion in Hardware, and removed Software table
     generation
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Private subobject */
/* REMOVED. Hardware does not use tables
 *
typedef struct {
  struct jpeg_color_deconverter pub;

  // Private state for YCC->RGB conversion
  int * Cr_r_tab;    // => table for Cr to R conversion
  int * Cb_b_tab;    // => table for Cb to B conversion
  INT32 * Cr_g_tab;     // => table for Cr to G conversion
  INT32 * Cb_g_tab;     // => table for Cb to G conversion
} my_color_deconverter;

typedef my_color_deconverter * my_cconvert_ptr;
*/
```

```
/**************** YCbCr -> RGB conversion: most common case **************/

/*
 * YCbCr is defined per CCIR 601-1, except that Cb and Cr are
 * normalized to the range 0..MAXJSAMPLE rather than -0.5 .. 0.5.
 * The conversion equations to be implemented are therefore
 * R = Y                + 1.40200 * Cr
 * G = Y - 0.34414 * Cb - 0.71414 * Cr
 * B = Y + 1.77200 * Cb
 * where Cb and Cr represent the incoming values less CENTERJSAMPLE.
 * (These numbers are derived from TIFF 6.0 section 21, dated 3-June-92.)
 *
 * To avoid floating-point arithmetic, we represent the fractional constants
 * as integers scaled up by 2^16 (about 4 digits precision); we have to divide
 * the products by 2^16, with appropriate rounding, to get the correct answer.
 * Notice that Y, being an integral input, does not contribute any fraction
 * so it need not participate in the rounding.
 *
 * For even more speed, we avoid doing any multiplications in the inner loop
 * by precalculating the constants times Cb and Cr for all possible values.
 * For 8-bit JSAMPLEs this is very reasonable (only 256 entries per table);
 * for 12-bit samples it is still acceptable.  It's not very reasonable for
 * 16-bit samples, but if you want lossless storage you shouldn't be changing
 * colorspace anyway.
 * The Cr=>R and Cb=>B values can be rounded to integers in advance; the
 * values for the G calculation are left scaled up, since we must add them
 * together before rounding.
 */

...

/*
 * REMOVED. Hardware does not use tables
 *
LOCAL(void)
build_ycc_rgb_table (j_decompress_ptr cinfo)
{
  my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
  int i;
  INT32 x;
  SHIFT_TEMPS

  cconvert->Cr_r_tab = (int *)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (MAXJSAMPLE+1) * SIZEOF(int));
  cconvert->Cb_b_tab = (int *)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (MAXJSAMPLE+1) * SIZEOF(int));
  cconvert->Cr_g_tab = (INT32 *)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (MAXJSAMPLE+1) * SIZEOF(INT32));
  cconvert->Cb_g_tab = (INT32 *)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (MAXJSAMPLE+1) * SIZEOF(INT32));

  for (i = 0, x = -CENTERJSAMPLE; i <= MAXJSAMPLE; i++, x++) {
    // i is the actual input pixel value, in the range 0..MAXJSAMPLE
    // The Cb or Cr value we are thinking of is x = i - CENTERJSAMPLE
    // Cr=>R value is nearest int to 1.40200 * x
    cconvert->Cr_r_tab[i] = (int)
          RIGHT_SHIFT(FIX(1.40200) * x + ONE_HALF, SCALEBITS);
    // Cb=>B value is nearest int to 1.77200 * x
    cconvert->Cb_b_tab[i] = (int)
          RIGHT_SHIFT(FIX(1.77200) * x + ONE_HALF, SCALEBITS);
    // Cr=>G value is scaled-up -0.71414 * x
    cconvert->Cr_g_tab[i] = (- FIX(0.71414)) * x;
    // Cb=>G value is scaled-up -0.34414 * x
    // We also add in ONE_HALF so that need not do it in inner loop
    cconvert->Cb_g_tab[i] = (- FIX(0.34414)) * x + ONE_HALF;
  }
}
*/


/*
 * Perform YCC colour space to RGB conversion row by row
 * IN HARDWARE
 */
METHODDEF(void)
ycc_rgb_convert (j_decompress_ptr cinfo, JSAMPIMAGE input_buf, JDIMENSION input_row,
    JSAMPARRAY output_buf, int num_rows)
{
  my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
  register JSAMPROW outptr;
  register JSAMPROW inptr0, inptr1, inptr2;
  register JDIMENSION col;
  unsigned int* yptr;   // integer pointer at the y array
  unsigned int* cbptr;  // integer pointer at the cb array
  unsigned int* crptr;  // integer pointer at the cr array
  volatile unsigned int* rgbptr; // integer pointer for rgb array

  unsigned int *ycc_reg;
  unsigned int *rgb_reg;

  // Since 4 pixels are computed at once the loop has fewer iterations
  JDIMENSION num_cols = (cinfo->output_width)>>2;

   // register pointers
  ycc_reg = (unsigned int *)(cinfo->c_convert_reg);
  rgb_reg = (unsigned int *)(cinfo->c_convert_reg + 12);

  while (--num_rows >= 0) {
    yptr = (unsigned int*)(input_buf[0][input_row]);  // y
    cbptr = (unsigned int*)(input_buf[1][input_row]); // cb
    crptr = (unsigned int*)(input_buf[2][input_row]); // cr

    input_row++;
    outptr = *output_buf++;

    rgbptr = (unsigned int*)outptr;

    for (col = 0; col < num_cols; col++)
   {
      // write in the YCC values
      ycc_reg[0] = *yptr++;
      ycc_reg[1] = *cbptr++;
      ycc_reg[2] = *crptr++;

      // read the RGB values
      *rgbptr++ = rgb_reg[0];
      *rgbptr++ = rgb_reg[1];
      *rgbptr++ = rgb_reg[2];
   } // for loop
  } // while loop

}


GLOBAL(void)
jinit_color_deconverter (j_decompress_ptr cinfo)
{
  my_cconvert_ptr cconvert;
  int ci;

  cconvert = (my_cconvert_ptr)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            SIZEOF(my_color_deconverter));
  cinfo->cconvert = (struct jpeg_color_deconverter *) cconvert;
  cconvert->pub.start_pass = start_pass_dcolor;

  /* Make sure num_components agrees with jpeg_color_space */
  switch (cinfo->jpeg_color_space) {
  case JCS_GRAYSCALE:
    if (cinfo->num_components != 1)
      ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    break;

  case JCS_RGB:
  case JCS_YCbCr:
    if (cinfo->num_components != 3)
      ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    break;

  case JCS_CMYK:
  case JCS_YCCK:
    if (cinfo->num_components != 4)
      ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    break;

  default:        /* JCS_UNKNOWN can be anything */
    if (cinfo->num_components < 1)
      ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    break;
  }

  /* Set out_color_components and conversion method based on requested space.
   * Also clear the component_needed flags for any unused components,
   * so that earlier pipeline stages can avoid useless computation.
   */

  switch (cinfo->out_color_space) {
  case JCS_GRAYSCALE:
    cinfo->out_color_components = 1;
    if (cinfo->jpeg_color_space == JCS_GRAYSCALE ||
   cinfo->jpeg_color_space == JCS_YCbCr) {
      cconvert->pub.color_convert = grayscale_convert;
      /* For color->grayscale conversion, only the Y (0) component is needed */
      for (ci = 1; ci < cinfo->num_components; ci++)
   cinfo->comp_info[ci].component_needed = FALSE;
    } else
      ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;

  case JCS_RGB:
    cinfo->out_color_components = RGB_PIXELSIZE;
    if (cinfo->jpeg_color_space == JCS_YCbCr) {
      cconvert->pub.color_convert = ycc_rgb_convert;
      //build_ycc_rgb_table(cinfo);    // REMOVED. Hardware does not use tables
    } else if (cinfo->jpeg_color_space == JCS_GRAYSCALE) {
      cconvert->pub.color_convert = gray_rgb_convert;
    } else if (cinfo->jpeg_color_space == JCS_RGB && RGB_PIXELSIZE == 3) {
      cconvert->pub.color_convert = null_convert;
    } else
      ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;

  case JCS_CMYK:
    cinfo->out_color_components = 4;
    if (cinfo->jpeg_color_space == JCS_YCCK) {
      cconvert->pub.color_convert = ycck_cmyk_convert;
      //build_ycc_rgb_table(cinfo);
    } else if (cinfo->jpeg_color_space == JCS_CMYK) {
      cconvert->pub.color_convert = null_convert;
    } else
```

```
      ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;

  default:
    /* Permit null conversion to same output space */
    if (cinfo->out_color_space == cinfo->jpeg_color_space) {
      cinfo->out_color_components = cinfo->num_components;
      cconvert->pub.color_convert = null_convert;
    } else        /* unsupported non-null conversion */
      ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;
  }

  if (cinfo->quantize_colors)
    cinfo->output_components = 1; /* single colormapped output component */
  else
    cinfo->output_components = cinfo->out_color_components;
}
```

jdcolor.c

C.2 ReadImage.c

```
/*
 * ReadImage.c
 *
 * Created 2012, Dan MacDonald.
 * This file serves as the entry point for utilizing the software library, libjpeg,
 * Developed by the Independent JPEG Group, and modified for hardware acceleration
 * by Dan MacDonald
 *
 * This program accepts image arguments, decompresses the JPEG image and optionally
 * writes the image to the systems frame buffer, or writes the output to a file to
 * test its validity
 *
 */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "/opt/petalinux-v2.1-final-full/software/petalinux-dist/stage/usr/include/
    jpeglib.h"
#include <stdlib.h>  // used for malloc
#include <memory.h>  // used for memcpy
#include <unistd.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

/* MACROS */
#define C_CONVERT_PHYSADDR 0xA6E30000
#define IDCT_PHYSADDR 0xA6E20000

#define MAP_SIZE 4096UL // 4k (one page size)
#define MAP_MASK (MAP_SIZE - 1)

unsigned char *raw_image = NULL;

/* Function prototypes */
int read_jpeg_file( char *filename, int scale_factor );
int write_jpeg_file( char *filename );
void write_to_fb(void);

/* Global Parameters */
```

```
int width;
int height;
int bytes_per_pixel = 3;
int color_space = JCS_RGB;

/* main
 * Entry point for decompressing an image using libjpeg, modified and not modified
 *
 * Acceptable arg sequence:
 * <program> <image> <show fb(1 or 0)> <scale factor (1-8)>
 */
int main(int argc, char * argv[])
{
   char *outfilename = "image_output.jpg";
   int fb_tf;
   int scale_factor;


   if (argc >3)
   {
      scale_factor = atoi(argv[3]);
      fb_tf = atoi(argv[2]);

      if (read_jpeg_file(argv[1], scale_factor) >0)
      {
         printf("\n");
         // write the output file to compare with original
         //if (write_jpeg_file(outfilename) <0 )
         // return -1;

         //write the output to the framebuffer
         if (fb_tf == 1)
         write_to_fb();
      }
      else
      return -1;
   }
   else
   {
      printf("Invalid arguments: \nFormat: \n%s [photo.jpg] <fb> <Scale-factor> \n\
         photo.jpg = image to decompress \n\
         fb = 1 to show, 0 dont show\n\
         Scale-factor = 1-8\n", argv[0]);
   }

   return 0;
}


/* read_jpeg_file
 * This function Reads from a jpeg file on disk specified by filename and saves into
    the
 * raw_image buffer in an uncompressed format.
 *
 * returns positive integer if successful, -1 otherwise
 */
int read_jpeg_file( char *filename, int scale_factor )
{

   struct jpeg_decompress_struct cinfo;
   struct jpeg_error_mgr jerr;
   JSAMPROW row_pointer[1];

   FILE *infile = fopen( filename, "rb" );
   unsigned long location = 0;
   int i = 0;
   int fd;
   void *c_convert_reg_map_base;
   void *idct_reg_map_base;

   if ( !infile )
   {
      printf("Error opening jpeg file %s\n!", filename );
      return -1;
   }

   cinfo.err = jpeg_std_error( &jerr );

   jpeg_create_decompress( &cinfo );
   jpeg_stdio_src( &cinfo, infile );

   jpeg_read_header( &cinfo, TRUE );

   // Uncomment the following to output image information, if needed. */
   /*
   printf( "JPEG File Information: \n" );
   printf( "Image width and height: %d pixels and %d pixels.\n", cinfo.image_width,
       cinfo.image_height );
   printf( "Color components per pixel: %d.\n", cinfo.num_components );
   printf( "Color space: %d.\n", cinfo.jpeg_color_space );
   */

   // scale on the decompression (numerator/denominator)
   cinfo.scale_num = 1;
   cinfo.scale_denom = scale_factor;

