8-1-2009

VHDL modeling and synthesis of the JPEG-XR inverse transform

Peter Frandina

VHDL Modeling and Synthesis of the JPEG-XR Inverse Transform

by

Peter Frandina

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Computer Engineering

Supervised by

Professor Dr. Kenneth W. Hsu
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2009

Approved by:

Dr. Kenneth W. Hsu, Professor
Thesis Advisor, Department of Computer Engineering

Dr. Marcin Lukowiak, Assistant Professor
Committee Member, Department of Computer Engineering

Dr. Roy Melton, Lecturer
Committee Member, Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

VHDL Modeling and Synthesis of the
JPEG-XR Inverse Transform

I, Peter Frandina, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

_____

Peter Frandina

_____

Date

Acknowledgments

This section is devoted to expressing my gratitude towards the individuals who contributed in making this work possible. I would first like to thank Dr. Hsu, for remaining patient and for encouraging me to continue when progress seemed unreachable. I would also like to thank Dr. Lukowiak for helping me to develop a strong interest in VHDL. Finally I would like to thank Dr. Melton for spending the time to make this work possible.

Abstract

VHDL Modeling and Synthesis of the JPEG-XR Inverse Transform

Peter Frandina

Supervising Professor: Dr. Kenneth W. Hsu

The Joint Photographic Experts Group (JPEG) recently approved Microsoft's HD Photo format to become the newest JPEG standard under the name JPEG-XR, an indication of the format's extended range capabilities. Although the HD Photo format offers many of the same improvements as JPEG 2000, the group's first standardized improvement on the original JPEG format, HD Photo requires much less computational complexity and can be compressed or decompressed considerably faster.

This thesis explores a VHDL model of the inverse lapped biorthogonal transform used in the soon to be released JPEG-XR decompression algorithm, targeting an Altera FPGA. Testing is performed using VHDL testbenches for simulation, and C-language software models. The final implementation is analyzed according to size, speed, and power consumption. It is presented that this implementation of the inverse transform uses 12,796 Combinational ALUTs, 17,908 total registers, and 141,258 total block memory bits on a Stratix II FPGA. It is capable of completing image transforms at a linear rate proportional to the number of pixels in the image with a latency of 99 clock cycles. This represents nearly 2% of the memory bits used on a similarly constrained JPEG-XR forward transform implementation and includes 12 less registers in the pipeline.

Contents

List of Tables

List of Figures

Glossary

DCT Discrete Cosine Transform.

DWT Discrete Wavelet Transform.

FPGA Field Programmable Gate Array.

HD High Definition.

ILBT Inverse Lapped Biorthogonal Transform.

IPCT Inverse Photo Core Transform.

IPOF Inverse Photo Overlap Filter.

ISO International Organization for Standardization.

ITU International Telecommunication Union.

JPEG Joint Photographic Experts Group.

LBT Lapped Biorthogonal Transform.

PCT Photo Core Transform.

POF Photo Overlap Filter.

RAM Random Access Memory.

ROI Region of Interest Decoding.

VHDL Very High Speed Integrated Circuit Hardware Description Language.

Chapter 1

Introduction

Modern digital cameras have the ability to capture images of amazing quality, however so much raw image data is generated that the digital file of the image would need to be several gigabytes in size to preserve all of the information. A collection of these images in raw form would then be too large to feasibly store on personal computer memory and nearly impossible to email or print across a network. For this reason images are compressed to reduce the size of the image file and to effectively allow for quicker transmission across data networks. When the compressed image is then desired for viewing, editing, or printing, the corresponding decompression algorithm is used to restore the original image.

In order to have compatible software for the many different digital cameras available, a standard is needed so that images compressed by these different cameras can rightfully be restored during decompression. A committee was formed in 1986 to serve as a working group to create the standard for still image compression. This committee, the Joint Photographic Experts Group (JPEG), was a combined effort of the International Organization for Standardization (ISO) and the International Telecommunication Union (ITU). The group has since defined many standards and most recently approved Microsoft's HD Photo format to become the newest JPEG standard under the name JPEG-XR, representing its support for extended range capabilities. This new soon-to-be standard has passed a committee ballot and is currently listed in the publication stage on the ISO website. JPEG-XR provides an algorithm that reduces compression size and the complexity of math operations used during compression and decompression.

The main contribution of this work consists of an implementation of the inverse transform used in the JPEG-XR decompression algorithm, and a comparison of the JPEG-XR paths for compression and decompression. A comparison of this JPEG-XR inverse transform implementation to a similarly constrained JPEG-XR forward transform implementation is also provided.

The subsequent chapters of this document are organized in the following manner. Chapter 2 provides the reader with a basic understanding of common compression and decompression techniques for digital images. Chapter 3 begins with an explanation of the features supported by JPEG-XR and the relationship between its compression and decompression paths. Chapter 4 delves into a detailed description of the JPEG-XR decompression algorithm. Chapter 5 provides the reader with a description of the VHDL implementation used in this research. Chapter 6 explains the verification methods and test procedures used to verify correct functionality of the implementation. Chapter 7 discusses the results gathered from this work and provides an analysis with respect to the forward transform implementation in [2]. Chapter 8 provides suggestions for future work, including areas that can be optimized using alternative techniques, additional features that improve or build on this implementation, and further development in general for the compression and decompression of digital images. This chapter concludes the thesis with a discussion of its contributions to the field of digital image compression and decompression

Chapter 2

Literature Review

2.1 Why Compression is Necessary

When a photograph is captured digitally, it would seem ideal in most cases to retain as much information as possible from the digital camera. Then when the image is desired for viewing or printing, all of the captured information is readily available. Preserving all of the captured information in its raw form however would require several gigabytes of storage media and either a very fast communication media or require a long delay time to send and receive images. For this reason, the raw data of the digital image is compressed into a smaller file size. Compression takes advantage of the redundancy in an image so that the number of bits needed to represent the given image can be reduced. For example neighboring pixels in an image are not independent, but rather very much correlated. This compression allows less data to be required for storage and less bandwidth to be required for transmission.

Compressed image files are coded and therefore anyone wishing to understand the file must know how to return the information back to the original image, a process known as decompression. Decompression is a complimentary process to compression which decodes the compressed file and returns the data to the image as it was originally captured. There are many tradeoffs for the various compression and decompression algorithms including algorithmic complexity, transform completion time, and transform quality. In most cases the primary consideration is whether or not any damaging artifacts or degraded quality as

a result of the reduction in storage size can be tolerated. Transforms are therefore grouped according to this property into two categories; lossy and lossless.

2.2 Lossy vs Lossless

Formats that can compress and decompress a raw image without losing information or introducing artifacts are called lossless. These are often referred to as ideal transforms, since they do not alter the data being transformed. Formats that compress and decompress a raw image with even minimal loss of information or artifacts introduced are called lossy. The determination of which format should be used is essentially dependent on the application field.

An argument is often made in favor of lossy formats that a certain amount of image data loss can actually be afforded without degrading the quality perceived by the human eye. For example, the human eye is much more sensitive to variations in luminance than to variations in color. This less important information can be rounded off to achieve better compression. In addition to the smaller compression size, lossy transformations are typically much less complex than lossless transforms and can therefore be compressed and decompressed considerably faster. The combination of reduced size and faster speed allows lossy formats to be common for personal photo use and even preferred for the internet.

Although lossless formats cannot compress images as small as lossy formats, they do allow reconstruction of the images without any information loss or introduction of artifacts. Therefore applications which take advantage of the high fidelity image information require a lossless transformation. National security applications that analyze images using computers can use the additional color information which is not necessarily perceived by the human eye. Medical imaging is another field which prefers lossless transforms. Any transform artifact or color round off error in a medical image has the potential to result in incorrect diagnosis or the passing of a life threatening problem that is not seen by the doctor analyzing the images.

2.3 Existing Formats

There are many formats for the purpose of digitally compressing raw data, especially when considered in the context of digital photography. However the most popular formats are those that have been established as a standard by the Joint Photographic Experts Group (JPEG). These standardized formats are the ones which developers incorporate compatibility for in their hardware and software products.

2.3.1 JPEG

The group established its first standard, which is now commonly referred to as the JPEG standard, for still images in 1990 (though formally named ISO/IEC IS 10918-1 and ITU-T Recommendation T.81) and was soon adopted by commercial applications in 1991 [6]. Partly due to the booming growth of the personal computer and digital camera markets and the popularity of the internet, JPEG became the definitive standard through the 1990s. There are now several modes defined for JPEG, including baseline, lossless, progressive and hierarchical, though the baseline remains most popular and supports lossy coding only [7].

The baseline mode uses a technique which divides the image into 8x8 blocks and applies a discrete cosine transform (DCT) to each. The transformed blocks are quantized with a uniform scalar quantizer, zig-zag scanned and subsequently entropy coded with Huffman coding. Each 8x8 block yields 64 coefficients whose quantization step sizes are defined in a quantization table, and remain constant for all blocks. A predictive method is then used to separately code the DC coefficients of all blocks [7]. The decompression process for all intents and purposes executes these steps in reverse order to decode the coefficients, apply an inverse of the transform and return to an image similar to the one that was compressed.

Although a lossless mode exists for JPEG, it is based on a completely different algorithm than the baseline mode, and thus is not directly compatible with the baseline mode.

This presents a major obstacle for JPEG, as the lossless mode does not support lossy coding.

2.3.2 JPEG 2000

As the years passed and digital cameras significantly improved, the megapixel quality and sizes of digital images were also growing. In 2000, the JPEG group released a new standard meant as an update on the original JPEG standard. It became known as JPEG 2000, though formally identified as ISO/IEC 15444-1 and ITU-T Recommendation T.800.

JPEG 2000 introduced many improvements upon the initial JPEG standard in terms of both the compression ratio and the feature set. This includes the ability to perform either lossy or lossless compression, parseable and progressive bitstreams, error resilience, region of interest (ROI) decoding, and several other features using a single algorithm [7].

The updated format is based on the discrete wavelet transform (DWT), scalar quantization, context modeling, arithmetic coding and post-compression rate allocation [7]. The DWT can be implemented using either the reversible Le Gall (5,3) taps filter or the non-reversible Daubechies (9,7) taps biorthogonal filter [8]. The latter provides higher compression though does not support lossless coding. The quantizer remains independent for each sub-band while utilizing a dead-zone scalar technique. Each sub-band is partitioned into rectangular code-blocks, which are typically 64x64, and then entropy coded using context modeling and bit-plane arithmetic coding. Then the coded data is organized into layers of varying quality levels by using the post compression rate allocation, and finally output to the code-stream in packets [7].

A strong focus on features however resulted in additional complexity which unfortunately was enough for commercial applications to never fully adopt the JPEG 2000 standard. Nearly twenty years after its initial release, the JPEG standard remains as the clear dominant format for commercial digital photography.

Chapter 3

Theory

3.1 JPEG-XR

The JPEG-XR standard provides many of the same improvements presented in JPEG 2000, but without the algorithm implementation complexities as a result of integer only operations. JPEG-XR is designed for high dynamic range and its scalability will allow it to sustain the increasing size of images.

The process for JPEG-XR compression at a high level is comparable to that of JPEG. Although there are many differences at the lower level, Figure 3.1 shows that the steps in each process are very similar. The main differences are simply the type of transform and the methods used for quantizing, scanning, and encoding.