   // Hardware Register memory mapping
   fd = open("/dev/mem",O_RDWR | O_SYNC);

   if (fd <0)
   {
      printf("Failure to open the file /dev/mem \n");
      fclose(infile);
      return -1;
   }
   else
   {
      // allocate memory mappings in jpeg decompress structure
      c_convert_reg_map_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
          fd, C_CONVERT_PHYSADDR & ~MAP_MASK);

      if(c_convert_reg_map_base <0)
         goto fail_mmap2;

      idct_reg_map_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
          IDCT_PHYSADDR & ~MAP_MASK);

      if(idct_reg_map_base <0)
         goto fail_mmap3;

      cinfo.c_convert_reg = c_convert_reg_map_base + (C_CONVERT_PHYSADDR & MAP_MASK);
      cinfo.idct_reg = idct_reg_map_base + (IDCT_PHYSADDR & MAP_MASK);

      goto continue_with_image;

      fail_mmap3:
      munmap(c_convert_reg_map_base, MAP_SIZE);
      printf("Failure to mmap idct registers\n");

      fail_mmap2:
      printf("Failure to mmap colour converter registers\n");

      fclose(infile);
      fclose(fd);
      return -1;
   }
```

```
   continue_with_image:

   /* Start decompression of the JPEG */

   jpeg_start_decompress( &cinfo );

   // allocate the image dimensions to the global variables for other functions
   width = cinfo.output_width;
   height = cinfo.output_height;

   // allocate memory to hold the uncompressed image
   raw_image = (unsigned char*)malloc( cinfo.output_width*cinfo.output_height*cinfo.
       num_components );
   row_pointer[0] = (unsigned char *)malloc( cinfo.output_width*cinfo.num_components
       );

   while( cinfo.output_scanline < cinfo.output_height )
   {
      jpeg_read_scanlines( &cinfo, row_pointer, 1 );
      for( i=0; i<width*cinfo.num_components;i++)
         raw_image[location++] = row_pointer[0][i];
   }

   // wrap up decompression, destroy objects, free pointers and close open files
   jpeg_finish_decompress( &cinfo );

   // unmap the memory
   munmap(c_convert_reg_map_base, MAP_SIZE);
   munmap(idct_reg_map_base, MAP_SIZE);

   fclose(fd);

   jpeg_destroy_decompress( &cinfo );
   free(row_pointer[0]);
   fclose(infile);

   return 1;
}

/* write_jpeg_file
 * This function writes the raw image data stored in the raw_image buffer
 * to a jpeg image with default compression and smoothing options in the file
 * specified by *filename.
 *
 * returns positive integer if successful, -1 otherwise
 *
 */
int write_jpeg_file( char *filename )
{
   struct jpeg_compress_struct cinfo;
   struct jpeg_error_mgr jerr;

   JSAMPROW row_pointer[1];
   FILE *outfile = fopen( filename, "wb" );

   if ( !outfile )
   {
      printf("Error opening output jpeg file %s\n!", filename );
      return -1;
   }
   cinfo.err = jpeg_std_error( &jerr );
   jpeg_create_compress(&cinfo);
   jpeg_stdio_dest(&cinfo, outfile);

   // Setting the parameters of the output file
   cinfo.image_width = width;
   cinfo.image_height = height;
   cinfo.input_components = bytes_per_pixel;
   cinfo.in_color_space = color_space;

   jpeg_set_defaults( &cinfo );

   // Now do the compression ..
   jpeg_start_compress( &cinfo, TRUE );

   while( cinfo.next_scanline < cinfo.image_height )
   {
      row_pointer[0] = &raw_image[ cinfo.next_scanline * cinfo.image_width *  cinfo.
          input_components];
      jpeg_write_scanlines( &cinfo, row_pointer, 1 );
   }
   // Clean up
   jpeg_finish_compress( &cinfo );
   jpeg_destroy_compress( &cinfo );

   fclose(outfile);
   return 1;
}

/* write_to_fb
 * This function writes the raw image to the frame buffer to be displayed on the
     screen
 */
void write_to_fb()
{
    int fbfd = 0;
    struct fb_var_screeninfo vinfo;
    struct fb_fix_screeninfo finfo;
    long int screensize = 0;
    char *fbp = 0;
    int x = 0, y = 0;
    int x_limit;
    long int location = 0;
    long int count =0;
    int xoffset = 0;
    int yoffset =0;
    int crop_offset = 0;

    // Open the file for reading and writing
    fbfd = open("/dev/fb0", O_RDWR);
    if (fbfd == -1) {
        perror("Error: cannot open framebuffer device");
        exit(1);
    }

    // Get fixed screen information
    if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo) == -1) {
        perror("Error reading fixed information");
        exit(2);
    }

    // Get variable screen information
    if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo) == -1) {
        perror("Error reading screen information");
        exit(3);
    }

    screensize = vinfo.xres_virtual * vinfo.yres * vinfo.bits_per_pixel/8;

    // Map the device to memory
    fbp = (char *)mmap(0, screensize, PROT_WRITE, MAP_SHARED, fbfd, 0);
    if ((int)fbp == -1) {
        perror("Error: failed to map framebuffer device to memory");
        exit(4);
    }
   // do any necessary cropping
   if (height>vinfo.yres)
   height = vinfo.yres;

   if(width > vinfo.xres)
   {
      crop_offset = width - vinfo.xres;
      x_limit = vinfo.xres;
   }
   else
      x_limit = width;


   // raw image counter
   count =0;

   for (y=0 ; y < height; y++)
   {
      for (x=0 ; x < x_limit; x++)
      {
         location = (x+vinfo.xoffset) * (vinfo.bits_per_pixel/8) + (y+vinfo.yoffset)
             * finfo.line_length;

         // colour the pixels from the image
         *(fbp + location + 1) = raw_image[count];   // red
         *(fbp + location + 2) = raw_image[count+1]; // green
         *(fbp + location + 3) = raw_image[count+2]; // blue
         *(fbp + location + 0) = 0;        // alpha is ignored

         //increment raw_image pointer
         count +=3;
      }
      count += 3*crop_offset;
   }

    munmap(fbp, screensize);
    close(fbfd);
}
```

ReadImage.c

Appendix D

VHDL Code

D.1 2D IDCT

```
----------------------------------------------------------------------------------
-- Company: University of Windsor
-- Engineer: Dan MacDonald
--
-- Create Date:    02/14/2012
-- Design Name:
-- Module Name:    idct
-- Project Name:   jpeg_idct
-- Target Devices:
-- Tool versions:
-- Description: This is an 8 point 1D-IDCT (row or column) which is used to compute
    the 2D-IDCT for libjpeg
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 1.01 - Edited to allow one idct to do a row and column idct
-- Revision 1.50 - Removed clocking. Asynchronous idct
-- Revision 2.00 - Redesigned to be clocked to improve longest path
-- Revision 2.20 - Increased resolution with constant 'm'
-- Additional Comments:
--
----------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity idct is
       port (
   idct_clk : in std_logic;
   idct_reset : in std_logic;
   idct_done : out std_logic;
   in0: in signed(15 downto 0);
   in1: in signed(15 downto 0);
   in2: in signed(15 downto 0);
   in3: in signed(15 downto 0);
   in4: in signed(15 downto 0);
   in5: in signed(15 downto 0);
   in6: in signed(15 downto 0);
   in7: in signed(15 downto 0);
   out0: out signed(15 downto 0);
   out1: out signed(15 downto 0);
   out2: out signed(15 downto 0);
   out3: out signed(15 downto 0);
   out4: out signed(15 downto 0);
   out5: out signed(15 downto 0);
   out6: out signed(15 downto 0);
   out7: out signed(15 downto 0));

  attribute use_dsp48 : string;
  attribute use_dsp48 of idct : entity is "no";
end idct;

architecture idct_arc of idct is

-------------------------------------------------------------
-- types, signals and constants
-- idct_array = array type used to do intermediate calculations
-------------------------------------------------------------
   constant m : integer := 32;
   type idct_array is array(0 to 7) of signed(m-1 downto 0);
   type state_type is (IDCT_STAGE1, IDCT_STAGE2, IDCT_STAGE3, ADJUSTMENTS);
   signal idct_state    : state_type;

   signal stage1,stage2, stage3, stage4: idct_array;
   signal temp1, temp2: signed(m-1 downto 0);
   signal idct_done2 : std_logic;

begin

-------------------------------------------------------------
------------------- 1D -IDCT process ------------------------
-------------------------------------------------------------


idct_proc: process(in0, in1, in2, in3, in4, in5, in6, in7, idct_clk, idct_reset)

begin
   if (rising_edge(idct_clk)) then
      if(idct_reset = '1') then
         --reset/initial conditions
         idct_done2 <= '0';
         idct_state <= IDCT_STAGE1;
      else

         case(idct_state) is
            when IDCT_STAGE1 =>
               ------------------- stage 1 -------------------
               stage1(2) <= resize((in2 * 139) - (in6 * 335),m);  -- stage1(2) = in2*
                   cos6 - in6*sin6 (<<8)
               stage1(3) <= resize((in2 * 335) + (in6 * 139),m);  -- stage1(3) = in2*
                   sin6 + in6*cos6 (<<8)
```

```
        stage1(4) <= resize(181 * in1 - 181 * in7,m);    --stage1(4) = temp1 -
            temp2 (<<8)
        stage1(7) <= resize(181 * in1 + 181 * in7,m);    --stage1(7) = temp1 +
            temp2 (<<8)

        idct_state <= IDCT_STAGE2;

    when IDCT_STAGE2 =>
        ------------------- stage 2 -------------------

        stage3(0) <= resize( (in0 & x"00") + (in4 & x"00") + stage1(3),m);
            -- stage3(0) = in0(<<8) + in4(<<8) + stage1(3)(<<8)
        stage3(3) <= resize( (in0 & x"00") + (in4 & x"00") - stage1(3),m);
            -- stage3(3) = in0(<<8) + in4(<<8) - stage1(3)(<<8)
        stage3(1) <= resize( (in0 & x"00") - (in4 & x"00") + stage1(2),m);
            -- stage3(1) = in0(<<8) - in4(<<8) + stage1(3)(<<8)
        stage3(2) <= resize( (in0 & x"00") - (in4 & x"00") - stage1(2),m);
            -- stage3(2) = in0(<<8) - in4(<<8) - stage1(3)(<<8)

        stage2(4) <= resize(stage1(4) + (in5 & x"00"), m); -- stage2(4) =
            stage1(4)(<<8) + in5(<<8)
        stage2(6) <= resize(stage1(4) - (in5 & x"00"), m); -- stage2(6) =
            stage1(4)(<<8) - in5(<<8)

        stage2(7) <= resize(stage1(7) + (in3 & x"00"), m); -- stage2(7) =
            stage1(7)(<<8) + in3(<<8)
        stage2(5) <= resize(stage1(7) - (in3 & x"00"), m); -- stage2(5) =
            stage1(7)(<<8) - in3(<<8)

        idct_state <= IDCT_STAGE3;

    when IDCT_STAGE3 =>
        ------------------- stage 3 -------------------

        stage3(4) <= resize((stage2(4)*301) - (stage2(7)*201),m); -- stage3(4)
             = stage2(4)*cos3(<<16) - stage2(7)*sin3(<<16)
        stage3(7) <= resize((stage2(4)*201) + (stage2(7)*301),m); -- stage3(7)
             = stage2(4)*sin3(<<16) + stage2(7)*cos3(<<16)

        stage3(5) <= resize((stage2(5)*355) - (stage2(6)*71),m); -- stage3(5)
            = stage2(3)*cos1(<<16) - stage2(6)*sin1(<<16)
        stage3(6) <= resize((stage2(5)*71) + (stage2(6)*355),m); -- stage3(6)
            = stage2(3)*sin1(<<16) + stage2(6)*cos1(<<16)

        idct_state <= ADJUSTMENTS;

    WHEN ADJUSTMENTS =>
        ----- stage 4 & final adjustments mult by(1/sqrt(8)) ------

        out0 <= resize(signed(shift_right(((stage3(0) & x"00") + stage3(7)) *
            91, 24)),16);
        out1 <= resize(signed(shift_right(((stage3(1) & x"00") + stage3(6)) *
            91, 24)),16);
        out2 <= resize(signed(shift_right(((stage3(2) & x"00") + stage3(5)) *
            91, 24)),16);
        out3 <= resize(signed(shift_right(((stage3(3) & x"00") + stage3(4)) *
            91, 24)),16);
        out4 <= resize(signed(shift_right(((stage3(3) & x"00") - stage3(4)) *
            91, 24)),16);
        out5 <= resize(signed(shift_right(((stage3(2) & x"00") - stage3(5)) *
            91, 24)),16);
        out6 <= resize(signed(shift_right(((stage3(1) & x"00") - stage3(6)) *
            91, 24)),16);
        out7 <= resize(signed(shift_right(((stage3(0) & x"00") - stage3(7)) *
            91, 24)),16);

        idct_done2 <= '1';
        end case;


      end if;  -- end of else reset==0
   end if;  -- end of clk edge


end process idct_proc;

-- a non-clock dependent done signal
DONE_PROC : process(idct_done2, idct_reset) is
begin

   if(idct_reset = '1') then
      idct_done <= '0';
   else
      idct_done <= idct_done2;
   end if;

end process;

end architecture idct_arc;
```

idct.vhd

```
------------------------------------------------------------------------------
-- idct_2d.vhd - entity/architecture pair
------------------------------------------------------------------------------
-- Filename:          idct_2d.vhd
-- Version:           4.00.a
-- Description:       Top level design, instantiates library components and user
    logic.
-- Date:              Mon Jul 23 16:25:13 2012 (by Create and Import Peripheral
    Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

library proc_common_v3_00_a;
use proc_common_v3_00_a.proc_common_pkg.all;
use proc_common_v3_00_a.ipif_pkg.all;

library plbv46_slave_single_v1_01_a;
use plbv46_slave_single_v1_01_a.plbv46_slave_single;

library idct_2d_v4_00_a;
use idct_2d_v4_00_a.user_logic;

entity idct_2d is
  generic
  (
    --USER generics added here

    -- Bus protocol parameters, do not add to or delete
    C_BASEADDR                     : std_logic_vector     := X"FFFFFFFF";
    C_HIGHADDR                     : std_logic_vector     := X"00000000";
    C_SPLB_AWIDTH                  : integer              := 32;
    C_SPLB_DWIDTH                  : integer              := 128;
    C_SPLB_NUM_MASTERS             : integer              := 8;
    C_SPLB_MID_WIDTH               : integer              := 3;
    C_SPLB_NATIVE_DWIDTH           : integer              := 32;
    C_SPLB_P2P                     : integer              := 0;
    C_SPLB_SUPPORT_BURSTS          : integer              := 0;
    C_SPLB_SMALLEST_MASTER         : integer              := 32;
    C_SPLB_CLK_PERIOD_PS            : integer              := 10000;
    C_INCLUDE_DPHASE_TIMER          : integer              := 0;
    C_FAMILY                        : string               := "virtex5"
  );
  port
  (
    --USER ports added here

    -- Bus protocol ports, do not add to or delete
    SPLB_Clk                        : in  std_logic;
    SPLB_Rst                        : in  std_logic;
    PLB_ABus                        : in  std_logic_vector(0 to 31);
    PLB_UABus                       : in  std_logic_vector(0 to 31);
    PLB_PAValid                     : in  std_logic;
    PLB_SAValid                     : in  std_logic;
    PLB_rdPrim                      : in  std_logic;
    PLB_wrPrim                      : in  std_logic;
    PLB_masterID                    : in  std_logic_vector(0 to C_SPLB_MID_WIDTH-1);
    PLB_abort                       : in  std_logic;
    PLB_busLock                     : in  std_logic;
    PLB_RNW                         : in  std_logic;
    PLB_BE                          : in  std_logic_vector(0 to C_SPLB_DWIDTH/8-1);
    PLB_MSize                       : in  std_logic_vector(0 to 1);
    PLB_size                        : in  std_logic_vector(0 to 3);
    PLB_type                        : in  std_logic_vector(0 to 2);
    PLB_lockErr                     : in  std_logic;
    PLB_wrDBus                      : in  std_logic_vector(0 to C_SPLB_DWIDTH-1);
    PLB_wrBurst                     : in  std_logic;
    PLB_rdBurst                     : in  std_logic;
    PLB_wrPendReq                   : in  std_logic;
    PLB_rdPendReq                   : in  std_logic;
    PLB_wrPendPri                   : in  std_logic_vector(0 to 1);
    PLB_rdPendPri                   : in  std_logic_vector(0 to 1);
    PLB_reqPri                      : in  std_logic_vector(0 to 1);
    PLB_TAttribute                  : in  std_logic_vector(0 to 15);
    Sl_addrAck                      : out std_logic;
    Sl_SSize                        : out std_logic_vector(0 to 1);
    Sl_wait                         : out std_logic;
    Sl_rearbitrate                  : out std_logic;
    Sl_wrDAck                       : out std_logic;
    Sl_wrComp                       : out std_logic;
    Sl_wrBTerm                      : out std_logic;
    Sl_rdDBus                       : out std_logic_vector(0 to C_SPLB_DWIDTH-1);
    Sl_rdWdAddr                     : out std_logic_vector(0 to 3);
    Sl_rdDAck                       : out std_logic;
    Sl_rdComp                       : out std_logic;
    Sl_rdBTerm                      : out std_logic;
    Sl_MBusy                        : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1);
    Sl_MWrErr                       : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1);
    Sl_MRdErr                       : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1);
    Sl_MIRQ                         : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1)
  );

  attribute SIGIS : string;
  attribute SIGIS of SPLB_Clk       : signal is "CLK";
  attribute SIGIS of SPLB_Rst       : signal is "RST";

end entity idct_2d;
------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture IMP of idct_2d is
  ------------------------------------------
  -- Array of base/high address pairs for each address range
  ------------------------------------------
  constant ZERO_ADDR_PAD                  : std_logic_vector(0 to 31) := (others =>
      '0');
```

```
constant USER_SLV_BASEADDR              : std_logic_vector     := C_BASEADDR;
constant USER_SLV_HIGHADDR              : std_logic_vector     := C_HIGHADDR;

constant IPIF_ARD_ADDR_RANGE_ARRAY      : SLV64_ARRAY_TYPE     :=
  (
    ZERO_ADDR_PAD & USER_SLV_BASEADDR,  -- user logic slave space base address
    ZERO_ADDR_PAD & USER_SLV_HIGHADDR   -- user logic slave space high address
  );

------------------------------------------
-- Array of desired number of chip enables for each address range
------------------------------------------
constant USER_SLV_NUM_REG               : integer              := 22;
constant USER_NUM_REG                   : integer              := USER_SLV_NUM_REG;

constant IPIF_ARD_NUM_CE_ARRAY          : INTEGER_ARRAY_TYPE   :=
  (
    0  => pad_power2(USER_SLV_NUM_REG)  -- number of ce for user logic slave space
  );

------------------------------------------
-- Ratio of bus clock to core clock (for use in dual clock systems)
-- 1 = ratio is 1:1
-- 2 = ratio is 2:1
------------------------------------------
constant IPIF_BUS2CORE_CLK_RATIO        : integer              := 1;

------------------------------------------
-- Width of the slave data bus (32 only)
------------------------------------------
constant USER_SLV_DWIDTH                : integer              :=
    C_SPLB_NATIVE_DWIDTH;
constant IPIF_SLV_DWIDTH                : integer              :=
    C_SPLB_NATIVE_DWIDTH;

------------------------------------------
-- Index for CS/CE
------------------------------------------
constant USER_SLV_CS_INDEX              : integer              := 0;
constant USER_SLV_CE_INDEX              : integer              :=
    calc_start_ce_index(IPIF_ARD_NUM_CE_ARRAY, USER_SLV_CS_INDEX);
constant USER_CE_INDEX                  : integer              := USER_SLV_CE_INDEX
    ;

------------------------------------------
-- IP Interconnect (IPIC) signal declarations
------------------------------------------
signal ipif_Bus2IP_Clk                : std_logic;
signal ipif_Bus2IP_Reset              : std_logic;
signal ipif_IP2Bus_Data               : std_logic_vector(0 to IPIF_SLV_DWIDTH-1);
signal ipif_IP2Bus_WrAck              : std_logic;
signal ipif_IP2Bus_RdAck              : std_logic;
signal ipif_IP2Bus_Error              : std_logic;
signal ipif_Bus2IP_Addr               : std_logic_vector(0 to C_SPLB_AWIDTH-1);
signal ipif_Bus2IP_Data               : std_logic_vector(0 to IPIF_SLV_DWIDTH-1);
signal ipif_Bus2IP_RNW                : std_logic;
signal ipif_Bus2IP_BE                 : std_logic_vector(0 to IPIF_SLV_DWIDTH/8-1);
signal ipif_Bus2IP_CS                 : std_logic_vector(0 to ((
    IPIF_ARD_ADDR_RANGE_ARRAY'length)/2)-1);
signal ipif_Bus2IP_RdCE               : std_logic_vector(0 to calc_num_ce(
    IPIF_ARD_NUM_CE_ARRAY)-1);
signal ipif_Bus2IP_WrCE               : std_logic_vector(0 to calc_num_ce(
    IPIF_ARD_NUM_CE_ARRAY)-1);
signal user_Bus2IP_RdCE               : std_logic_vector(0 to USER_NUM_REG-1);
signal user_Bus2IP_WrCE               : std_logic_vector(0 to USER_NUM_REG-1);
signal user_IP2Bus_Data               : std_logic_vector(0 to USER_SLV_DWIDTH-1);
signal user_IP2Bus_RdAck              : std_logic;
signal user_IP2Bus_WrAck              : std_logic;
```

```
  signal user_IP2Bus_Error              : std_logic;

  -- IDCT connections
signal idct_out0,    idct_out1,  idct_out2,  idct_out3,  idct_out4,  idct_out5,
    idct_out6,  idct_out7 : signed(15 downto 0);
signal idct_in0,   idct_in1,  idct_in2,  idct_in3,  idct_in4,  idct_in5,  idct_in6,
    idct_in7 : signed( 15 downto 0);
signal idct_reset, idct_done : std_logic;

   component idct
   port (
      idct_clk : in std_logic;
      idct_reset : in std_logic;
      idct_done : out std_logic;
      in0: in signed(15 downto 0);
      in1: in signed(15 downto 0);
      in2: in signed(15 downto 0);
      in3: in signed(15 downto 0);
      in4: in signed(15 downto 0);
      in5: in signed(15 downto 0);
      in6: in signed(15 downto 0);
      in7: in signed(15 downto 0);
      out0: out signed(15 downto 0);
      out1: out signed(15 downto 0);
      out2: out signed(15 downto 0);
      out3: out signed(15 downto 0);
      out4: out signed(15 downto 0);
      out5: out signed(15 downto 0);
      out6: out signed(15 downto 0);
      out7: out signed(15 downto 0)
      );
   end component;
begin

  ------------------------------------------
  -- instantiate plbv46_slave_single
  ------------------------------------------

  PLBV46_SLAVE_SINGLE_I : entity plbv46_slave_single_v1_01_a.plbv46_slave_single
    generic map
    (
      C_ARD_ADDR_RANGE_ARRAY         => IPIF_ARD_ADDR_RANGE_ARRAY,
      C_ARD_NUM_CE_ARRAY             => IPIF_ARD_NUM_CE_ARRAY,
      C_SPLB_P2P                     => C_SPLB_P2P,
      C_BUS2CORE_CLK_RATIO           => IPIF_BUS2CORE_CLK_RATIO,
      C_SPLB_MID_WIDTH               => C_SPLB_MID_WIDTH,
      C_SPLB_NUM_MASTERS             => C_SPLB_NUM_MASTERS,
      C_SPLB_AWIDTH                  => C_SPLB_AWIDTH,
      C_SPLB_DWIDTH                  => C_SPLB_DWIDTH,
      C_SIPIF_DWIDTH                 => IPIF_SLV_DWIDTH,
      C_INCLUDE_DPHASE_TIMER         => C_INCLUDE_DPHASE_TIMER,
      C_FAMILY                       => C_FAMILY
    )
    port map
    (
      SPLB_Clk                       => SPLB_Clk,
      SPLB_Rst                       => SPLB_Rst,
      PLB_ABus                       => PLB_ABus,
      PLB_UABus                      => PLB_UABus,
      PLB_PAValid                    => PLB_PAValid,
      PLB_SAValid                    => PLB_SAValid,
      PLB_rdPrim                     => PLB_rdPrim,
      PLB_wrPrim                     => PLB_wrPrim,
      PLB_masterID                   => PLB_masterID,
      PLB_abort                      => PLB_abort,
      PLB_busLock                    => PLB_busLock,
      PLB_RNW                        => PLB_RNW,
      PLB_BE                         => PLB_BE,
    PLB_MSize                       => PLB_MSize,
    PLB_size                        => PLB_size,
    PLB_type                        => PLB_type,
    PLB_lockErr                     => PLB_lockErr,
    PLB_wrDBus                      => PLB_wrDBus,
    PLB_wrBurst                     => PLB_wrBurst,
    PLB_rdBurst                     => PLB_rdBurst,
    PLB_wrPendReq                   => PLB_wrPendReq,
    PLB_rdPendReq                   => PLB_rdPendReq,
    PLB_wrPendPri                   => PLB_wrPendPri,
    PLB_rdPendPri                   => PLB_rdPendPri,
    PLB_reqPri                      => PLB_reqPri,
    PLB_TAttribute                  => PLB_TAttribute,
    Sl_addrAck                      => Sl_addrAck,
    Sl_SSize                        => Sl_SSize,
    Sl_wait                         => Sl_wait,
    Sl_rearbitrate                  => Sl_rearbitrate,
    Sl_wrDAck                       => Sl_wrDAck,
    Sl_wrComp                       => Sl_wrComp,
    Sl_wrBTerm                      => Sl_wrBTerm,
    Sl_rdDBus                       => Sl_rdDBus,
    Sl_rdWdAddr                     => Sl_rdWdAddr,
    Sl_rdDAck                       => Sl_rdDAck,
    Sl_rdComp                       => Sl_rdComp,
    Sl_rdBTerm                      => Sl_rdBTerm,
    Sl_MBusy                        => Sl_MBusy,
    Sl_MWrErr                       => Sl_MWrErr,
    Sl_MRdErr                       => Sl_MRdErr,
    Sl_MIRQ                         => Sl_MIRQ,
    Bus2IP_Clk                      => ipif_Bus2IP_Clk,
    Bus2IP_Reset                    => ipif_Bus2IP_Reset,
    IP2Bus_Data                     => ipif_IP2Bus_Data,
    IP2Bus_WrAck                    => ipif_IP2Bus_WrAck,
    IP2Bus_RdAck                    => ipif_IP2Bus_RdAck,
    IP2Bus_Error                    => ipif_IP2Bus_Error,
    Bus2IP_Addr                     => ipif_Bus2IP_Addr,
    Bus2IP_Data                     => ipif_Bus2IP_Data,
    Bus2IP_RNW                      => ipif_Bus2IP_RNW,
    Bus2IP_BE                       => ipif_Bus2IP_BE,
    Bus2IP_CS                       => ipif_Bus2IP_CS,
    Bus2IP_RdCE                     => ipif_Bus2IP_RdCE,
    Bus2IP_WrCE                     => ipif_Bus2IP_WrCE
  );