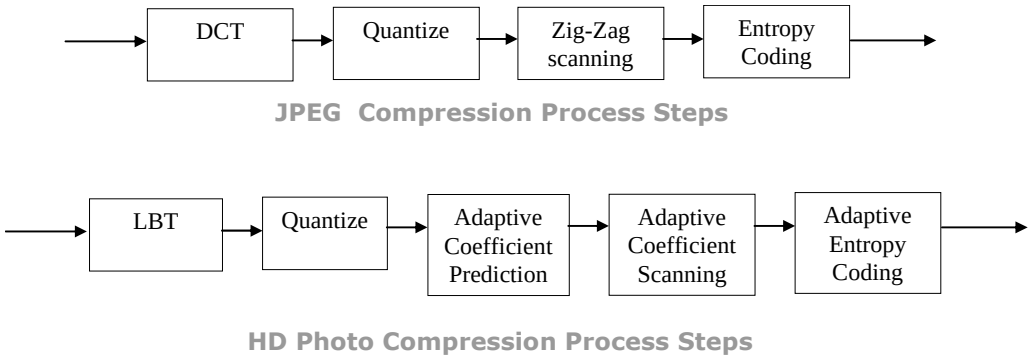

Figure 3.1: JPEG and JPEG-XR Compression Process [3]

Both JPEG and JPEG-XR begin with a transform stage which transforms the pixel information from the spatial domain to the frequency domain. Then a quantization stage divides each coefficient by some integer value, rounding to the nearest integer, so that lossy transforms only need to maintain no more than the information necessary. For lossless transforms a quantization factor of 1 is used and thus will not affect the transformed coefficients. This quantization table of conversion factors must be carried along with the compressed file so that it may later be used in decompression. Next the coefficients are scanned (using a predictive method for JPEG-XR) for order of increasing frequency and finally entropy coded.

3.1.1 Supported Formats

Most digital cameras use a bit depth of 8 bits per pixel to capture images, though today it is common for more intense applications to use a bit depth of 16 bits per pixel. It is possible however that a small set of applications require even more. This corresponds to having 256 color values to represent a pixel with 8 bit depth, and 65536 color values to represent a pixel with 16 bit depth. It is generally the case where 0 represents zero intensity for a color channel (i.e. black is when all channels are 0), and the maximum represents full intensity or full saturation of a color channel (i.e. white is when all channels are maximum). The specific unsigned integer values chosen to represent real world colors will vary depending on the manufacturer of the camera, because each camera and model has its own limitations for the number of bits used. Therefore many color profiles have been determined with a specified gamma curve for each to describe the correlation between the bits used in the profile and real world colors. JPEG-XR includes full compatibility with existing devices by supporting a wide range of profiles in three numerical formats (unsigned integer, fixed point, and floating point). It should be noted that lossless compression and decompression is only supported up to a bit depth of 26 bits per pixel. Higher bit depths are clipped within the algorithm and thus result in a lossy mode. Table 3.1 provides a full listing of supported pixel formats including the common name (Format), number of color channels

represented including the alpha channel (NC), whether or not the alpha channel is present (Alpha), number of bits per sample per color channel in the decoded image (BPC), number of bits per pixel in the decoded image (BPP), numeric interpretation of values (Num), and the name corresponding to the structure of pixels (Color).

3.1.2 Data Hierarchy

An image is composed of one or two image planes: a required primary image plane and an optional alpha image plane. Each image plane is an ordered set of components and each component is an array of samples. JPEG-XR defines additional partitions for an image so that smaller sections of the total image may be processed separately. This has a beneficial effect of decreased memory necessary for processing.

Spatial Hierarchy

The smallest element of an image is a pixel. Each 4x4 set of pixels is grouped as a block. Then each 4x4 set of blocks is grouped as a macroblock. A set of Macroblocks can then be grouped as a tile, though the number of macroblocks included along the width and height may vary between tiles. At the highest level of the hierarchy, the tiles come together to form the entire image. A diagram of this partitioning is shown in Figure 3.2.

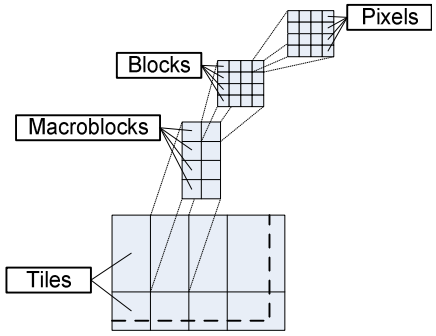

Figure 3.2: Overview of Image Partitioning [1]

Format	NC	Alpha	BPC	BPP	Num	Color
24bppRGB	3		8	24	UINT	RGB
24bppBGR	3		8	24	UINT	RGB
32bppBGR	3		8	24	UINT	RGB
48bppRGB	3		16	48	UINT	RGB
48bppRGBFixedPoint	3		16	48	SINT	RGB
48bppRGBHalf	3		16	48	Float	RGB
96bppRGBFixedPoint	3		32	96	SINT	RGB
128bppRGBFloat	3		32	128	Float	RGB
32bppBGRA	4	✓	8	32	UINT	RGB
64bppRGBA	4	✓	16	64	UINT	RGB
64bppRGBAFixedPoint	4	✓	16	64	SINT	RGB
64bppRGBAHalf	4	✓	16	64	Float	RGB
128bppRGBAFixedPoint	4	✓	32	128	SINT	RGB
128bppRGBAFloat	4	✓	32	128	Float	RGB
32bppPBGRA	4	✓	8	32	UINT	RGB
64bppPRGBA	4	✓	16	64	UINT	RGB
128bppPRGBAFloat	4	✓	32	128	Float	RGB
32bppCMYK	4		8	32	UINT	CMYK
40bppCMYKAlpha	5	✓	8	40	UINT	CMYK
64bppCMYK	4		16	64	UINT	CMYK
80bppCMYKAlpha	5	✓	16	80	UINT	CMYK
24bpp3Channels	3		8	24	UINT	n-Chn
32bpp4Channels	4		8	32	UINT	n-Chn
40bpp5Channels	5		8	40	UINT	n-Chn
48bpp6Channels	6		8	48	UINT	n-Chn
56bpp7Channels	7		8	56	UINT	n-Chn
64bpp8Channels	8		8	64	UINT	n-Chn
32bpp3ChannelsAlpha	4	✓	8	32	UINT	n-Chn
40bpp4ChannelsAlpha	5	✓	8	40	UINT	n-Chn
48bpp5ChannelsAlpha	6	✓	8	48	UINT	n-Chn
56bpp6ChannelsAlpha	7	✓	8	56	UINT	n-Chn
64bpp7ChannelsAlpha	8	✓	8	64	UINT	n-Chn
72bpp8ChannelsAlpha	9	✓	8	72	UINT	n-Chn
48bpp3Channels	3		16	48	UINT	n-Chn
64bpp4Channels	4		16	64	UINT	n-Chn
80bpp5Channels	5		16	80	UINT	n-Chn
96bpp6Channels	6		16	96	UINT	n-Chn
112bpp7Channels	7		16	112	UINT	n-Chn
128bpp8Channels	8		16	128	UINT	n-Chn
64bpp3ChannelsAlpha	4	✓	16	64	UINT	n-Chn
80bpp4ChannelsAlpha	5	✓	16	80	UINT	n-Chn
96bpp5ChannelsAlpha	6	✓	16	96	UINT	n-Chn
112bpp6ChannelsAlpha	7	✓	16	112	UINT	n-Chn
128bpp7ChannelsAlpha	8	✓	16	128	UINT	n-Chn
144bpp8ChannelsAlpha	9	✓	16	144	UINT	n-Chn
8bppGray	1		8	8	UINT	Gray
16bppGray	1		16	16	UINT	Gray
16bppGrayFixedPoint	1		16	16	SINT	Gray
16bppGrayHalf	1		16	16	Float	Gray
32bppGrayFixedPoint	1		32	32	SINT	Gray
32bppGrayFloat	1		32	32	Float	Gray
BlackWhite	1		1	1	UINT	Gray
16bppBGR555	3		5	16	UINT	RGB
16bppBGR565	3		5,6,5	16	UINT	RGB
32bppBGR101010	3		10	32	UINT	RGB
32bppRGBE	3		16	32	Float	RGB

Table 3.1: Supported Pixel Formats [1]

Frequency Hierarchy

Image data is also represented in the frequency domain, as shown in Figure 3.3. The transform coefficients associated with each component and macroblock are split into three frequency bands, called DC coefficients, lowpass coefficients, and highpass coefficients. According to this hierarchy, each macroblock of a component contains 256 transform coefficients. Of these coefficients, 1 is DC, 15 are lowpass, and 240 are highpass. This division of hierarchy supports an ability to directly decompress at three different resolutions.

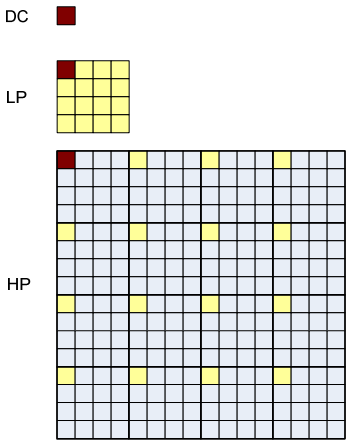

Figure 3.3: Frequency Hierarchy of a Macroblock [4]

3.1.3 Bitstream Structure

The bitstream of a JPEG-XR compressed image is structured as in Figure 3.4. The image may be compressed into either the spatial format or frequency format, and which one is used is entirely up to the encoder. In either case, the bitstream begins with a header to identify important descriptors about the file, and then an index table to identify where the remaining data pieces exist in the file. Then a sequence of tiles follows for either case.

The spatial structure follows with the coefficients of each macroblock stored together, and macroblocks stored sequentially in raster scan order (left to right, then top to bottom). This allows for random access into any macroblock.

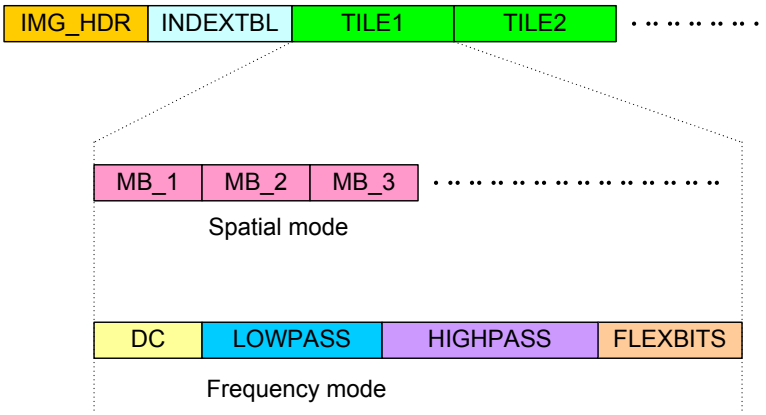

Figure 3.4: JPEG-XR Bitstream Structure [1]

The frequency structure follows with the coefficients grouped according to frequency band. Each band of coefficients is stored in raster scan order. This is the preferred storage for fast decompression, as it allows random access into any frequency band.

3.1.4 Supported Features

Sequential Decoding

The spatial bitstream structure, with macroblocks serialized in raster scan order, plays into a direct memory benefit. Macroblocks can be decoded in raster scan order and thus avoid any significant bitstream buffer.

Region of Interest Decoding

Just as in JPEG 2000, tiling enables region of interest decoding. Each tile is entropy coded independently in JPEG-XR, and therefore only those tiles relevant to the region of interest need to be decoded. This can drastically reduce the level of effort and amount of time to decode thumbnails (i.e. for internet viewing) of larger stored files.