------------------------------------------
-- instantiate User Logic
------------------------------------------

USER_LOGIC_I : entity idct_2d_v4_00_a.user_logic
  generic map
  (
    --USER generics mapped here

    C_SLV_DWIDTH                    => USER_SLV_DWIDTH,
    C_NUM_REG                       => USER_NUM_REG
  )
  port map
  (
 -- idct input ports
 IDCT_IN0  => idct_in0,     IDCT_IN1  => idct_in1,
 IDCT_IN2  => idct_in2,     IDCT_IN3  => idct_in3,
 IDCT_IN4  => idct_in4,     IDCT_IN5  => idct_in5,
 IDCT_IN6  => idct_in6,     IDCT_IN7  => idct_in7,
 -- idct output ports
 IDCT_OUT0  => idct_out0,   IDCT_OUT1  => idct_out1,
 IDCT_OUT2  => idct_out2,   IDCT_OUT3  => idct_out3,
 IDCT_OUT4  => idct_out4,   IDCT_OUT5  => idct_out5,
 IDCT_OUT6  => idct_out6,   IDCT_OUT7  => idct_out7,
   IDCT_RESET => idct_reset,
   IDCT_DONE => idct_done,

      Bus2IP_Clk                      => ipif_Bus2IP_Clk,
      Bus2IP_Reset                    => ipif_Bus2IP_Reset,
      Bus2IP_Data                     => ipif_Bus2IP_Data,
      Bus2IP_BE                       => ipif_Bus2IP_BE,
      Bus2IP_RdCE                     => user_Bus2IP_RdCE,
      Bus2IP_WrCE                     => user_Bus2IP_WrCE,
      IP2Bus_Data                     => user_IP2Bus_Data,
      IP2Bus_RdAck                    => user_IP2Bus_RdAck,
      IP2Bus_WrAck                    => user_IP2Bus_WrAck,
      IP2Bus_Error                    => user_IP2Bus_Error
    );

   idct_I: idct port map(
   in0 => idct_in0,  out0 => idct_out0,
   in1 => idct_in1,  out1 => idct_out1,
   in2 => idct_in2,  out2 => idct_out2,
   in3 => idct_in3,  out3 => idct_out3,
   in4 => idct_in4,  out4 => idct_out4,
   in5 => idct_in5,  out5 => idct_out5,
   in6 => idct_in6,  out6 => idct_out6,
   in7 => idct_in7,  out7 => idct_out7,

   idct_clk => ipif_Bus2IP_Clk,
   idct_reset => idct_reset,
   idct_done => idct_done
   );
  ------------------------------------------
  -- connect internal signals
  ------------------------------------------
  ipif_IP2Bus_Data <= user_IP2Bus_Data;
  ipif_IP2Bus_WrAck <= user_IP2Bus_WrAck;
  ipif_IP2Bus_RdAck <= user_IP2Bus_RdAck;
  ipif_IP2Bus_Error <= user_IP2Bus_Error;

  user_Bus2IP_RdCE <= ipif_Bus2IP_RdCE(USER_CE_INDEX to USER_CE_INDEX+USER_NUM_REG-1)
      ;
  user_Bus2IP_WrCE <= ipif_Bus2IP_WrCE(USER_CE_INDEX to USER_CE_INDEX+USER_NUM_REG-1)
      ;

end IMP;
```

idct_2d.vhd

```
------------------------------------------------------------------------------
-- user_logic.vhd - entity/architecture pair
------------------------------------------------------------------------------
-- Filename:           user_logic.vhd
-- Version:            4.00.a
-- Description:        User logic design. Interface to Software accessible registers,
-- dequantization multiplication, and computation of 16 1D-IDCTs to form the 2D-IDCT
-- Date:               Mon Jul 23 16:25:13 2012 (by Create and Import Peripheral
    Wizard)
-- VHDL Standard:      VHDL'93
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

library proc_common_v3_00_a;
use proc_common_v3_00_a.proc_common_pkg.all;

entity user_logic is
```

```
  generic
  (
    C_SLV_DWIDTH                      : integer              := 32;
    C_NUM_REG                         : integer              := 22
  );
  port
  (
   -- dct i/o ports
   IDCT_IN0, IDCT_IN1, IDCT_IN2, IDCT_IN3, IDCT_IN4, IDCT_IN5, IDCT_IN6, IDCT_IN7 :
       out signed(15 downto 0);
   IDCT_OUT0, IDCT_OUT1, IDCT_OUT2, IDCT_OUT3, IDCT_OUT4, IDCT_OUT5, IDCT_OUT6,
       IDCT_OUT7 : in signed(15 downto 0);
   IDCT_DONE   : in std_logic;
   IDCT_RESET  : out std_logic;

    -- Bus protocol ports, do not add to or delete
    Bus2IP_Clk                        : in  std_logic;
    Bus2IP_Reset                      : in  std_logic;
    Bus2IP_Data                       : in  std_logic_vector(0 to C_SLV_DWIDTH-1);
    Bus2IP_BE                         : in  std_logic_vector(0 to C_SLV_DWIDTH/8-1);
    Bus2IP_RdCE                       : in  std_logic_vector(0 to C_NUM_REG-1);
    Bus2IP_WrCE                       : in  std_logic_vector(0 to C_NUM_REG-1);
    IP2Bus_Data                       : out std_logic_vector(0 to C_SLV_DWIDTH-1);
    IP2Bus_RdAck                      : out std_logic;
    IP2Bus_WrAck                      : out std_logic;
    IP2Bus_Error                      : out std_logic
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );

  attribute SIGIS : string;
  attribute SIGIS of Bus2IP_Clk    : signal is "CLK";
  attribute SIGIS of Bus2IP_Reset  : signal is "RST";

end entity user_logic;

------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture IMP of user_logic is

   -- data type definitions
   type coef_matrix is array(0 to 7, 0 to 7) of signed(0 to 15);  -- 2d array(matrix)
        for holding row idct's
   type short_arr is array(0 to 7) of std_logic_vector(0 to 15);  -- array of 16 bit
       std_logic_vectors
   type int_arr is array(0 to 7) of std_logic_vector(0 to 31); -- array of 32 bit
       std_logic_vectors
   type char_arr is array(0 to 7) of std_logic_vector(0 to 7); -- array of 8 bit
       std_logic_vectors

   signal arr_matrix :coef_matrix;
   signal arr_input  :short_arr;
   signal arr_quant  :int_arr;
   signal arr_output :char_arr;

   -- counter for the rows & for the cols
   signal s_row_cnt  :integer;
   signal s_col_cnt  :integer;
   signal s_trans_cnt   : integer ;
   signal s_reset    :std_logic;
   signal s_lastRow  :std_logic_vector(2 downto 0);
   signal s_done     :std_logic;

   -- post 2d-IDCT range limiting
   signal s_range_lim0 : unsigned(0 to 7);
   signal s_range_lim1 : unsigned(0 to 7);
   signal s_range_lim2 : unsigned(0 to 7);
   signal s_range_lim3 : unsigned(0 to 7);
   signal s_range_lim4 : unsigned(0 to 7);
   signal s_range_lim5 : unsigned(0 to 7);
   signal s_range_lim6 : unsigned(0 to 7);
   signal s_range_lim7 : unsigned(0 to 7);

  ------------------------------------------
  -- Signals for user logic slave model s/w accessible register example
  ------------------------------------------
  signal slv_reg_write_sel              : std_logic_vector(0 to 21);
  signal slv_reg_read_sel               : std_logic_vector(0 to 21);
  signal slv_ip2bus_data                : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_read_ack                   : std_logic;
  signal slv_write_ack                  : std_logic;

begin

  slv_reg_write_sel <= Bus2IP_WrCE(0 to 21);
  slv_reg_read_sel  <= Bus2IP_RdCE(0 to 21);
  slv_write_ack     <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2) or
      Bus2IP_WrCE(3) or Bus2IP_WrCE(4) or Bus2IP_WrCE(5) or Bus2IP_WrCE(6) or
      Bus2IP_WrCE(7) or Bus2IP_WrCE(8) or Bus2IP_WrCE(9) or Bus2IP_WrCE(10) or
      Bus2IP_WrCE(11) or Bus2IP_WrCE(12) or Bus2IP_WrCE(13) or Bus2IP_WrCE(14) or
      Bus2IP_WrCE(15) or Bus2IP_WrCE(16) or Bus2IP_WrCE(17) or Bus2IP_WrCE(18) or
      Bus2IP_WrCE(19) or Bus2IP_WrCE(20) or Bus2IP_WrCE(21);
  slv_read_ack      <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2) or
      Bus2IP_RdCE(3) or Bus2IP_RdCE(4) or Bus2IP_RdCE(5) or Bus2IP_RdCE(6) or
      Bus2IP_RdCE(7) or Bus2IP_RdCE(8) or Bus2IP_RdCE(9) or Bus2IP_RdCE(10) or
      Bus2IP_RdCE(11) or Bus2IP_RdCE(12) or Bus2IP_RdCE(13) or Bus2IP_RdCE(14) or
      Bus2IP_RdCE(15) or Bus2IP_RdCE(16) or Bus2IP_RdCE(17) or Bus2IP_RdCE(18) or
      Bus2IP_RdCE(19) or Bus2IP_RdCE(20) or Bus2IP_RdCE(21);

   ---------------------------------------
   -- Register Write Process
   ---------------------------------------

    SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is

     begin

    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
      if Bus2IP_Reset = '1' then
         arr_output(0)  <= (others => '0');
         arr_output(1)  <= (others => '0');
         arr_output(2)  <= (others => '0');
         arr_output(3)  <= (others => '0');
         arr_output(4)  <= (others => '0');
         arr_output(5)  <= (others => '0');
         arr_output(6)  <= (others => '0');
         arr_output(7)  <= (others => '0');

         arr_quant(0)  <= (others => '0');
         arr_quant(1)  <= (others => '0');
         arr_quant(2)  <= (others => '0');
         arr_quant(3)  <= (others => '0');
         arr_quant(4)  <= (others => '0');
         arr_quant(5)  <= (others => '0');
         arr_quant(6)  <= (others => '0');
         arr_quant(7)  <= (others => '0');

         arr_input(0)  <= (others => '0');
         arr_input(1)  <= (others => '0');
         arr_input(2)  <= (others => '0');
         arr_input(3)  <= (others => '0');
         arr_input(4)  <= (others => '0');
         arr_input(5)  <= (others => '0');
         arr_input(6)  <= (others => '0');
         arr_input(7)  <= (others => '0');
```

```
        s_row_cnt <= -1;
        s_reset <= '0';
        s_trans_cnt <= 0;

        s_lastRow <= (others => '0');
        s_done <= '0';
        -- 1D IDCT initial conditions
        IDCT_RESET <= '1';

     else

  case slv_reg_write_sel is
     when "100000000000000000000" => arr_quant(0) <= Bus2IP_Data(0 to C_SLV_DWIDTH
         -1);
          if(s_col_cnt > 6 ) then
             s_reset <= '1';
             s_row_cnt <= -1;
             s_trans_cnt <= 0;
          else
             s_reset <= '0';
          end if;

          s_lastRow <= (others => '0');

     when "010000000000000000000" => arr_quant(1) <= Bus2IP_Data(0 to C_SLV_DWIDTH
         -1);
     when "001000000000000000000" => arr_quant(2) <= Bus2IP_Data(0 to C_SLV_DWIDTH
         -1);
     when "000100000000000000000" => arr_quant(3) <= Bus2IP_Data(0 to C_SLV_DWIDTH
         -1);
     when "000010000000000000000" => arr_quant(4) <= Bus2IP_Data(0 to C_SLV_DWIDTH
         -1);
     when "000001000000000000000" => arr_quant(5) <= Bus2IP_Data(0 to C_SLV_DWIDTH
         -1);
     when "000000100000000000000" => arr_quant(6) <= Bus2IP_Data(0 to C_SLV_DWIDTH
         -1);
     when "000000010000000000000" => arr_quant(7) <= Bus2IP_Data(0 to C_SLV_DWIDTH
         -1);
     when "000000001000000000000" =>
        arr_input(0) <= Bus2IP_Data(0 to 15);
        arr_input(1) <= Bus2IP_Data(16 to 31);
     when "000000000100000000000" =>
        arr_input(2) <= Bus2IP_Data(0 to 15);
        arr_input(3) <= Bus2IP_Data(16 to 31);
     when "000000000010000000000" =>
        arr_input(4) <= Bus2IP_Data(0 to 15);
        arr_input(5) <= Bus2IP_Data(16 to 31);
     when "000000000001000000000" =>

        IDCT_IN0 <= resize(signed(arr_input(0)) * signed(arr_quant(0)),16)(15 downto
             0);
        IDCT_IN1 <= resize(signed(arr_input(1)) * signed(arr_quant(1)),16)(15 downto
             0);
        IDCT_IN2 <= resize(signed(arr_input(2)) * signed(arr_quant(2)),16)(15 downto
             0);
        IDCT_IN3 <= resize(signed(arr_input(3)) * signed(arr_quant(3)),16)(15 downto
             0);
        IDCT_IN4 <= resize(signed(arr_input(4)) * signed(arr_quant(4)),16)(15 downto
             0);
        IDCT_IN5 <= resize(signed(arr_input(5)) * signed(arr_quant(5)),16)(15 downto
             0);
        IDCT_IN6 <= resize(signed(Bus2IP_Data(0 to 15)) * signed(arr_quant(6)),16)
            (15 downto 0);
        IDCT_IN7 <= resize(signed(Bus2IP_Data(16 to 31)) * signed(arr_quant(7)),16)
            (15 downto 0);

        if(IDCT_DONE = '0') then
```

```
            IDCT_RESET <= '0';

            -- increment row counter when the idct is ready
            s_row_cnt <= s_row_cnt +1;
         end if;

      when others => null;
   end case;

   -- control the output of the idct
   if(s_row_cnt > -1 and s_row_cnt <8) then

      if(s_col_cnt = 0 and s_lastRow < "111") then
         s_done <= '0';

         if(IDCT_DONE = '1') then
            -- read the outputs if its ready
            arr_matrix(s_row_cnt,0) <= IDCT_OUT0;
            arr_matrix(s_row_cnt,1) <= IDCT_OUT1;
            arr_matrix(s_row_cnt,2) <= IDCT_OUT2;
            arr_matrix(s_row_cnt,3) <= IDCT_OUT3;
            arr_matrix(s_row_cnt,4) <= IDCT_OUT4;
            arr_matrix(s_row_cnt,5) <= IDCT_OUT5;
            arr_matrix(s_row_cnt,6) <= IDCT_OUT6;
            arr_matrix(s_row_cnt,7) <= IDCT_OUT7;

            -- reset the 1D-IDCT
            IDCT_RESET <= '1';
         end if;

      elsif(s_trans_cnt < 8 ) then

         if(IDCT_DONE = '1') then
            -- store the column idcts into the matrix
            arr_matrix(0, s_trans_cnt) <= IDCT_OUT0;
            arr_matrix(1, s_trans_cnt) <= IDCT_OUT1;
            arr_matrix(2, s_trans_cnt) <= IDCT_OUT2;
            arr_matrix(3, s_trans_cnt) <= IDCT_OUT3;
            arr_matrix(4, s_trans_cnt) <= IDCT_OUT4;
            arr_matrix(5, s_trans_cnt) <= IDCT_OUT5;
            arr_matrix(6, s_trans_cnt) <= IDCT_OUT6;
            arr_matrix(7, s_trans_cnt) <= IDCT_OUT7;

            s_trans_cnt <= s_trans_cnt + 1;

            -- reset the 1D-IDCT
            IDCT_RESET <= '1';

         end if;
      end if;
   end if;

   ------ column delay from lastrow ------
   if (s_row_cnt = 7) then
      if(s_lastRow = "111" and s_trans_cnt < 8) then

         if(IDCT_DONE = '0') then
            IDCT_RESET <= '0';   -- start the idct

            IDCT_IN0 <= arr_matrix(0,s_trans_cnt);
            IDCT_IN1 <= arr_matrix(1,s_trans_cnt);
            IDCT_IN2 <= arr_matrix(2,s_trans_cnt);
            IDCT_IN3 <= arr_matrix(3,s_trans_cnt);
            IDCT_IN4 <= arr_matrix(4,s_trans_cnt);
            IDCT_IN5 <= arr_matrix(5,s_trans_cnt);
            IDCT_IN6 <= arr_matrix(6,s_trans_cnt);
            IDCT_IN7 <= arr_matrix(7,s_trans_cnt);
```

```
        end if;  -- end of idct_done =0

     elsif (s_trans_cnt = 8) then
        s_done <= '1';
     else
        s_lastRow <= s_lastRow + "001";
     end if;

  end if;

  -------------- range limiting -----------------
  if (s_col_cnt <8 ) then -- the user shouldn't start reading until s_done = 1

     if(arr_matrix(s_col_cnt,0) < -128) then arr_output(0) <= x"00";
     elsif(arr_matrix(s_col_cnt,0) > 127) then arr_output(0) <= x"FF";
     else arr_output(0) <= std_logic_vector(resize(unsigned(arr_matrix(s_col_cnt,0)
         + x"0080"),8));
     end if;

     if(arr_matrix(s_col_cnt,1) < -128) then arr_output(1) <= x"00";
     elsif(arr_matrix(s_col_cnt,1) > 127) then arr_output(1) <= x"FF";
     else arr_output(1) <= std_logic_vector(resize(unsigned(arr_matrix(s_col_cnt,1)
         + x"0080"),8));
     end if;

     if(arr_matrix(s_col_cnt,2) < -128) then arr_output(2) <= x"00";
     elsif(arr_matrix(s_col_cnt,2) > 127) then arr_output(2) <= x"FF";
     else arr_output(2) <= std_logic_vector(resize(unsigned(arr_matrix(s_col_cnt,2)
         + x"0080"),8));
     end if;

     if(arr_matrix(s_col_cnt,3) < -128) then arr_output(3) <= x"00";
     elsif(arr_matrix(s_col_cnt,3) > 127) then arr_output(3) <= x"FF";
     else arr_output(3) <= std_logic_vector(resize(unsigned(arr_matrix(s_col_cnt,3)
         + x"0080"),8));
     end if;

     if(arr_matrix(s_col_cnt,4) < -128) then arr_output(4) <= x"00";
     elsif(arr_matrix(s_col_cnt,4) > 127) then arr_output(4) <= x"FF";
     else arr_output(4) <= std_logic_vector(resize(unsigned(arr_matrix(s_col_cnt,4)
         + x"0080"),8));
     end if;

     if(arr_matrix(s_col_cnt,5) < -128) then arr_output(5) <= x"00";
     elsif(arr_matrix(s_col_cnt,5) > 127) then arr_output(5) <= x"FF";
     else arr_output(5) <= std_logic_vector(resize(unsigned(arr_matrix(s_col_cnt,5)
         + x"0080"),8));
     end if;

     if(arr_matrix(s_col_cnt,6) < -128) then arr_output(6) <= x"00";
     elsif(arr_matrix(s_col_cnt,6) > 127) then arr_output(6) <= x"FF";
     else arr_output(6) <= std_logic_vector(resize(unsigned(arr_matrix(s_col_cnt,6)
         + x"0080"),8));
     end if;

     if(arr_matrix(s_col_cnt,7) < -128) then arr_output(7) <= x"00";
     elsif(arr_matrix(s_col_cnt,7) > 127) then arr_output(7) <= x"FF";
     else arr_output(7) <= std_logic_vector(resize(unsigned(arr_matrix(s_col_cnt,7)
         + x"0080"),8));
     end if;
  end if;
     end if; -- end of clk event
   end if; -- end of reset
 end process SLAVE_REG_WRITE_PROC;


  ---------------------------------------
  -- Register Read Process
  --------------------------------------

 SLAVE_REG_READ_PROC : process(Bus2IP_Clk, slv_reg_read_sel, arr_quant(0), arr_quant
     (1), arr_quant(2), arr_quant(3), arr_quant(4), arr_quant(5), arr_quant(6),
     arr_quant(7), arr_input(0),arr_input(1),arr_input(2),arr_input(3),arr_input(4),
     arr_input(5),arr_input(6),arr_input(7), arr_output(0), arr_output(1), arr_output
     (2), arr_output(3), arr_output(4), arr_output(5), arr_output(6), arr_output(7) )
      is
  begin
    if Bus2IP_Clk'event and Bus2IP_Clk = '0' then
   if Bus2IP_Reset = '1' then
      s_col_cnt <= 0;

   else
       case slv_reg_read_sel is
         when "0000000010000000000000" => slv_ip2bus_data <= arr_input(0) & arr_input
             (1);       -- inptr(0) & inptr(1)
         when "0000000001000000000000" => slv_ip2bus_data <= arr_input(2) & arr_input
             (3);       -- inptr(2) & inptr(3)
         when "0000000000100000000000" => slv_ip2bus_data <= arr_input(4) & arr_input
             (5);       -- inptr(4) & inptr(5)
         when "0000000000010000000000" => slv_ip2bus_data <= arr_input(6) & arr_input
             (7);       -- inptr(6) & inptr(7)
         when "0000000000001000000000" => slv_ip2bus_data <= x"00000000";
                -- Not used
         when "0000000000000100000000" =>
         slv_ip2bus_data <= arr_output(0) & arr_output(1) & arr_output(2) &
             arr_output(3);   -- outptr(0-3)
         when "0000000000000010000000" =>
         slv_ip2bus_data <= arr_output(4) & arr_output(5) & arr_output(6) &
             arr_output(7);   -- outptr(4-7)

         -- update the col count
         s_col_cnt <= s_col_cnt + 1;
         when "0000000000000000000001" => slv_ip2bus_data <= "000000000000000000000"
             & s_done;
         when others => slv_ip2bus_data <= (others => '0');
       end case;

      -- reset the column counter
      if (s_reset = '1') then
         s_col_cnt <= 0;
      end if;
   end if;

     end if;   -- end of clk event

  end process SLAVE_REG_READ_PROC;


  ------------------------------------------
  -- Example code to drive IP to Bus signals
  ------------------------------------------
  IP2Bus_Data  <= slv_ip2bus_data when slv_read_ack = '1' else
                  (others => '0');

  IP2Bus_WrAck <= slv_write_ack;
  IP2Bus_RdAck <= slv_read_ack;
  IP2Bus_Error <= '0';

end IMP;
```

user_logic_idct.vhd

D.2 Colour Converter

```
------------------------------------------------------------------------------
-- c_converter.vhd - entity/architecture pair
------------------------------------------------------------------------------
-- Filename:          c_converter.vhd
-- Version:           3.00.b
-- Description:       Top level design, instantiates library components and user
    logic.
-- Date:              Thu Aug 16 14:31:08 2012 (by Create and Import Peripheral
    Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v3_00_a;
use proc_common_v3_00_a.proc_common_pkg.all;
use proc_common_v3_00_a.ipif_pkg.all;

library plbv46_slave_single_v1_01_a;
use plbv46_slave_single_v1_01_a.plbv46_slave_single;

library c_converter_v3_00_b;
use c_converter_v3_00_b.user_logic;

entity c_converter is
  generic
  (
    --USER generics added here

    -- Bus protocol parameters, do not add to or delete
    C_BASEADDR                     : std_logic_vector     := X"FFFFFFFF";
    C_HIGHADDR                     : std_logic_vector     := X"00000000";
    C_SPLB_AWIDTH                  : integer              := 32;
    C_SPLB_DWIDTH                  : integer              := 128;
    C_SPLB_NUM_MASTERS             : integer              := 8;
    C_SPLB_MID_WIDTH               : integer              := 3;
    C_SPLB_NATIVE_DWIDTH           : integer              := 32;
    C_SPLB_P2P                     : integer              := 0;
    C_SPLB_SUPPORT_BURSTS          : integer              := 0;
    C_SPLB_SMALLEST_MASTER         : integer              := 32;
    C_SPLB_CLK_PERIOD_PS           : integer              := 10000;
    C_INCLUDE_DPHASE_TIMER         : integer              := 0;
    C_FAMILY                       : string               := "virtex5"
  );
  port
  (
    --USER ports added here