Fast 16:1 and 4:1 Thumbnail Decoding (Spatial Scalability)

The three levels of frequency bands in combination with the two-stage transform, allow direct decoding into three different resolutions. A 16:1 resolution can be achieved by decoding only the DC coefficients. A 4:1 resolution can be achieved by decoding only the DC and lowpass coefficients. And of course a full resolution is achieved by decoding all coefficients. This feature is also useful for decoding thumbnails.

3.2 Previous Work

Forward Lapped Biorthogonal Transform Implementation

The JPEG-XR compression process was recently modeled by Seth Groder using Verilog to target an Altera FPGA [2]. One of the blocks included in this design is the forward lapped biorthogonal transform. Because this transform is implemented as a series of lifting operations, it is a completely reversible procedure. The inverse transform essentially executes the lifting operations in reverse order to complete the lossless compression and decompression process. Although the full implementation of [2] uses three forward LBTs, an analysis of a single instance of the forward LBT is provided in terms of layout size, timing, and power. These metrics are used for a comparison to the results achieved by the inverse transformation implemented in this work.

Chapter 4

Design

4.1 Design Constraints

It would be ideal to design an ILBT module such that it will work with all combinations of the options available for the decompression process. Unfortunately this generic characteristic is not a good requirement to have for a hardware implantation. An FPGA has a limited number of logic blocks, so designs must be constrained in order to fit on a device. Additional logic that is rarely used might require the use of a larger FPGA or increase the power dissipation. When competing against high speed processors, it is helpful for the FPGA design to be trimmed of options in order to reduce latency and the amount of routing required. Reducing the number of options also reduces the number of test cases and makes the design considerably less difficult to validate. Specifically this design constrains the ILBT in the following ways.

First of all, the image input to the ILBT must contain a length and width of pixels which is evenly divisible by sixteen. In the compression scheme, if the width or height of an image is not evenly divisible by sixteen, additional filler pixels are added to internally force the image to have a length and width that is evenly divisible by sixteen.

Secondly, the maximum size of an input image is constrained to be 8400x6600 pixels. This allows approximately fifty six megapixels for an 11x14 inch image at 600 DPI. A limit on the maximum image size is required so that an exact number of bits can be used to represent in hardware the index of a pixel, block or macroblock, and this specific size

was chosen to match the requirement of the forward transform in [2] to allow for a level comparison.

The implementation assumes an 8 bit unsigned pixel format so that 16 bit internal signals can be used in calculations. The JPEG-XR specification clearly indicates that while input pixel formats may be larger than 8 bit, an 8 bit format will never overflow in calculations of an implementation that uses 16 bits. Anything larger than 8 bit has the potential to overflow 16 bits internally. This is also a similar requirement for [2].

JPEG-XR supports lossy modes which down sample the chroma color channels, and thus allow a reduced memory footprint and faster transformation time. The YUV422 color space is half sampled in the horizontal direction and the YUV420 color space is half sampled in both the horizontal and vertical directions. This implementation will only support the fully sampled YUV444 color space to reduce the amount of logic required and to allow a more equal comparison to the forward transform in [2].

4.2 Inverse Lapped Biorthogonal Transform

JPEG-XR uses an inverse lapped biorthogonal transform for sample reconstruction. This is actually a two step process that involves a first level inverse photo core transform (IPCT) with an optional inverse photo overlap filter (IPOF) applied to the DC and lowpass coefficients, and then a second level IPCT with an optional IPOF applied to the resulting values from the first level combined with highpass coefficients. This is shown in Figure 4.1.

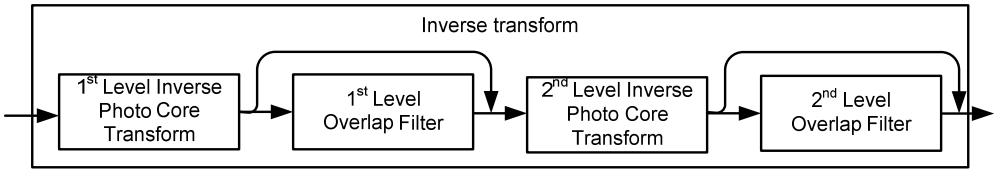

Figure 4.1: JPEG-XR Inverse Transform Process [4]

The IPCT is actually very similar to the Inverse Discrete Cosine Transform (IDCT) used by the JPEG inverse transform. Both are block transforms but use different block

sizes; JPEG uses 8x8 blocks and JPEG-XR uses 4x4 blocks. The IPCT can exploit spatial correlation within the block, but a problem common to block transforms is the potential for blocking artifacts. This is due to the fact that block transforms cannot exploit redundancy across block boundaries, and thus the boundaries can sometimes be seen in a lossy transformed image. As a measure of prevention for these artifacts, the IPOF is optionally applied. The IPOF is designed to compliment the IPCT by exploiting correlation across block boundaries.

There are three different combinations of IPCT and IPOF that can be used. For the fastest transforms and least used resources, no overlap filtering is performed. For a compromised effort, only the second level overlap filter is used. For the best quality images, speed and resources must be sacrificed to accommodate both the first and second level overlap filters. This work implements a design with both levels of overlap filtering.

Figure 4.2 provides an example for the ILBT of a 48x32 pixel image. It is shown that the IPCT operates on macroblock boundaries for the first level and block boundaries for the second level, and that the IPOF operates on intersections of macroblocks for the first level and intersections of blocks for the second level.

Although there are many ways to implement the ILBT design, they can be grouped into three categories as shown in Figure 4.3. The first approach in Figure 4.3A requires the least memory and least logic gates, but is not very efficient since each process block must finish completely before the next process block may begin. The approach in Figure 4.3B requires only slightly more memory to become more efficient because an IPCT and IPOF in the same level can operate in parallel. However, the first level process blocks must still finish completely before the second level process blocks may begin. The approach in Figure 4.3C requires the most memory and most logic gates, and as a result can complete the ILBT in the least amount of time. The implementation of this work is similar to Figure 4.3C, but eliminates the partial buffer between the first and second levels. This style is most suited to a hardware implementation which can exploit parallel operations.

The next sections continue with a description of the elementary operators that make

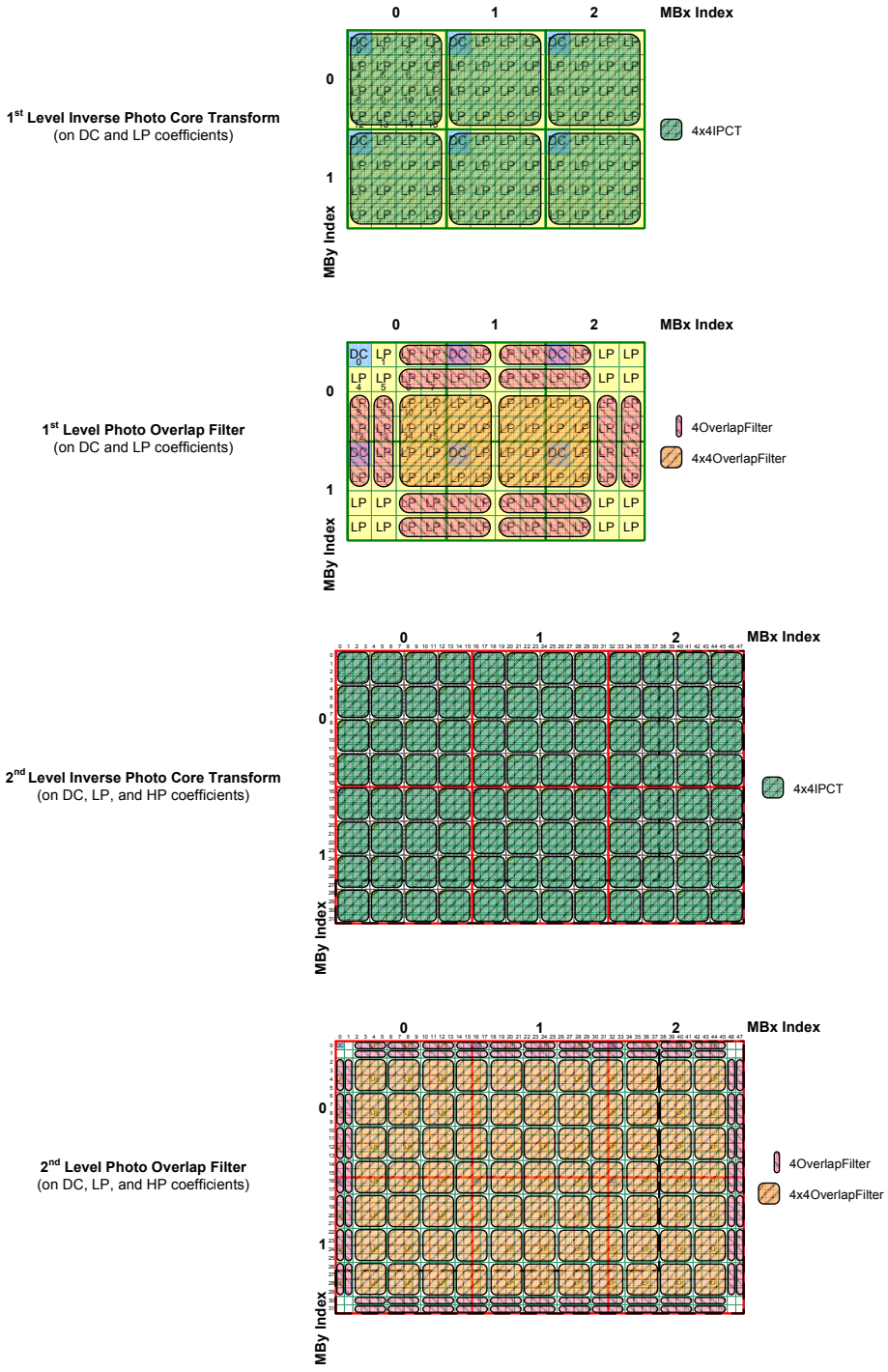

Figure 4.2: Example Steps for ILBT of a 48x32 Pixel Image

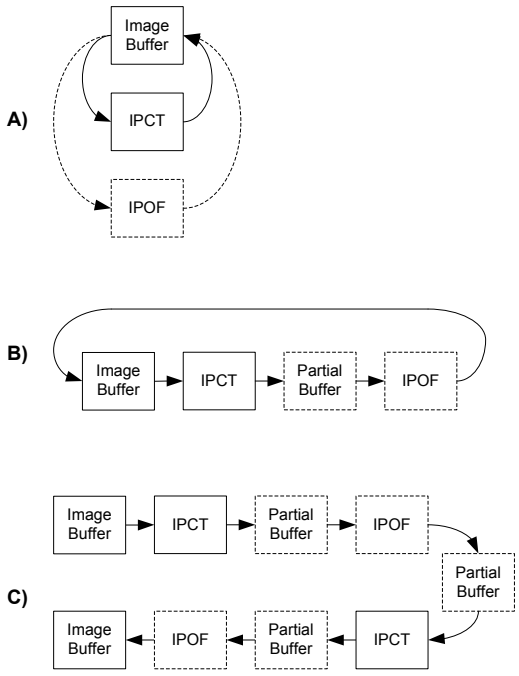

Figure 4.3: Categories of Implementation for the ILBT

up the IPCT and IPOF. Within each operator, mention is made of the number of nontrivial operations. Trivial operations are those that can easily be completed within one clock cycle, such as a single instance of addition, subtraction or bit shift. Non-trivial operations are those that require more than one clock cycle to complete, such as multiplication.

4.3 Inverse Photo Core Transform

The IPCT uses a combination of the following four operators:

- 2x2 Hadamard Transform
- Inverse 1D Rotation
- Inverse 2D Rotation
- Inverse Permutation

These functions utilize a design technique known as lifting in combination with the Kronecker product to separate the 2D transform by first performing 1D transforms on the rows of data, and then performing 1D transforms on the columns of data. Each 1D operator is factorized into a combination of elementary 2-point rotation operators, and then 2D operators are implemented by taking the Kronecker products of the elementary 2-point operators [5]. By using only lifting operations, this inverse transform can completely return to the original data that was transformed and thus is a lossless process.

4.3.1 2x2 Inverse Hadamard Transform

The 2x2 Inverse Hadamard Transform is a 2-point operator as shown in Equation 4.1 [5]. By taking the Kronecker product of T_H with itself, the 2x2 IPCT is formed as seen in Equation 4.2 [5].

$$T_H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{4.1}$$

$$T_H = Kron\,(T_H, T_H) = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \tag{4.2}$$

These formulae can then be reduced to a simple set of lifting instructions involving only trivial operations, which are shown as pseudo-code in Figure 4.4.

4.3.2 2x2 Inverse Odd Transform

The Kronecker product of the 2-point rotation operator in Equation 4.3 and the 2-point Hadamard operator forms the 2x2 Inverse Odd Transform operator [5].

```
2x2T_h(a,b,c,d, R) {
    a += d;
    b –= c;
    t1 = ((a – b + R) >> 1);
    t2 = c;
    c = t1 – d;
    d = t1 – t2;
    a –= d;
    b += c;
}
```

Figure 4.4: Pseudo-code of 2x2 Inverse Hadamard Transform [1]

$$T_R = \begin{pmatrix} -\cos(\frac{\pi}{8}) & \sin(\frac{\pi}{8}) \\ \sin(\frac{\pi}{8}) & \cos(\frac{\pi}{8}) \end{pmatrix} \tag{4.3}$$

When reduced to a set of lifting equations, this operator contains 4 non-trivial operations. The pseudo-code for this operator is shown in Figure 4.5.

```
InvT_odd (a,b,c,d) {
    b += d;
    a –= c;
    d –= (b >> 1);
    c += ((a + 1) >> 1);

    a –= ((3*b + 4) >> 3);
    b += ((3*a + 4) >> 3);
    c –= ((3*d + 4) >> 3);
    d += ((3*c + 4) >> 3;

    c –= ((b + 1) >> 1);
    d = ((a + 1) >> 1) – d;
    b += c;
    a –= d;
}
```

Figure 4.5: Pseudo-code of 2x2 Inverse Odd Transform [1]

4.3.3 2x2 Inverse Odd Odd Transform

The Kronecker product of the 2-point rotation in Equation 4.2 with itself forms the 2x2 Inverse Odd Odd Transform operator. This reduced set of lifting equations contains only 3

non-trivial operations. Pseudo-code for this operator is shown in Figure 4.6.

```
InvT_odd_odd(a,b,c,d) {
d += a;
c –= b;
t1 = d >> 1;
t2 = c >> 1;
a –= t1;
b += t2;

a –= ((b * 3 + 3) >> 3);
b += ((a * 3 + 3) >> 2);
a –= ((b * 3 + 4) >> 3);

b –= t2;
a += t1;
c += b;
d –= a;

b = –b;
c = –c;
}
```

Figure 4.6: Pseudo-code of 2x2 Inverse Odd Odd Transform [1]

4.3.4 4x4 Inverse Permutation

A 4x4 IPCT initially begins by reordering coefficients as shown in Figure 4.7. This shuffling does not require any pipelined stages and does not include any instances of addition / subtraction.

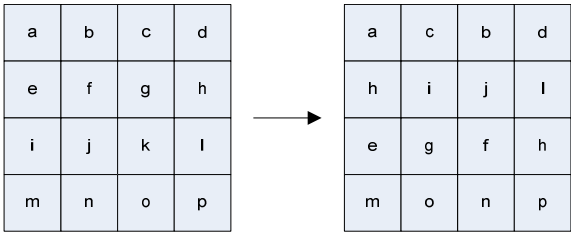

Figure 4.7: Coefficient Re-ordering to begin 4x4 Inverse Core Transform [5]

4.3.5 4x4 Inverse Photo Core Transform

The 4x4 Inverse Photo Core Transform uses a combination of the Hadamard, inverse odd, and inverse odd odd transforms. It can be broken up into three sections with the first being a simple reordering of the inverse permutation. The second and third sections each consist of four 2x2 transforms which may be executed in any sequence as long as a section is completely finished before the next section begins. Pseudo-code for the 4x4 IPCT operator is provided in Figure 4.8. In total, a 4x4 IPCT contains 11 non-trivial operations.

```
4x4ICT (Coeff[0], …, Coeff[15]) {
  /* Permute the coefficients */
  InvPermute(Coeff[0], Coeff[1], …, Coeff[15]);
  /* First stage */
  2x2T_h(Coeff[0], Coeff[1], Coeff[4], Coeff[5], 1);
  InvT_odd(Coeff[2], Coeff[3], Coeff[6], Coeff[7]);
  InvT_odd(Coeff[8], Coeff[12], Coeff[9], Coeff[13]);
  InvT_odd_odd(Coeff[10], Coeff[11], Coeff[14], Coeff[15]);
  /* Second stage */
  2x2T_h(Coeff[0], Coeff[3], Coeff[12], Coeff[15], 0);
  2x2T_h(Coeff[5], Coeff[6], Coeff[9], Coeff[10] , 0);
  2x2T_h(Coeff[1], Coeff[2], Coeff[13], Coeff[14] , 0);
  2x2T_h(Coeff[4], Coeff[7], Coeff[8], Coeff[11] , 0);
}
```

Figure 4.8: Pseudo-code of 4x4 Inverse Photo Core Transform [1]

4.4 Inverse Photo Overlap Filter

The IPOF operates on the boundaries of macroblocks and blocks, so a 4-point IPOF is required in addition to a 4x4 IPOF for the boundaries occurring along the outer edge of an image. These IPOFs utilize the following four operators along with some from the IPCT:

- 2-Point Inverse Rotation
- 2-Point Inverse Scale
- 2x2 Inverse Post Hadamard Transform
- 2x2 Inverse Post Odd Odd Transform

4.4.1 2-Point Inverse Rotation

The 2-Point Inverse Rotation operator does not contain any non-trivial operations, and is represented as pseudo-code in Figure 4.9.

```
invRotate (a,b) {
    a –= ((b + 1) >> 1);
    b += ((a + 1) >> 1);
}
```

Figure 4.9: Pseudo-code of 2-Point Inverse Rotation [1]

4.4.2 2-Point Inverse Scale

The 2-Point Inverse Scale operator contains two non-trivial operations, and is represented as pseudo-code in Figure 4.10.

```
invScale (a,b) {
    a += b;
    b = (a >> 1) – b;

    a += (b * 3 + 0) >> 3;
    b += ((a * 3 + 0) >> 4;
}
```

Figure 4.10: Pseudo-code of 2-Point Inverse Scale [1]

4.4.3 2x2 Inverse Post Hadamard Transform

The 2x2 Inverse Post Hadamard Transform contains only one non-trivial operation. The pseudo-code for this operator is shown in Figure 4.11

4.4.4 2x2 Inverse Odd Odd Post Transform

The 2x2 Inverse Odd Odd Post Transform contains three non-trivial operations. The pseudo-code for this operator is shown in Figure 4.12.

```
2x2T_h_POST (a,b,c,d)  {
    b -= c;
    a += (d * 3 + 4) >> 3;
    d -= (b >> 1);
    c = ((a – b) >> 1) – c;
    t1 = c;
    c = d;
    d = t1;
    a -= d;
    b += c;
}
```

Figure 4.11: Pseudo-code of 2x2 Inverse Post Hadamard Transform [1]

```
InvT_odd_odd_POST(a,b,c,d) {
    d += a;
    c -= b;
    t1 = d >> 1;
    t2 = c >> 1;
    a -= t1;
    b += t2;

    a -= (b*3 + 6) >> 3;
    b += (a * 3 + 2) >> 2;
    a -= (b * 3 + 4) >> 3;

    b -= t2;
    a += t1;
    c += b;
    d -= a;
}
```

Figure 4.12: Pseudo-code of 2x2 Inverse Odd Odd Post Transform [1]

4.4.5 4-Point Inverse Photo Overlap Filter

The 4-Point Inverse Overlap Filter is required for filtering along the two-coefficient wide border of an image, where a macroblock or block does not intersect with anything beyond the edges of the image. This filter contains a total of six non-trivial operations and utilizes the 2-point inverse rotate operator. Pseudo-code for the 4-point filter is provided in Figure 4.13.

```
4OverlapFilter (a,b,c,d) {
    a += d;
    b += c;
    d –= ((a + 1) >> 1);
    c –= ((b + 1) >> 1);

    invRotate(c, d);

    d += ((a + 1) >> 1);
    c += ((b + 1) >> 1);
    a –= d – ((d * 3 + 16) >> 5);
    b –= c – ((c * 3 + 16) >> 5);
    d += ((a * 3 + 8) >> 4);
    c += ((b * 3 + 8) >> 4);
    a += ((d * 3 + 16) >> 5);
    b += ((c * 3 + 16) >> 5);
}
```

Figure 4.13: Pseudo-code of 4-Point Inverse Photo Overlap Filter [1]

4.4.6 4x4 Inverse Photo Overlap Filter

The 4x4 Inverse Photo Overlap Filter uses a combination of the Hadamard, inverse rotation, inverse scale, inverse post odd odd, and post Hadamard transforms. In a similar fashion to the 4x4 IPCT, the 4x4 IPOF can be broken up into four sections and the operators within each section may be executed in parallel. The first and fourth sections contain only 2x2 transforms, while the second and third sections contain a combination of 2x2 and 2-point transforms. The pseudo-code for this operator is shown in Figure 4.14. In total, a 4x4 IPOF contains 15 non-trivial operations.

```
4x4OverlapFilter (a,b,...,p) {
    2x2T_h (a, d, m, p, 0);
    2x2T_h (b, c, n, o, 0);
    2x2T_h (e, h, i, l, 0);
    2x2T_h (f, g, j, k, 0);

    invRotate (n, m);
    invRotate (j, i);
    invRotate (h, d);
    invRotate (g, c);
    InvT_odd_odd_POST (k, l, o, p);

    invScale (a, p);
    invScale (b, o);
    invScale (e, l);
    invScale (f, k);

    2x2T_h_POST (a, d, m, p, 0);
    2x2T_h_POST (b, c, n, o, 0);
    2x2T_h_POST (e, h, i, l, 0);
    2x2T_h_POST (f, g, j, k, 0);
}
```

Figure 4.14: Pseudo-code of 4x4 Inverse Photo Overlap Filter [1]

Chapter 5

Implementation

5.1 Inverse Lapped Biorthogonal Transform

The Inverse Lapped Biorthogonal Transform is implemented as seen in Figure 5.1. It is composed of 99 pipelined stages with 736 total instances of addition / subtraction. The two rightmost columns of data output from each IPCT are sent to RAM so that they can be used in the IPOF after the IPCT completes for the macroblock / block of the adjacent column. An external controller module is used to begin the transform upon assertion of the input sync pulse and to direct signals to the appropriate modules at the correct clock cycle. This controller is also responsible for asserting the output signal data_valid while valid transformed pixels are present on the ILBT data outputs.