    -- Bus protocol ports, do not add to or delete
    SPLB_Clk                       : in  std_logic;
    SPLB_Rst                       : in  std_logic;
    PLB_ABus                       : in  std_logic_vector(0 to 31);
    PLB_UABus                      : in  std_logic_vector(0 to 31);
    PLB_PAValid                    : in  std_logic;
    PLB_SAValid                    : in  std_logic;
    PLB_rdPrim                     : in  std_logic;
    PLB_wrPrim                     : in  std_logic;
    PLB_masterID                   : in  std_logic_vector(0 to C_SPLB_MID_WIDTH-1);
    PLB_abort                      : in  std_logic;
    PLB_busLock                    : in  std_logic;
    PLB_RNW                        : in  std_logic;
    PLB_BE                         : in  std_logic_vector(0 to C_SPLB_DWIDTH/8-1);
    PLB_MSize                      : in  std_logic_vector(0 to 1);
    PLB_size                      : in  std_logic_vector(0 to 3);
    PLB_type                      : in  std_logic_vector(0 to 2);
    PLB_lockErr                   : in  std_logic;
    PLB_wrDBus                    : in  std_logic_vector(0 to C_SPLB_DWIDTH-1);
    PLB_wrBurst                   : in  std_logic;
    PLB_rdBurst                   : in  std_logic;
    PLB_wrPendReq                 : in  std_logic;
    PLB_rdPendReq                 : in  std_logic;
    PLB_wrPendPri                 : in  std_logic_vector(0 to 1);
    PLB_rdPendPri                 : in  std_logic_vector(0 to 1);
    PLB_reqPri                    : in  std_logic_vector(0 to 1);
    PLB_TAttribute                : in  std_logic_vector(0 to 15);
    Sl_addrAck                    : out std_logic;
    Sl_SSize                      : out std_logic_vector(0 to 1);
    Sl_wait                       : out std_logic;
    Sl_rearbitrate                : out std_logic;
    Sl_wrDAck                     : out std_logic;
    Sl_wrComp                     : out std_logic;
    Sl_wrBTerm                    : out std_logic;
    Sl_rdDBus                     : out std_logic_vector(0 to C_SPLB_DWIDTH-1);
    Sl_rdWdAddr                   : out std_logic_vector(0 to 3);
    Sl_rdDAck                     : out std_logic;
    Sl_rdComp                     : out std_logic;
    Sl_rdBTerm                    : out std_logic;
    Sl_MBusy                      : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1);
    Sl_MWrErr                     : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1);
    Sl_MRdErr                     : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1);
    Sl_MIRQ                       : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1)
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );

  attribute SIGIS : string;
  attribute SIGIS of SPLB_Clk       : signal is "CLK";
  attribute SIGIS of SPLB_Rst       : signal is "RST";

end entity c_converter;

------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture IMP of c_converter is

  ------------------------------------------
  -- Array of base/high address pairs for each address range
  ------------------------------------------
  constant ZERO_ADDR_PAD                  : std_logic_vector(0 to 31) := (others =>
      '0');
  constant USER_SLV_BASEADDR              : std_logic_vector     := C_BASEADDR;
  constant USER_SLV_HIGHADDR              : std_logic_vector     := C_HIGHADDR;

  constant IPIF_ARD_ADDR_RANGE_ARRAY      : SLV64_ARRAY_TYPE     :=
    (
      ZERO_ADDR_PAD & USER_SLV_BASEADDR,  -- user logic slave space base address
      ZERO_ADDR_PAD & USER_SLV_HIGHADDR   -- user logic slave space high address
    );

  ------------------------------------------
  -- Array of desired number of chip enables for each address range
  ------------------------------------------
  constant USER_SLV_NUM_REG               : integer              := 6;
  constant USER_NUM_REG                   : integer              := USER_SLV_NUM_REG;

  constant IPIF_ARD_NUM_CE_ARRAY          : INTEGER_ARRAY_TYPE   :=
    (
      0  => pad_power2(USER_SLV_NUM_REG)  -- number of ce for user logic slave space
    );
  ------------------------------------------
  -- Ratio of bus clock to core clock (for use in dual clock systems)
  -- 1 = ratio is 1:1
  -- 2 = ratio is 2:1
  ------------------------------------------
  constant IPIF_BUS2CORE_CLK_RATIO        : integer              := 1;

  ------------------------------------------
  -- Width of the slave data bus (32 only)
  ------------------------------------------
  constant USER_SLV_DWIDTH                : integer              :=
      C_SPLB_NATIVE_DWIDTH;

  constant IPIF_SLV_DWIDTH                : integer              :=
      C_SPLB_NATIVE_DWIDTH;

  ------------------------------------------
  -- Index for CS/CE
  ------------------------------------------
  constant USER_SLV_CS_INDEX              : integer              := 0;
  constant USER_SLV_CE_INDEX              : integer              :=
      calc_start_ce_index(IPIF_ARD_NUM_CE_ARRAY, USER_SLV_CS_INDEX);

  constant USER_CE_INDEX                  : integer              := USER_SLV_CE_INDEX
      ;

  ------------------------------------------
  -- IP Interconnect (IPIC) signal declarations
  ------------------------------------------
  signal ipif_Bus2IP_Clk                : std_logic;
  signal ipif_Bus2IP_Reset              : std_logic;
  signal ipif_IP2Bus_Data               : std_logic_vector(0 to IPIF_SLV_DWIDTH-1);
  signal ipif_IP2Bus_WrAck              : std_logic;
  signal ipif_IP2Bus_RdAck              : std_logic;
  signal ipif_IP2Bus_Error              : std_logic;
  signal ipif_Bus2IP_Addr               : std_logic_vector(0 to C_SPLB_AWIDTH-1);
  signal ipif_Bus2IP_Data               : std_logic_vector(0 to IPIF_SLV_DWIDTH-1);
  signal ipif_Bus2IP_RNW                : std_logic;
  signal ipif_Bus2IP_BE                 : std_logic_vector(0 to IPIF_SLV_DWIDTH/8-1);
  signal ipif_Bus2IP_CS                 : std_logic_vector(0 to ((
      IPIF_ARD_ADDR_RANGE_ARRAY'length)/2)-1);
  signal ipif_Bus2IP_RdCE               : std_logic_vector(0 to calc_num_ce(
      IPIF_ARD_NUM_CE_ARRAY)-1);
  signal ipif_Bus2IP_WrCE               : std_logic_vector(0 to calc_num_ce(
      IPIF_ARD_NUM_CE_ARRAY)-1);
  signal user_Bus2IP_RdCE               : std_logic_vector(0 to USER_NUM_REG-1);
  signal user_Bus2IP_WrCE               : std_logic_vector(0 to USER_NUM_REG-1);
  signal user_IP2Bus_Data               : std_logic_vector(0 to USER_SLV_DWIDTH-1);
  signal user_IP2Bus_RdAck              : std_logic;
  signal user_IP2Bus_WrAck              : std_logic;
  signal user_IP2Bus_Error              : std_logic;

begin

  ------------------------------------------
  -- instantiate plbv46_slave_single
  ------------------------------------------
  PLBV46_SLAVE_SINGLE_I : entity plbv46_slave_single_v1_01_a.plbv46_slave_single
    generic map
    (
      C_ARD_ADDR_RANGE_ARRAY          => IPIF_ARD_ADDR_RANGE_ARRAY,
      C_ARD_NUM_CE_ARRAY              => IPIF_ARD_NUM_CE_ARRAY,
      C_SPLB_P2P                      => C_SPLB_P2P,
      C_BUS2CORE_CLK_RATIO            => IPIF_BUS2CORE_CLK_RATIO,
      C_SPLB_MID_WIDTH                => C_SPLB_MID_WIDTH,
      C_SPLB_NUM_MASTERS              => C_SPLB_NUM_MASTERS,
      C_SPLB_AWIDTH                   => C_SPLB_AWIDTH,
      C_SPLB_DWIDTH                   => C_SPLB_DWIDTH,
      C_SIPIF_DWIDTH                => IPIF_SLV_DWIDTH,
      C_INCLUDE_DPHASE_TIMER        => C_INCLUDE_DPHASE_TIMER,
      C_FAMILY                      => C_FAMILY
    )
    port map
    (
      SPLB_Clk                      => SPLB_Clk,
      SPLB_Rst                      => SPLB_Rst,
      PLB_ABus                      => PLB_ABus,
      PLB_UABus                     => PLB_UABus,
      PLB_PAValid                   => PLB_PAValid,
      PLB_SAValid                   => PLB_SAValid,
      PLB_rdPrim                    => PLB_rdPrim,
      PLB_wrPrim                    => PLB_wrPrim,
      PLB_masterID                  => PLB_masterID,
      PLB_abort                     => PLB_abort,
      PLB_busLock                   => PLB_busLock,
      PLB_RNW                       => PLB_RNW,
      PLB_BE                        => PLB_BE,
      PLB_MSize                     => PLB_MSize,
      PLB_size                      => PLB_size,
      PLB_type                      => PLB_type,
      PLB_lockErr                   => PLB_lockErr,
      PLB_wrDBus                    => PLB_wrDBus,
      PLB_wrBurst                   => PLB_wrBurst,
      PLB_rdBurst                   => PLB_rdBurst,
      PLB_wrPendReq                 => PLB_wrPendReq,
      PLB_rdPendReq                 => PLB_rdPendReq,
      PLB_wrPendPri                 => PLB_wrPendPri,
      PLB_rdPendPri                 => PLB_rdPendPri,
      PLB_reqPri                    => PLB_reqPri,
      PLB_TAttribute                => PLB_TAttribute,
      Sl_addrAck                    => Sl_addrAck,
      Sl_SSize                      => Sl_SSize,
      Sl_wait                       => Sl_wait,
      Sl_rearbitrate                => Sl_rearbitrate,
      Sl_wrDAck                     => Sl_wrDAck,
      Sl_wrComp                     => Sl_wrComp,
      Sl_wrBTerm                    => Sl_wrBTerm,
      Sl_rdDBus                     => Sl_rdDBus,
      Sl_rdWdAddr                   => Sl_rdWdAddr,
      Sl_rdDAck                     => Sl_rdDAck,
      Sl_rdComp                     => Sl_rdComp,
      Sl_rdBTerm                    => Sl_rdBTerm,
      Sl_MBusy                      => Sl_MBusy,
      Sl_MWrErr                     => Sl_MWrErr,
      Sl_MRdErr                     => Sl_MRdErr,
      Sl_MIRQ                       => Sl_MIRQ,
      Bus2IP_Clk                    => ipif_Bus2IP_Clk,
      Bus2IP_Reset                  => ipif_Bus2IP_Reset,
      IP2Bus_Data                   => ipif_IP2Bus_Data,
      IP2Bus_WrAck                  => ipif_IP2Bus_WrAck,
      IP2Bus_RdAck                  => ipif_IP2Bus_RdAck,
      IP2Bus_Error                  => ipif_IP2Bus_Error,
      Bus2IP_Addr                   => ipif_Bus2IP_Addr,
      Bus2IP_Data                   => ipif_Bus2IP_Data,
      Bus2IP_RNW                    => ipif_Bus2IP_RNW,
      Bus2IP_BE                     => ipif_Bus2IP_BE,
      Bus2IP_CS                     => ipif_Bus2IP_CS,
      Bus2IP_RdCE                   => ipif_Bus2IP_RdCE,
      Bus2IP_WrCE                   => ipif_Bus2IP_WrCE
    );

  ------------------------------------------
  -- instantiate User Logic
  ------------------------------------------
  USER_LOGIC_I : entity c_converter_v3_00_b.user_logic
    generic map
```

```
    (
      -- MAP USER GENERICS BELOW THIS LINE ---------------
      --USER generics mapped here
      -- MAP USER GENERICS ABOVE THIS LINE ---------------