5.2 Inverse Photo Core Transform

5.2.1 4x4 Inverse Photo Core Transform

The 4x4 Inverse Photo Core Transform is implemented as seen in Figure 5.2. It is composed of 18 pipelined stages with 107 total instances of addition / subtraction. In order to facilitate a pipelined design, additional registers were added so that all related outputs from one section would arrive at the next section on the same clock cycle.

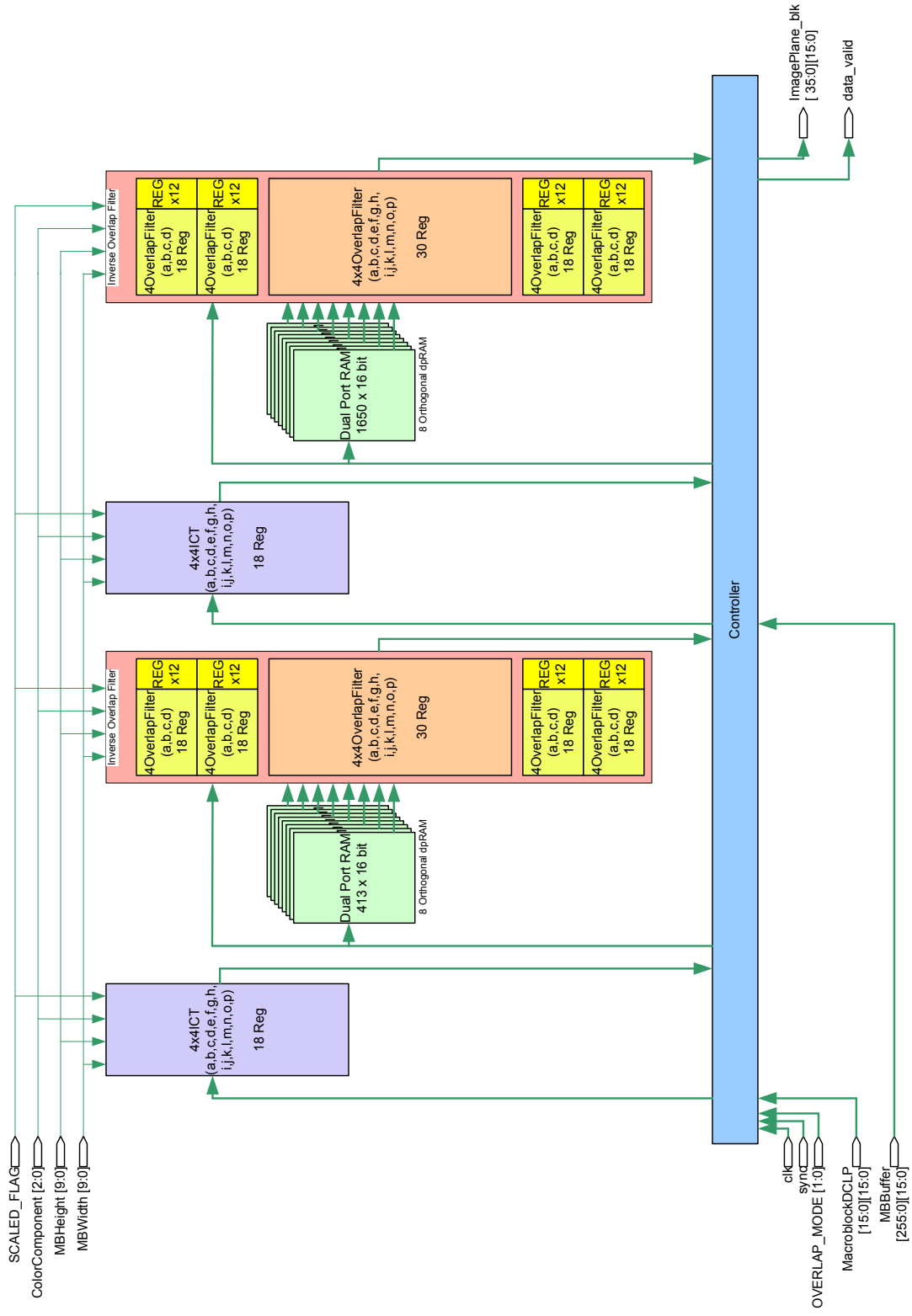

Figure 5.1: Structural Implementation of Inverse Lapped Biorthogonal Transform

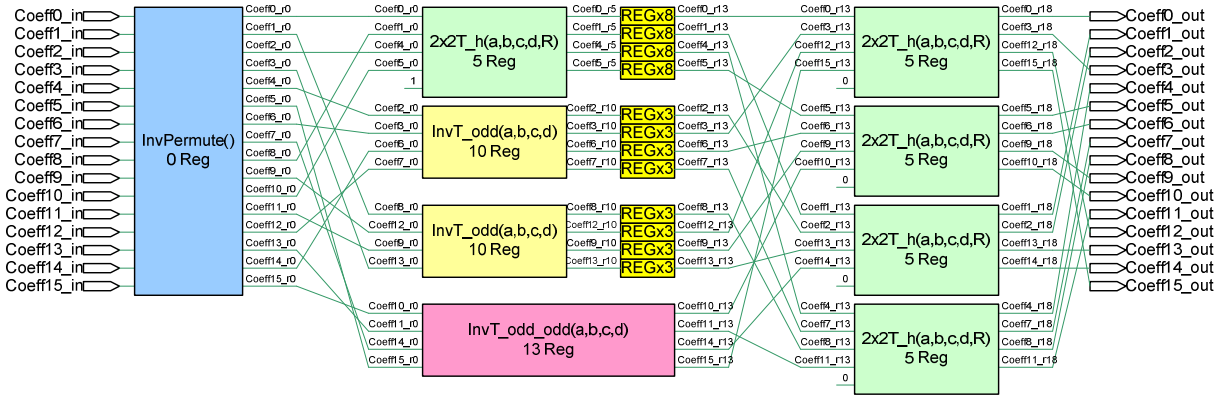

Figure 5.2: Structural Implementation of 4x4 Inverse Photo Core Transform

5.2.2 4x4 Inverse Permutation

The 4x4 Inverse Permutation is implemented as seen in Figure 5.3. This shuffling does not require any pipelined stages and does not include any instances of addition / subtraction. This function is implemented as a reassignment of signals in name only.

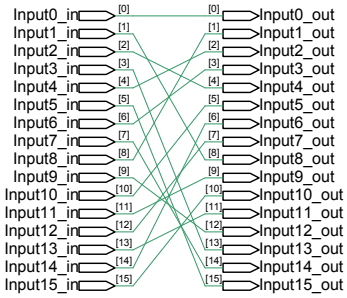

Figure 5.3: Structural Implementation of 4x4 Inverse Permutation

5.2.3 2x2 Inverse Hadamard Transform

The 2x2 Inverse Hadamard Transform is implemented as seen in Figure 5.4. It is composed of 5 pipelined stages with 8 instances of addition / subtraction.

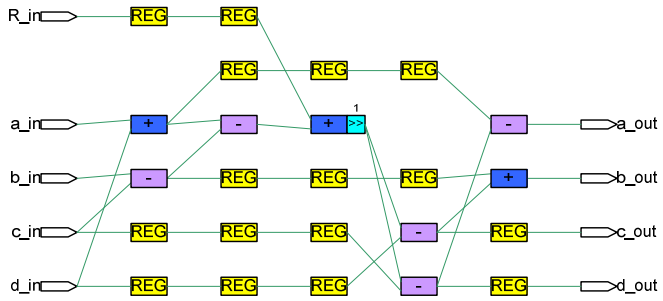

Figure 5.4: Structural Implementation of 2x2 Inverse Hadamard Transform

5.2.4 2x2 Inverse Odd Transform

The 2x2 Inverse Odd Transform is implemented as seen in Figure 5.5. It is composed of 10 pipelined stages with 24 instances of addition / subtraction.

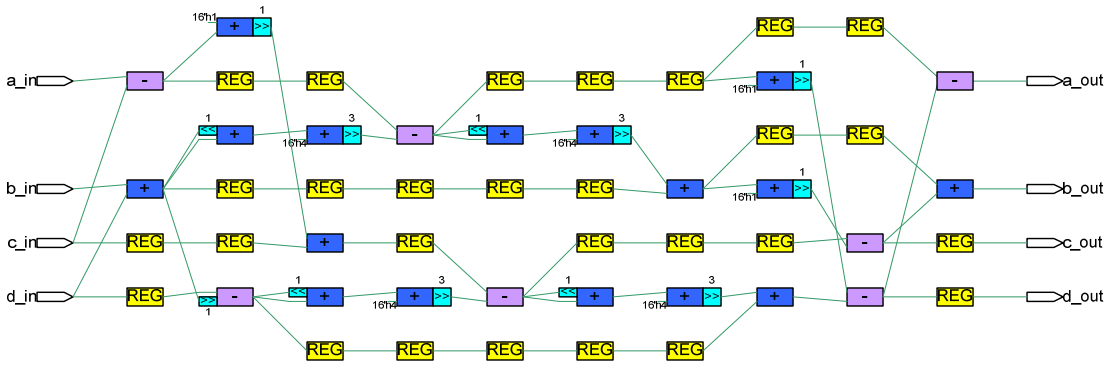

Figure 5.5: Structural Implementation of 2x2 Inverse Odd Transform

5.2.5 2x2 Inverse Odd Odd Transform

The 2x2 Inverse Odd Odd Transform is implemented as seen in Figure 5.6. It is composed of 13 pipelined stages with 19 instances of addition / subtraction.

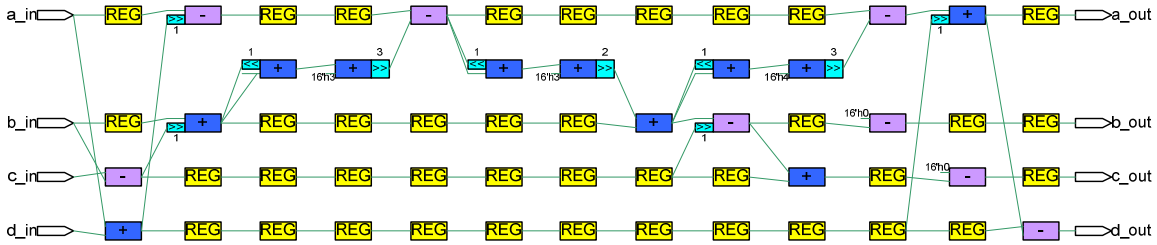

Figure 5.6: Structural Implementation of 2x2 Inverse Odd Odd Transform

5.3 Inverse Photo Overlap Filter

5.3.1 4x4 Inverse Photo Overlap Filter

The 4x4 Inverse Photo Overlap Filter is implemented as seen in Figure 5.7. It is composed of 30 pipelined stages with 125 total instances of addition / subtraction. Similar to the implementation of the 4x4 IPCT, additional registers are used to line up all outputs of a section on the same number of clock cycles so that the 4x4 IPOF can be pipelined.