      C_SLV_DWIDTH                    => USER_SLV_DWIDTH,
      C_NUM_REG                       => USER_NUM_REG
    )
    port map
    (
      -- MAP USER PORTS BELOW THIS LINE ------------------
      --USER ports mapped here
      -- MAP USER PORTS ABOVE THIS LINE ------------------

      Bus2IP_Clk                      => ipif_Bus2IP_Clk,
      Bus2IP_Reset                    => ipif_Bus2IP_Reset,
      Bus2IP_Data                     => ipif_Bus2IP_Data,
      Bus2IP_BE                       => ipif_Bus2IP_BE,
      Bus2IP_RdCE                     => user_Bus2IP_RdCE,
      Bus2IP_WrCE                     => user_Bus2IP_WrCE,
      IP2Bus_Data                     => user_IP2Bus_Data,
      IP2Bus_RdAck                    => user_IP2Bus_RdAck,
      IP2Bus_WrAck                    => user_IP2Bus_WrAck,
      IP2Bus_Error                    => user_IP2Bus_Error
    );

  ------------------------------------------
  -- connect internal signals
  ------------------------------------------
  ipif_IP2Bus_Data <= user_IP2Bus_Data;
  ipif_IP2Bus_WrAck <= user_IP2Bus_WrAck;
  ipif_IP2Bus_RdAck <= user_IP2Bus_RdAck;
  ipif_IP2Bus_Error <= user_IP2Bus_Error;

  user_Bus2IP_RdCE <= ipif_Bus2IP_RdCE(USER_CE_INDEX to USER_CE_INDEX+USER_NUM_REG-1)
      ;
  user_Bus2IP_WrCE <= ipif_Bus2IP_WrCE(USER_CE_INDEX to USER_CE_INDEX+USER_NUM_REG-1)
      ;

end IMP;
```

c_converter.vhd

```
------------------------------------------------------------------------------
-- user_logic.vhd - entity/architecture pair
------------------------------------------------------------------------------
-- Filename:          user_logic.vhd
-- Version:           3.00.b
-- Description:       User logic design. Interface to Software accessible registers,
-- and computation of RGB from YCC input of 4 pixels
-- Date:              Thu Aug 16 14:31:08 2012 (by Create and Import Peripheral
    Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

library proc_common_v3_00_a;
use proc_common_v3_00_a.proc_common_pkg.all;

entity user_logic is
  generic
  (
    C_SLV_DWIDTH                   : integer              := 32;
    C_NUM_REG                      : integer              := 6
  );
  port
  (
    Bus2IP_Clk                       : in  std_logic;
    Bus2IP_Reset                     : in  std_logic;
    Bus2IP_Data                      : in  std_logic_vector(0 to C_SLV_DWIDTH-1);
    Bus2IP_BE                        : in  std_logic_vector(0 to C_SLV_DWIDTH/8-1);
    Bus2IP_RdCE                      : in  std_logic_vector(0 to C_NUM_REG-1);
    Bus2IP_WrCE                      : in  std_logic_vector(0 to C_NUM_REG-1);
    IP2Bus_Data                      : out std_logic_vector(0 to C_SLV_DWIDTH-1);
    IP2Bus_RdAck                     : out std_logic;
    IP2Bus_WrAck                     : out std_logic;
    IP2Bus_Error                     : out std_logic
  );

  attribute SIGIS : string;
  attribute SIGIS of Bus2IP_Clk    : signal is "CLK";
  attribute SIGIS of Bus2IP_Reset  : signal is "RST";

  -- use DSP48 slices for speed
  attribute use_dsp48 : string;
  attribute use_dsp48 of user_logic : entity is "yes";
end entity user_logic;

------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture IMP of user_logic is
   -- data types and constants
   type state_type is (IDLE,YCC_SETUP,YCC0, YCC1, YCC2, YCC3);

   constant CrR : signed(23 downto 0)      := x"0166E9"; -- 1.402 << 16
   constant CbB : signed(23 downto 0)      := x"01C5A2"; -- 1.772 << 16
   constant CrG : signed(23 downto 0)      := x"00B6D2"; -- 0.71414 << 16
   constant CbG : signed(23 downto 0)      := x"00581A"; -- 0.34414 << 16
   constant ONE_HALF : signed(23 downto 0)      := x"008000"; -- "fudge factor" to
       assist rounding
   signal convert_state : state_type;

  ------------------------------------------
  -- Signals for user logic slave model s/w accessible register example
  ------------------------------------------
     signal s_Y0   : std_logic_vector(0 to 7);
     signal s_Y1   : std_logic_vector(0 to 7);
     signal s_Y2   : std_logic_vector(0 to 7);
     signal s_Y3   : std_logic_vector(0 to 7);

     signal s_Cb0  : std_logic_vector(0 to 7);
     signal s_Cb1  : std_logic_vector(0 to 7);
     signal s_Cb2  : std_logic_vector(0 to 7);
     signal s_Cb3  : std_logic_vector(0 to 7);

     signal s_Cr0  : std_logic_vector(0 to 7);
     signal s_Cr1  : std_logic_vector(0 to 7);
     signal s_Cr2  : std_logic_vector(0 to 7);
     signal s_Cr3  : std_logic_vector(0 to 7);

     signal s_R0   : std_logic_vector(0 to 7);
     signal s_R1   : std_logic_vector(0 to 7);
     signal s_R2   : std_logic_vector(0 to 7);
     signal s_R3   : std_logic_vector(0 to 7);

     signal s_G0   : std_logic_vector(0 to 7);
     signal s_G1   : std_logic_vector(0 to 7);
     signal s_G2   : std_logic_vector(0 to 7);
     signal s_G3   : std_logic_vector(0 to 7);
     signal s_B0   : std_logic_vector(0 to 7);
     signal s_B1   : std_logic_vector(0 to 7);
     signal s_B2   : std_logic_vector(0 to 7);
     signal s_B3   : std_logic_vector(0 to 7);

     signal slv_reg_write_sel              : std_logic_vector(0 to 5);
     signal slv_reg_read_sel               : std_logic_vector(0 to 5);
     signal slv_ip2bus_data                : std_logic_vector(0 to C_SLV_DWIDTH-1);
     signal slv_read_ack                   : std_logic;
     signal slv_write_ack                  : std_logic;

begin

  slv_reg_write_sel <= Bus2IP_WrCE(0 to 5);
  slv_reg_read_sel  <= Bus2IP_RdCE(0 to 5);
  slv_write_ack     <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2) or
      Bus2IP_WrCE(3) or Bus2IP_WrCE(4) or Bus2IP_WrCE(5);
  slv_read_ack      <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2) or
      Bus2IP_RdCE(3) or Bus2IP_RdCE(4) or Bus2IP_RdCE(5);

  -- implement slave model software accessible register(s)
  SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is

   variable s_Y      : std_logic_vector(0 to 7);
   variable s_Cb     : std_logic_vector(0 to 7);
   variable s_Cr     : std_logic_vector(0 to 7);
   variable s_R      : std_logic_vector(0 to 7);
   variable s_G      : std_logic_vector(0 to 7);
   variable s_B      : std_logic_vector(0 to 7);
   variable tempR, tempG, tempB: signed(31 downto 0);
   variable Red_raw, Green_raw, Blue_raw: signed(31 downto 0);

  begin

    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
      if Bus2IP_Reset = '1' then
   -- set initial values

   convert_state <= IDLE;

   s_Y0 <= (others => '0');
   s_Y1 <= (others => '0');
   s_Y2 <= (others => '0');
   s_Y3 <= (others => '0');

   s_Cb0 <= (others => '0');
   s_Cb1 <= (others => '0');
   s_Cb2 <= (others => '0');
   s_Cb3 <= (others => '0');

   s_Cr0 <= (others => '0');
   s_Cr1 <= (others => '0');
   s_Cr2 <= (others => '0');
   s_Cr3 <= (others => '0');

   s_R0 <= (others => '0');
   s_R1 <= (others => '0');
   s_R2 <= (others => '0');
   s_R3 <= (others => '0');

   s_G0 <= (others => '0');
   s_G1 <= (others => '0');
   s_G2 <= (others => '0');
   s_G3 <= (others => '0');

   s_B0 <= (others => '0');
   s_B1 <= (others => '0');
   s_B2 <= (others => '0');
  s_B3 <= (others => '0');

     else
       case slv_reg_write_sel is
         when "100000" =>
     s_Y0 <= Bus2IP_Data(0 to 7);
     s_Y1 <= Bus2IP_Data(8 to 15);
     s_Y2 <= Bus2IP_Data(16 to 23);
     s_Y3 <= Bus2IP_Data(24 to 31);
         when "010000" =>
           s_Cb0 <= Bus2IP_Data(0 to 7);
     s_Cb1 <= Bus2IP_Data(8 to 15);
     s_Cb2 <= Bus2IP_Data(16 to 23);
     s_Cb3 <= Bus2IP_Data(24 to 31);
         when "001000" =>
           s_Cr0 <= Bus2IP_Data(0 to 7);
     s_Cr1 <= Bus2IP_Data(8 to 15);
     s_Cr2 <= Bus2IP_Data(16 to 23);
     s_Cr3 <= Bus2IP_Data(24 to 31);

     convert_state <= YCC_SETUP;
         when others => null;
       end case;

------- Multiplex the inputs and outputs of the colour conversion -----
  case(convert_state) is
     when (IDLE) =>
        null;
     when (YCC_SETUP) =>
        s_Y := s_Y0;
        s_Cb := s_Cb0;
        s_Cr := s_Cr0;
        convert_state <= YCC0;

     when(YCC0) =>
        s_R0 <= s_R;
        s_G0 <= s_G;
        s_B0 <= s_B;

        s_Y := s_Y1;
        s_Cb := s_Cb1;
        s_Cr := s_Cr1;
     convert_state <= YCC1;

     when(YCC1) =>
        s_R1 <= s_R;
        s_G1 <= s_G;
        s_B1 <= s_B;

        s_Y := s_Y2;
        s_Cb := s_Cb2;
        s_Cr := s_Cr2;
        convert_state <= YCC2;

     when(YCC2) =>
        s_R2 <= s_R;
        s_G2 <= s_G;
        s_B2 <= s_B;

        s_Y := s_Y3;
        s_Cb := s_Cb3;
        s_Cr := s_Cr3;
        convert_state <= YCC3;

     when(YCC3) =>
        s_R3 <= s_R;
        s_G3 <= s_G;
        s_B3 <= s_B;
        convert_state <= IDLE;
     when others => null;
   end case;

     -------------------conversion------------------------
     -- RED Calculations
     tempR := resize(shift_right(((signed(x"000000" & unsigned(s_Cr))- x"00000080")
         * CrR) + ONE_HALF,16),32);
     Red_raw := tempR + signed(x"000000" & unsigned(s_Y));

     -- BLUE Calculations
     tempB := resize(shift_right(((signed(x"000000" & unsigned(s_Cb))- x"00000080")
         * CbB) + ONE_HALF,16),32);
     Blue_raw := tempB + signed(x"000000" & unsigned(s_Y));

     -- GREEN Calculations
     tempG := resize(shift_right(
     -CrG * (signed(x"000000" & unsigned(s_Cr))- x"00000080")
     -CbG * (signed(x"000000" & unsigned(s_Cb))- x"00000080") + ONE_HALF , 16) ,32);
     Green_raw := tempG + signed(x"000000" & unsigned(s_Y));

     end if; -- end of reset event
   end if; -- end of clk event

   -------------- Asynchronous range limiting -----------
        -- Range limit Reds
     if (Red_raw < 0) then s_R := x"00";
     elsif (Red_raw > 255) then s_R := x"FF";
     else
     s_R := std_logic_vector(Red_raw(7 downto 0));
     end if;

     -- Range limit Blues
     if (Blue_raw < 0) then s_B := x"00";
     elsif (Blue_raw > 255) then s_B := x"FF";
     else
     s_B := std_logic_vector(Blue_raw(7 downto 0));
     end if;

     -- Range limit Greens
     if (Green_raw < 0) then s_G := x"00";
     elsif (Green_raw > 255) then s_G := x"FF";
     else
     s_G := std_logic_vector(Green_raw(7 downto 0));
     end if;

 end process SLAVE_REG_WRITE_PROC;

 -- implement slave model software accessible register(s) read mux
 SLAVE_REG_READ_PROC : process( slv_reg_read_sel,
           s_Y0, s_Y2, s_Y3,
           s_Cb0, s_Cb1, s_Cb2, s_Cb3,
           s_Cr0, s_Cr1, s_Cr2, s_Cr3,
           s_R0, s_R1, s_R2, s_R3,
           s_G0, s_G1, s_G2, s_G3,
           s_B0, s_B1, s_B2, s_B3  ) is
 begin

   case slv_reg_read_sel is
     when "100000" => slv_ip2bus_data <= s_Y0 & s_Y1 & s_Y2 & s_Y3;
     when "010000" => slv_ip2bus_data <= s_Cb0 & s_Cb1 & s_Cb2 & s_Cb3;
     when "001000" => slv_ip2bus_data <= s_Cr0 & s_Cr1 & s_Cr2 & s_Cr3;

     when "000100" => slv_ip2bus_data <= s_R0 & s_G0 & s_B0 & s_R1;
     when "000010" => slv_ip2bus_data <= s_G1 & s_B1 & s_R2 & s_G2;
     when "000001" => slv_ip2bus_data <= s_B2 & s_R3 & s_G3 & s_B3;
```

```
      when others => slv_ip2bus_data <= (others => '0');
    end case;

  end process SLAVE_REG_READ_PROC;

  ------------------------------------------
  -- Example code to drive IP to Bus signals
  ------------------------------------------
  IP2Bus_Data  <= slv_ip2bus_data when slv_read_ack = '1' else
                  (others => '0');

  IP2Bus_WrAck <= slv_write_ack;
  IP2Bus_RdAck <= slv_read_ack;
  IP2Bus_Error <= '0';

end IMP;
```

user_logic_c_converter.vhd

Appendix E

Huffman Decoding Example

This appendix demonstrates the Huffman decoding of JPEG images [5]. Using the Huffman tables provided within the JPEG image, the raw bitstream can be decoded back into the DCT coefficients. JPEG images contain three colour channels (one luminance and two chrominance channels). Each of the three channels must be decoded to reconstruct the image.

E.1 Source Image

To simplify this example an image of 16x8 pixel image consisting of only black and white is used as shown in Figure E.1. This image was created with photo editing software, saving it as a grayscale JPEG image with no colour sub-sampling, and no

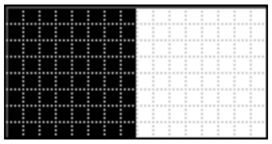
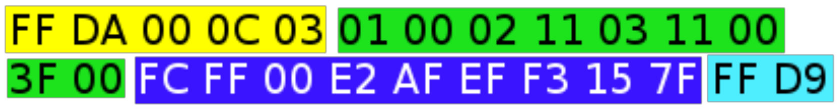


Figure E.1: Sample image



```
FF DA 00 0C 03 01 00 02 11 03 11 00
3F 00 FC FF 00 E2 AF EF F3 15 7F FF D9
```

Figure E.2: HEX dump from sample image

Optimizations. With the default settings the image uses the set of default Huffman tables, and samples the chrominance channels for each 8x8 block. These settings are what you'd find in most digital cameras.

The next step is to decode the raw image data to analyze. There are several software applications that can perform a HEX dump to view the bitstream. Decoding the image data by hand is not practical, but is educational for this purpose. In the dump the **Start of Scan, SOS**, marker (0xFFDA) is the starting point to look for (Figure E.2) highlighted in yellow. Following the SOS marker are a few bytes for additional details about the image highlighted in green, the actual scan data highlighted in blue, and finally terminated with an **End of Scan, EOS**, marker (0xFFD9).

The scan data to decode is

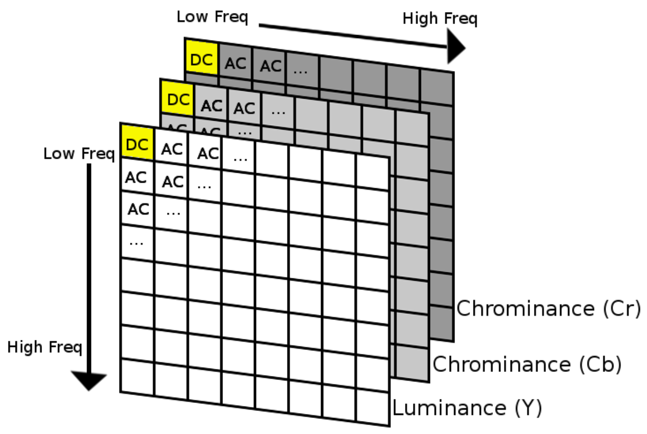


Figure E.3: DCT Frequency domain of an 8x8 JPEG block

FC FF 00 E2 AF EF F3 15 7F

To improve JPEG resiliency, JPEG markers are permitted to reside any part of the raw encoded data. Therefore a JPEG decoder must watch out for any occurrence of the 0xFF byte. In the example scan there exists an 0xFF byte followed by a 0x00 byte. This is known as the stuff byte, alerting the decoder that this is image data and not a JPEG marker. As a result FF00 can be replaced with FF reformatting the scan data to:

FC FF E2 AF EF F3 15 7F

Conventional images have three components (Y, Cb, and Cr). Within each 8x8 block there is one DC component, followed by 63 AC components. Figure E.3 shows this arrangement within an 8x8 block of no chroma sub-sampling.

The DC component represents the average value of all pixels in the block. For this example, the DC value will represent either black or white. Worth mentioning, but demonstrated later is that the DC component is encoded as a relative value with respect to the value of the DC component in the previous block. The first block in a JPEG image is assumed to have a previous value of 0.

Following the DC entry, are 1-63 entries used to describe the low and high frequency components of the DCT frequency domain (AC components). Early AC components represent low frequency, later ones represent high frequency image content. Since the 2D-DCT focuses its energy in the upper left corner of the 8x8 block many of the high frequency components in the image are zero.Due to the fact that the image used in this example has constant colour across each 8x8 block there are no non-zero AC components.

E.2 Huffman Table Extraction

From the bitstream Huffman Code tables can also be extracted from the JPEG image (Tables E.2 - E.5 at the end of this appendix). The tables are separated by the JPEG DHT marker. The following 4 tables were extracted from this sample image. Note using the Optimization feature, when creating the image, will create vastly different tables. The tables in the JPEG only provide length and code values, not the actual bit-string mapping. It is up to the decoder to build the binary tree representation of the DHT tables to derive the bit-strings.

The Huffman tree representation for this can be computed, but there is a simple pattern many encoders/decoders use. The pattern is:

1. Start at the lowest length of bits for the first code, and give it 0 with the

appropriate number of bits.

2. Increment the binary number by one for each additional code within the same bit length

3. When moving on to the next bit length; increment the binary number and shift left by one.

The Huffman DC Value Encoding table E.6 is the last table needed to fully decode the Huffman stream, but it is not included in the JPEG data. It must be generated, but is applicable to any JPEG file.

E.3 Image Decoding

Returning to the scan data, now converted into its binary representation.

F	C	F	F	E	2	A	F	E	F	F	3	1	5	7	F
1111	1100	1111	1111	1110	0010	1010	1111	1110	1111	1111	0011	0001	0101	0111	1111

E.3.1 Block 1 - Luminance

Luminance DC

Referring to the Luminance DC Table E.2, the bits of the bitstream are read one-by-one until a match is found.

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Code = 0A. Code 0A (10 in decimal) implies that the next 10 bits in the stream represent the signed value of the DC component. The next ten bits are:

1111 1100 1111 1111 11110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Using Table E.6 **0 1111 1111 1** has a DC Value of **-512**.

Luminance AC

Moving on to the Luminance AC component, the bitstream is matched to an entry in TableE.3.

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Code = 00 (EOB), meaning the 63 AC entries are all zero, and the Luminance channel is complete.

E.3.2 Block 1 - Chrominance

Chrominance(Cb) DC

Continue bit matching for the Chrominance DC component using Table E.4.

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Code = 00 (EOB).

Chrominance(Cb) AC

Bit match for Chrominance AC using Table E.5.

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Code = 00 (EOB). Continue with Chrominance Cr in the same manner.

Chrominance(Cr) DC

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Code = 00 (EOB).

Chrominance(Cr) AC

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Code = 00 (EOB). Block 1 is finished. Decoding scheme continues with Block 2.

E.3.3 Block 2 - Luminance

Luminance DC

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Code = 0A. This implies the value is stored in the next 10 bits of the stream.

1111 1100 1111 1111 1110 0010 1010 1111 11101111 1111 0011 0001 0101 0111 1111

Using Table E.6 **1111 1111 00** has a DC Value of **+1020**. As previously mentioned, since the DC values are relative to the preceding blocks, the final DC value of this block is +1020 + (-512) = +508.

Luminance AC

1111 1100 1111 1111 1110 0010 1010 1111 11101111 1111 0011 0001 0101
0111 1111

Code = 00 (EOB).

E.3.4 Block 2 - Chrominance

Chrominance(Cb) DC

1111 1100 1111 1111 1110 0010 1010 1111 11101111 1111 0011 0001 0101
0111 1111

Code = 00 (EOB).

Chrominance(Cb) AC

1111 1100 1111 1111 1110 0010 1010 1111 11101111 1111 0011 00010101
0111 1111

Code = 00 (EOB).

Chrominance(Cr) DC

1111 1100 1111 1111 1110 0010 1010 1111 11101111 1111 0011 0001 0101
0111 1111

Code = 00 (EOB).

Chrominance(Cr) AC

1111 1100 1111 1111 1110 0010 1010 1111 11101111 1111 0011 0001 01010111 1111

Code = 00 (EOB).

E.4 Finalizing

The remaining bits are discarded. This is because the scan data must end on a byte boundary. The Huffman conversion is now complete and these results can be passed to the IDCT.

Table E.1: Huffman Decoding Results

	Block 1			**Block 2**		
	Y	**Cb**	**Cr**	**Y**	**Cb**	**Cr**
DC	-512	0	0	+508	0	0
AC	0	0	0	0	0	0

E.5 Huffman Tables

Table E.2: Huffman Luminance (Y) DC table

Length	**Bits**	**Code**
3 bits	000	04
	000	05
	001	03
	010	02
	011	06
	100	01
	101	00 (End of Block)
4 bits	1110	07
5 bits	1111 0	08
6 bits	1111 10	09
7 bits	1111 110	0A
8 bits	1111 1110	0B

Table E.3: Huffman Luminance (Y) AC table

Length	**Bits**	**Code**
2 bits	00	01
	01	02
3 bits	100	03
4 bits	1010	11
	1011	04
	1100	00 (End of Block)
5 bits	1101 0	05
	1101 1	21
	1110 0	12
6 bits	1110 10	31
	1110 11	41
...	...	...
12 bits	...	...
	1111 1111 0011	F0 (ZRL)
	...	...
16 bits	...	...
	1111 1111 1111 1110	FA

Table E.4: Huffman Chrominance (Cb and Cr) DC table

Length	**Bits**	**Code**
2 bits	00	01
	01	00 (End of Block)
3 bits	100	02
	101	03
4 bits	1100	04
	1101	05
	1110	06
5 bits	1111 0 0	07
6 bits	1111 10	08
7 bits	1111 110	09
8 bits	1111 1110	0A
9 bits	1111 1111 0	0B

Table E.5: Huffman Chrominance (Cb and Cr) AC table

Length	**Bits**	**Code**
2 bits	00	01
	01	00 (End of Block)
3 bits	100	02
	101	11
4 bits	1100	03
5 bits	1101 0	04
	1101 1	21
	1110 00	12
6 bits	1110 01	31
	1110 10	41
...	...	...
	...	...
9 bits	1111 1100 0	F0 (ZRL)
	...	...
...	...	...
16 bits	...	...
	1111 1111 1111 1110	FA

Table E.6: Huffman DC Value Encoding

DC Code	**Size**	**Additional Bits**		**DC Value**	
00	0			0	
01	1	0	1	-1	1
02	2	00, 01	10, 11	-3, -2	2,3
03	3	000,001,010,011	100,101,110,111	-7,-6,-5,-4	4,5,6,7
04	4	0000,...,0111	1000,...,1111	-15,...,-8	8,...,15
05	5	0000 0,...	...,1111 1	-31,...,-16	16,...31
06	6	0000 00,...	...,1111 11	-63,...,-32	32,...,63
07	7	0000 000,...	...,1111 111	-127,...,-64	64,...,127
08	8	0000 0000,...	...,1111 1111	-255,...,-128	128,...,255
09	9	0000 0000 0,...	...,1111 1111 1	-511,...,-256	256,...,511
0A	10	0000 0000 00,...	...,1111 1111 11	-1023,...,-512	512,...,1023
0B	11	0000 0000 000,...	...,1111 1111 111	-2047,...,-1024	1024,...,2047

Vita Auctoris

Dan was born and raised in Windsor, Ontario. After completing his Bachelor of Applied Science Degree in Electrical Engineering at the University of Windsor, he pursued his Masters of Applied Science. With an undergraduate background focussed in the field of digital communications, he expanded his experience in the field of Embedded System Design.