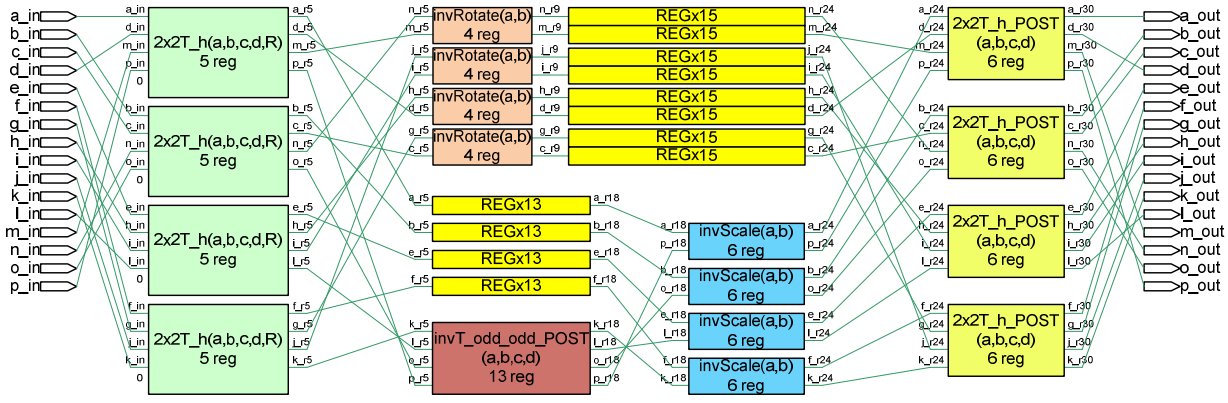

Figure 5.7: Structural Implementation of 4x4 Inverse Photo Overlap Filter

5.3.2 4-Point Inverse Photo Overlap Filter

The 4-Point Inverse Photo Overlap Filter is implemented as seen in Figure 5.8. It is composed of 18 pipelined stages with 34 instances of addition / subtraction. It utilizes the 2-Point Inverse Rotate function.

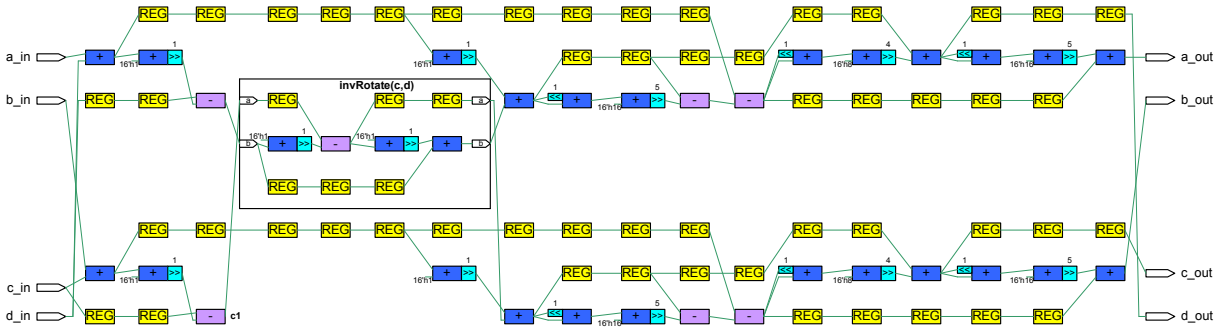

Figure 5.8: Structural Implementation of 4-Point Inverse Photo Overlap Filter

5.3.3 2x2 Inverse Post Hadamard Transform

The 2x2 Inverse Post Hadamard Transform is implemented as seen in Figure 5.9. It is composed of 6 pipelined stages with 9 instances of addition / subtraction.

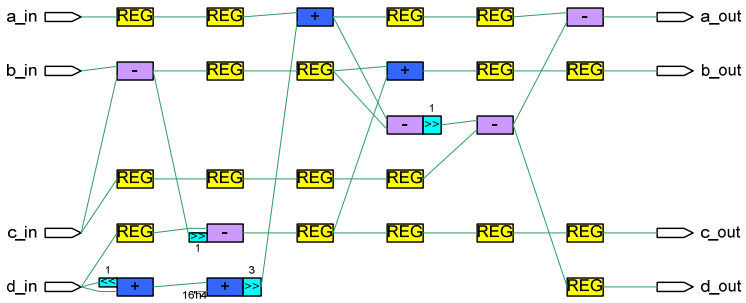

Figure 5.9: Structural Implementation of 2x2 Inverse Post Hadamard Transform

5.3.4 2x2 Inverse Odd Odd Post Transform

The 2x2 Inverse Odd Odd Post Transform is implemented as seen in Figure 5.10. It is composed of 13 pipelined stages with 17 instances of addition / subtraction.

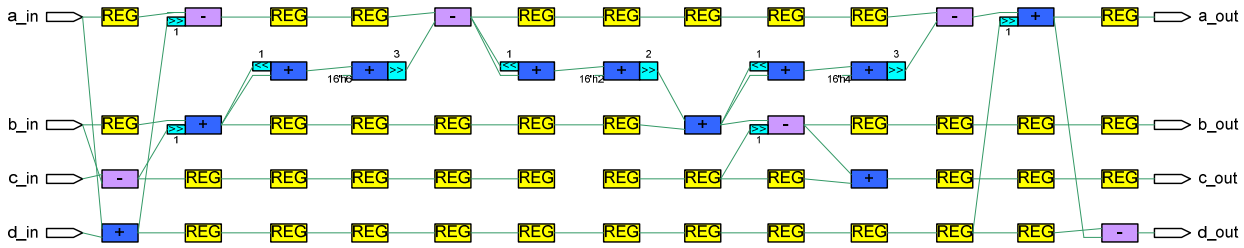

Figure 5.10: Structural Implementation of 2x2 Inverse Odd Odd Post Transform

5.3.5 2-Point Inverse Rotation

The 2-Point Inverse Rotation function is implemented as seen in Figure 5.11. It is composed of 4 pipelined stages with 4 instances of addition / subtraction.

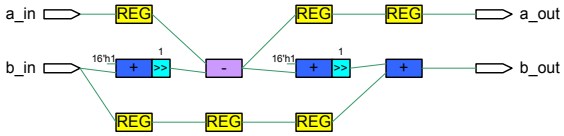

Figure 5.11: Structural Implementation of 2-Point Inverse Rotation

5.3.6 2-Point Inverse Scale

The 2-Point Inverse Scale function is implemented as seen in Figure 5.12. It is composed of 6 pipelined stages with 6 instances of addition / subtraction.

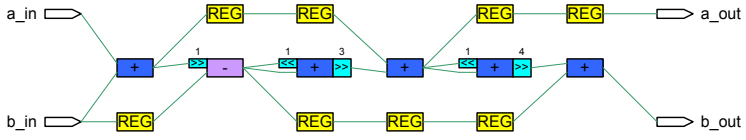

Figure 5.12: Structural Implementation of 2-Point Inverse Scale

Chapter 6

Verification

6.1 Software Model

The Inverse Lapped Biorthogonal Transform was modeled in software using the C programming language for the main purpose of verifying functionality of the VHDL code. Each functional block of the transform is separately coded as a C-function so that verification can be performed incrementally and individually by function. The top level C-code generates a series of test vector inputs for each function, and then captures both the inputs and resulting outputs in a file. This file is then used by the VHDL testbench to stimulate the VHDL code with predefined test vectors and to verify results against the known expected outputs. The top level C-code also generates a test-image for the purpose of verifying the entire inverse lapped biorthogonal transform. This test-image provides the simulated data representing the transform coefficients of an entire image plane for an image. It accepts parameters to define the macroblock width and macroblock height of the desired test image, so that a variety of image sizes can easily be generated for testing.

6.2 VHDL Testbench

A testbench was written in VHDL to verify each of the transform functions individually, and then a separate testbench was written to verify the entire inverse lapped biorthogonal transform at a top level.

6.2.1 Module Level Testbench

The module level testbench assumes that there exists a set of files containing appropriate input vectors and the corresponding output vectors for each function. It is written to serially step through each separate function, calling a common procedure to handle the input assignments and output verification checking of the functional module. This procedure takes in the number of registers in the pipeline of the functional module, the test-data file to be used, and the input and output signals connected to the component being tested. It is vital to know the number of registers in the pipeline so that it can be known when the outputs of the component should be tested against the output vectors supplied in the test-data file. Any discrepancies between the outputs of the VHDL under test and those specified in the test-data file are reported to the command line with a severity of failure (and thus allowing a stoppage of the test when configured as such in ModelSim). The VHDL inputs, VHDL outputs, and expected outputs are all reported to the command line for manual inspection by the user. Comparisons that correctly match the outputs in the test-data file can be reported to the command line with a severity of note, when configured as such in ModelSim. This allows the user to configure ModelSim to report the status of all tests performed, only those that succeed, or only those that fail.

6.2.2 Top-Level Testbench

The top-level testbench assumes that there exists a test-file containing transform coefficients for an entire image plane. Parameters are accepted to define the width and height of the test image in terms of macroblocks (i.e. number of pixels times 16). The testbench design initially asserts the reset signal while it loads two signals with data provided from a test-image file to represent the array of DC and Low Pass coefficients, and the array of High Pass coefficients which are required by the JPEG-XR standard to be available at the time the Inverse Lapped Biorthogonal Transform is performed. A read only RAM interface is included in the testbench, so that the ILBT VHDL code can read from these arrays when

appropriate. After the arrays are loaded, the reset signal is deasserted and a sync pulse is asserted logic high for a single clock period to kick off the ILBT module. Upon assertion of the ILBT output signal data_valid, the testbench begins collecting the output data values by block and writing the values to a file. Once the ILBT output signal data_valid is deasserted, indicating completion, the collected output file can be compared to the known output values provided from the c-language software simulation using any text compare software.

6.3 WinMerge

Version 2.6.8.0 of WinMerge is used in this work to compare the VHDL generated output image and the C-language software output image, and to verify that the two files are identical. The output of both the software implementation and the VHDL testbench are text files containing single space delimited numbers with each row representative of pixels on a separate line. WinMerge is an open source file compare and merge utility which runs on all modern Windows versions.

6.4 Test Cases

For the top-level testbench, the software test-image generator described above is used to pseudo-randomly select at run-time the integer pixel values ranging from 0 to 255. Test-images are used for sizes of a minimum of 1 macroblock by 1 macroblock, a maximum of 40 macroblocks by 40 macroblocks, and several randomly selected sizes in between. Both the width and height are varied to verify that the model is correct for non square images. For all tests, the compare using WinMerge verified that the VHDL model correctly produces the same numerical results as the C-language software implementation.

Chapter 7

Results and Analysis

This implementation was targeted on a Stratix II EP2S180 FPGA. A comparison of results is provided for an implementation of the forward JPEG-XR transform with similar design constraints on a Stratix III EP3SL340F1760C3 FPGA [2].

7.1 Sizing

The Stratix II EP2S180 FPGA contains 179,400 equivalent logic elements, 143,520 Adaptive Look-Up Tables (ALUTs), 9,383,040 total block memory bits, and 1171 I/O pins [9]. This is a relatively large FPGA, and is mostly needed to incorporate the large number of I/O for the inverse transform module. However, when this module is used in conjunction with VHDL or Verilog implementations of the remaining decompression modules, a smaller FPGA might be afforded by a smaller number of I/O.

A comparison of the sizing results for the top level inverse transform, a single inverse photo core transform module, and a single inverse photo overlap filter is provided in Table 7.1.

Module	Combinational ALUTs	Logic Registers	Memory Bits
1x Inverse Photo Core Transform	1882	2363	2410
1x Inverse Photo Overlap Filter	4017	5900	8563
Top Level Inverse Lapped Biorthogonal Transform	12796	17908	141258

Table 7.1: Sizing Results

Table 7.2 shows that this implementation offers an improvement on [2] in terms of layout size, with the most significant reduction in memory bits used.

	Forward Transform	Inverse Transform	Reduction
Combinational ALUTs	20444	12796	37.41%
Logic Registers	77143	17908	76.79%
Memory Bits	7295232	141258	98.06%

Table 7.2: Size comparison of this ILBT implementation to forward LBT in [2]

The 98% reduction in memory is not quite the equal comparison, because the forward transform in [2] uses memory bits to buffer the entire image at both the input and output of the transform module. This ILBT implementation requires the previous and following modules to be responsible for storage. However there is still a reasonable reduction in the number of memory bits used, since this ILBT implementation removes the partial buffer between the first level IPOF and the second level IPCT. For the specified maximum image size, this reduces the memory by 413 blocks of orthogonal 16 bit RAM.

7.2 Speed

Speed on an FPGA is constrained by a combination of the specific devices architecture and the worst case path containing the most combinatorial logic between two registers. A design with very little combinatorial logic between registers is almost solely limited by the routing on the device. In contrast, it does not matter how fast a device can pass information between registers if it needs to wait for combinatorial logic to complete before it can continue. The TimeQuest Timing Analyzer included with the Quartus II Design Suite was used to calculate the maximum clock speed achievable for the top level inverse transform, a single inverse photo core transform module, and a single inverse photo overlap filter. The resulting speeds are provided in Table 7.3.

This implementation is pipelined such that the time it takes to complete the inverse transform of an image is merely a linear function of the number of pixels in the image.

Module	Max Clock Frequency
1x Inverse Photo Core Transform	361.79 MHz
1x Inverse Photo Overlap Filter	311.53 MHz
Top Level Inverse Lapped Biorthogonal Transform	116.47 MHz

Table 7.3: Timing Results

The JPEG-XR inverse transform algorithm almost inherently introduces a delay equal to the number of blocks in the height of the image (if iterating by column) or the width of the image (if iterating by row). This can be seen more easily in Figure 4.2. The IPCT operates on a 4x4 block of coefficients, but then the IPOF operates on only two of those rows and columns. The IPOF additionally requires the four corner coefficients from the next column and next row to have completed the IPCT. If iterating by column, the first fully inverse transformed block would therefore need to wait for the first column and the top two blocks in the second column to complete the IPOF. In order to mitigate this delay for a linear transform time, the implementation in this work outputs blocks on the IPOF boundaries. This means that only half blocks are output for the first column and one and a half blocks are output for the last column. Figure 7.1 shows the resulting completion times of various sized images from simulations with a 50 MHz clock. Notice that the completion time is only a function of the number of pixels and is not dependent on the number of pixels specifically in either orientation along the width or height.

Since there are 99 stages in the pipeline path, there is a delay of 99 clock cycles before the first valid inverse transformed pixel is output. For a 50 MHz clock, this is a 1980 ns delay. After this initial delay, a new valid inverse transformed block is output at each rising edge of the clock. Equation 7.1 can be used to determine the time required to complete the inverse transform for an image of any given size.

$$TimeToCompleteILBT = \left(\frac{NumPixelsInImage}{16} + 99\right) * CLK\_PERIOD \quad (7.1)$$

Table 7.4 shows that this implementation offers a slight improvement on [2] in the max

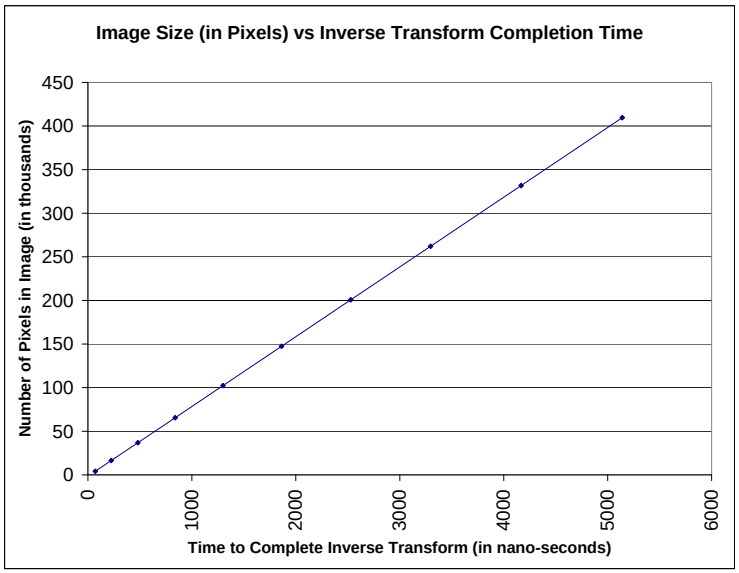

Figure 7.1: Relationship between Image Pixels and Completion Time

clock frequency that can be used. Since both designs are pipelined, an important metric is latency. This ILBT has 12 less clock cycles than [2] in the design of its pipeline.

	Forward Transform	Inverse Transform	Reduction
Max Clock Frequency (MHz)	115.34	116.47	-0.98%
Pipeline Latency (clock cycles)	111	99	10.81%

Table 7.4: Timing comparison of this ILBT implementation to forward LBT in [2]

7.3 Power

The PowerPlay Power Analyzer provided with the Quartus II Design Suite is used to generate an estimation of the power consumed by this design as programmed onto the FPGA. A comparison of the power results for the top level inverse transform, a single inverse photo core transform module, and a single inverse photo overlap filter is provided in Table 7.5. It should be noted that valid switching information was not provided and thus the resulting power estimates are not accurate for real world applications. The numbers are heavily I/O

weighted, and thus for the single IPCT and single IPOF, these numbers do not actually represent the power consumed when implemented within the overall ILBT. In a similar fashion, it is assumed that in most application scenarios the inverse transform would be used in concert with the rest of the JPEG-XR decompression procedure. This could in turn reduce the number of I/O required, and effectively allow for a smaller FPGA. A combination of a smaller FPGA and less I/O should significantly reduce the power consumption.

Module	I/O Thermal Power Dissipation (mW)	Core Dynamic Power Thermal Dissipation (mW)	Core Static Power Thermal Dissipation (mW)	Total Thermal Power Dissipation (mW)
1x Inverse Photo Core Transform	55.00	0.00	1366.55	1421.54
1x Inverse Photo Overlap Filter	73.61	0.00	1366.87	1440.48
Top Level Inverse Lapped Biorthogonal Transform	78.29	0.00	1366.96	1445.25

Table 7.5: Power Results

Valid switching information is not provided for this ILBT or [2], and therefore a power comparison is irrelevant. However Table 7.6 shows a comparison mostly representative of the difference in FPGA sizes.

	Forward Transform	Inverse Transform	Reduction
I/O Thermal Power Dissipation (mW)	78.29	42.61	45.57%
Core Dynamic Power Thermal Dissipation (mW)	0	0	0.00%
Core Static Power Thermal Dissipation (mW)	1366.96	1339	2.05%
Total Thermal Power Dissipation (mW)	1445.25	1381.62	4.40%

Table 7.6: Power comparison of this ILBT implementation to forward LBT in [2]

Chapter 8

Future Work and Conclusions

8.1 Problems Encountered

The JPEG-XR standard has not yet been officially released, and therefore a working draft of the standard was used to research the algorithms and implement functionality. This meant at often times identifying and correcting mistakes in the working draft. These misprints occasionally caused major rework when not caught early enough in the design and especially when they had an impact on the number of clock cycles of an operation.

8.2 Future Work

There are many ways to implement a design, and for the inverse lapped biorthogonal transform used in JPEG-XR, it is no exception. This work describes a single implementation of the inverse transform with a focus on pipelined design and reduced area and memory usage. Possible optimizations to this design include adding a second IPCT block to the second level of the transform in order to output an entire valid block during each output clock. Other work may be done to implement the remaining stages of the decompression algorithm in order to compare a full hardware implementation of the decompression algorithm.

The first optimization is one that would allow standardizing the data output from the inverse transform module. The implementation described in this work does not output

consistent amounts of data. For the first column of transformed blocks, only the left-most two columns of pixels are available. Outputs for the remaining columns of blocks are output along the boundaries of the inverse photo overlap filter. This means that the outputs for the last column of blocks will actually present six columns of pixels (those four associated with the inverse photo overlap filter and the remaining right-most two columns of pixels). This could be corrected by adding an extra block of memory and incorporating a delay in receiving all valid outputs equivalent to the number of blocks in a column of the image. This could also be corrected by sacrificing a larger number of registers and including an additional IPCT in the second stage of the inverse transform. This would introduce only a single clock delay, as the IPOF may begin in its entirety without waiting (a column of delay) for the necessary adjacent blocks to go through an IPCT.

It would also be beneficial to have the entire decompression process implemented on hardware. This would allow a greater comparison of JPEG-XR to existing formats and might expose some not yet seen benefits or drawbacks to a hardware implementation of this format. A full hardware decompression implementation could also be used in concert with the hardware implementation of the JPEG-XR encoder described in [2] to obtain a single chip capable of JPEG-XR compression and decompression.

8.3 Conclusions

This thesis work has explored an implementation of the JPEG-XR ILBT with a relatively small memory and logic footprint. One of the goals of this work was to compare the design and implementation of the inverse transform to the forward transform used in JPEG-XR compression. It was found that the order of operations allow the inverse transform to be implemented with much less memory than a similarly constrained forward transform. Implementation bottlenecks and synthesis results have been provided in hopes that future implementations of the JPEG-XR ILBT and even newer inverse transforms may continue to push the development of a decompression algorithm that can be implemented with small

size, high speed, and low power.

References

[1] G. J. Sullivan, "ISO/IEC JTC 1/SC 29/WG 1 Coding of Still Pictures," 2007.

[2] S. Groder, "Verilog modeling and synthesis of the hd photo compression algorithm," Master's thesis, Rochester Institute of Technology, 2008.

[3] F. Tse, "An Introduction to HD Photo." Xerox Corporation, September 2007.

[4] S. Srinivasan, "An Introduction to HD Photo." Microsoft Corporation, March 2007.

[5] S. Srinivasan, C. Tu, Z. Zhou, D. Ray, S. Regunathan, and G. Sullivan, "An Introduction to the HD Photo Technical Design." Microsoft Corporation.

[6] Joint Photographic Experts Group, "JPEG Homepage," July 2009. http://www.jpeg.org/jpeg/index.html.

[7] D. S. Cruz, T. Ebrahimi, J. Askelof, M. Larsson, and C. Christopoulos, "An analytical study of JPEG 2000 functionalities," in *Proc. Of the IEEE International Conference on Acoustics, Speech and Signal Processing*, July 2000.

[8] D. L. Gall and A. Tabatabai, "Sub-band coding of digital images using symmetric short kernel filters and arithmetic coding techniques," in *Proc. Of the IEEE International Conference on Acoustics, Speech and Signal Processing*, (New York, NY), pp. 761–765, 1988.

[9] Altera Corporation, "Altera Product Catalog," 2009. www.altera.com.

Appendix A

VHDL Code

```
--*********************************************************************************
--    File Name:  InverseILBT.vhd
--      Version:  01.00
--         Date:  July 18, 2009
--
--  Description:  Top Level of Inverse Lapped Biorthogonal Transform (ILBT)
--
--       Author:  Peter A. Frandina
--       School:  Rochester Institute of Technology
--      Project:  Thesis - HD Photo Decompression ILBT Algorithm
--*********************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use work.data_types_pkg.all;

entity InverseLBT is
  port(
    clk        : in  std_logic;
    reset      : in  std_logic;
    sync       : in  std_logic; --enable signal to begin processing
    data_valid : out std_logic; --indicates the module is outputting the image.

    --signals for reading the MacroblockDCLP variable input to this module.
    DCLP_in_data        : in type_Macroblock16;
    rd_DCLP_MBx_addr    : out std_logic_vector(9 downto 0); --output
    rd_DCLP_MBy_addr    : out std_logic_vector(9 downto 0); --output

    --signals for reading the MBBuffer variable input to this module.
    MBBuff_in_data      : in  type_Macroblock16;
    rd_MBBuff_MBx_addr  : out std_logic_vector(9 downto 0); --output
    rd_MBBuff_MBy_addr  : out std_logic_vector(9 downto 0); --output
    rd_MBBuff_Bx        : out integer range 0 to 3;         --output
    rd_MBBuff_By        : out integer range 0 to 3;         --output

    --Parameters
```

```
    MBWidth             : in  std_logic_vector(9 downto 0); --range 0->1024
    MBHeight            : in  std_logic_vector(9 downto 0); --range 0->1024
    NumComponent        : in  std_logic_vector(3 downto 0); --range 0->15 (0=lumma, 1-15=chroma)
    OVERLAP_MODE        : in  std_logic_vector(1 downto 0); --range 0->3
    INTERNAL_CLR_FMT    : in  std_logic_vector(2 downto 0); --range 0->7
    SCALED_FLAG         : in  std_logic;

    i                   : in  integer range 0 to 15;--(0=lumma, 1-15=chroma)

    --ImagePlane Outputs
    interior_out        : out type_Macroblock16;
    corner              : out type_Macroblock04;
    toprowww_rightcol_0 : out type_Macroblock04;
    toprowww_rightcol_1 : out type_Macroblock04;
    leftcoll_botomrow_0 : out type_Macroblock04;
    leftcoll_botomrow_1 : out type_Macroblock04
    );
end entity;

architecture beh of InverseLBT is
  --enable signals
  signal sync_1IT          : std_logic;
  signal data_valid_1IT    : std_logic;
  signal sync_1OF          : std_logic;
  signal data_valid_1OF    : std_logic;
  signal data_valid_m1_1OF : std_logic;
  signal sync_2IT          : std_logic;
  signal data_valid_2IT    : std_logic;
  signal sync_2OF          : std_logic;
  signal data_valid_2OF    : std_logic;

  --data signals
  --1IT data signals
  signal data_in_1IT_MB16  : type_Macroblock16;
  signal data_out_1IT_MB16 : type_Macroblock16;
  --1OF data signals
  signal data_in_1OF_interior_0_MB16          : type_Macroblock16;
  signal data_in_1OF_toprowww_rightcol_0_MB04  : type_Macroblock04;
  signal data_in_1OF_toprowww_rightcol_1_MB04  : type_Macroblock04;
  signal data_in_1OF_leftcoll_botomrow_0_MB04  : type_Macroblock04;
  signal data_in_1OF_leftcoll_botomrow_1_MB04  : type_Macroblock04;
  signal data_out_1OF_interior_0_MB16          : type_Macroblock16;
  signal data_out_1OF_toprowww_rightcol_0_MB04 : type_Macroblock04;
  signal data_out_1OF_toprowww_rightcol_1_MB04 : type_Macroblock04;
  signal data_out_1OF_leftcoll_botomrow_0_MB04 : type_Macroblock04;
  signal data_out_1OF_leftcoll_botomrow_1_MB04 : type_Macroblock04;

  --dpram signals for ram1
  signal wr1addr_sig : std_logic_vector(15 downto 0);
  signal rd1addr_sig : std_logic_vector(15 downto 0);
  signal wr1data_sig : type_arr08_slvMSBbit;
  signal rd1data_sig : type_arr08_slvMSBbit;
  signal we1_sig     : std_logic;
```

```
  signal re1_sig     : std_logic;

  --2IT data signals
  signal data_in_2IT_MB16  : type_Macroblock16;
  signal data_out_2IT_MB16 : type_Macroblock16;
  --2OF data signals
  signal data_in_2OF_interior_0_MB16           : type_Macroblock16;
  signal data_in_2OF_toprowww_rightcol_0_MB04  : type_Macroblock04;
  signal data_in_2OF_toprowww_rightcol_1_MB04  : type_Macroblock04;
  signal data_in_2OF_leftcoll_botomrow_0_MB04  : type_Macroblock04;
  signal data_in_2OF_leftcoll_botomrow_1_MB04  : type_Macroblock04;
  signal data_out_2OF_interior_0_MB16          : type_Macroblock16;
  signal data_out_2OF_toprowww_rightcol_0_MB04 : type_Macroblock04;
  signal data_out_2OF_toprowww_rightcol_1_MB04 : type_Macroblock04;
  signal data_out_2OF_leftcoll_botomrow_0_MB04 : type_Macroblock04;
  signal data_out_2OF_leftcoll_botomrow_1_MB04 : type_Macroblock04;

  --dpram signals for ram2
  signal wr2addr_sig : std_logic_vector(15 downto 0);
  signal rd2addr_sig : std_logic_vector(15 downto 0);
  signal wr2data_sig : type_arr08_slvMSBbit;
  signal rd2data_sig : type_arr08_slvMSBbit;
  signal we2_sig     : std_logic;
  signal re2_sig     : std_logic;

begin

u_controller : entity work.controller(rtl)
port map(
  clk                             => clk,
  reset                           => reset,
  sync                            => sync,          --input
  sync_1IT                        => sync_1IT,
  done_1IT                        => data_valid_1IT,
  sync_1OF                        => sync_1OF,
  done_1OF                        => data_valid_1OF,
  done_m1_1OF                     => data_valid_m1_1OF,
  sync_2IT                        => sync_2IT,
  done_2IT                        => data_valid_2IT,
  sync_2OF                        => sync_2OF,
  done_2OF                        => data_valid_2OF,
  done                            => data_valid,    --output

  MBHeight                        => MBHeight,
  MBWidth                         => MBWidth,
  --Parameter Inputs
  OVERLAP_MODE                    => OVERLAP_MODE,

  --data read from the DCLP variable
  data_in                         => DCLP_in_data,      --input
  rd_DCLP_MBx_addr                => rd_DCLP_MBx_addr,  --output
  rd_DCLP_MBy_addr                => rd_DCLP_MBy_addr,  --output
```

```
--data read from the MBBuffer variable
MBBuff_in_data                   => MBBuff_in_data,    --input
rd_MBBuff_MBx_addr               => rd_MBBuff_MBx_addr,--output
rd_MBBuff_MBy_addr               => rd_MBBuff_MBy_addr,--output
rd_MBBuff_Bx                     => rd_MBBuff_Bx,      --output
rd_MBBuff_By                     => rd_MBBuff_By,      --output

--1IT data signals
data_in_1IT                      => data_in_1IT_MB16,
data_out_1IT                     => data_out_1IT_MB16,
--1OF data signals
data_in_1OF_interior_0           => data_in_1OF_interior_0_MB16,
data_in_1OF_toprowww_rightcol_0  => data_in_1OF_toprowww_rightcol_0_MB04,
data_in_1OF_toprowww_rightcol_1  => data_in_1OF_toprowww_rightcol_1_MB04,
data_in_1OF_leftcoll_botomrow_0  => data_in_1OF_leftcoll_botomrow_0_MB04,
data_in_1OF_leftcoll_botomrow_1  => data_in_1OF_leftcoll_botomrow_1_MB04,
data_out_1OF_interior_0          => data_out_1OF_interior_0_MB16,
data_out_1OF_toprowww_rightcol_0 => data_out_1OF_toprowww_rightcol_0_MB04,
data_out_1OF_toprowww_rightcol_1 => data_out_1OF_toprowww_rightcol_1_MB04,
data_out_1OF_leftcoll_botomrow_0 => data_out_1OF_leftcoll_botomrow_0_MB04,
data_out_1OF_leftcoll_botomrow_1 => data_out_1OF_leftcoll_botomrow_1_MB04,

--2IT data signals
data_in_2IT                      => data_in_2IT_MB16,
data_out_2IT                     => data_out_2IT_MB16,
--2OF data signals
data_in_2OF_interior_0           => data_in_2OF_interior_0_MB16,
data_in_2OF_toprowww_rightcol_0  => data_in_2OF_toprowww_rightcol_0_MB04,
data_in_2OF_toprowww_rightcol_1  => data_in_2OF_toprowww_rightcol_1_MB04,
data_in_2OF_leftcoll_botomrow_0  => data_in_2OF_leftcoll_botomrow_0_MB04,
data_in_2OF_leftcoll_botomrow_1  => data_in_2OF_leftcoll_botomrow_1_MB04,
data_out_2OF_interior_0          => data_out_2OF_interior_0_MB16,
data_out_2OF_toprowww_rightcol_0 => data_out_2OF_toprowww_rightcol_0_MB04,
data_out_2OF_toprowww_rightcol_1 => data_out_2OF_toprowww_rightcol_1_MB04,
data_out_2OF_leftcoll_botomrow_0 => data_out_2OF_leftcoll_botomrow_0_MB04,
data_out_2OF_leftcoll_botomrow_1 => data_out_2OF_leftcoll_botomrow_1_MB04,

--signals for ram1
wr1addr            => wr1addr_sig,
wr1data            => wr1data_sig,
we1                => we1_sig,
rd1addr            => rd1addr_sig,
rd1data            => rd1data_sig,
re1                => re1_sig,
--signals for ram2
wr2addr            => wr2addr_sig,
wr2data            => wr2data_sig,
we2                => we2_sig,
rd2addr            => rd2addr_sig,
rd2data            => rd2data_sig,
re2                => re2_sig,

--ImagePlane outputs
```

```
  interior_out       => interior_out,
  corner             => corner,
  toprowww_rightcol_0 => toprowww_rightcol_0,
  toprowww_rightcol_1 => toprowww_rightcol_1,
  leftcoll_botomrow_0 => leftcoll_botomrow_0,
  leftcoll_botomrow_1 => leftcoll_botomrow_1
  );


--The number of clocks it takes to perform the FirstLevelInverseTransform
--depends on the parameter "INTERNAL_CLR_FMT":
--                            when YUV422 = 8 Reg,
--                            when YUV420 = 5 Reg,
--                            when Others = 18 Reg
u_FirstLevelInverseTransform : entity work.InversePhotoCoreTransform(rtl)
port map(
  --Control logic
  clk                => clk,            --in
  reset              => reset,          --in
  sync               => sync_1IT,       --in
  data_valid         => data_valid_1IT, --out
  --Data Inputs
  Macroblock16_in    => data_in_1IT_MB16,
  i                  => i,
  --Parameter Inputs
  INTERNAL_CLR_FMT   => INTERNAL_CLR_FMT,
  SCALED_FLAG        => SCALED_FLAG,
  --Data Outputs
  Macroblock16_out   => data_out_1IT_MB16
  );

gen_dpram1 : for gen1_cnt in 0 to 7 generate
  u1_dpram : entity work.dpram_MBHeight_x_16(rtl)
  port map(
    CLK   => clk,
    WE    => we1_sig,
    WADDR => wr1addr_sig,
    DIN   => wr1data_sig(gen1_cnt),
    RE    => re1_sig,
    RADDR => rd1addr_sig,
    DOUT  => rd1data_sig(gen1_cnt)
    );
end generate;

--The number of clocks it takes to perform the FirstLevelOverlapFilter is 30
u_FirstLevelOverlapFilter : entity work.InverseOverlapFilter(rtl)
port map(
  --Control logic
  clk                 => clk,              --in
  reset               => reset,            --in
  sync                => sync_1OF,         --in
  data_valid          => data_valid_1OF,   --out
  data_valid_m1       => data_valid_m1_1OF, --out
```

```
  --Data Inputs
  i_data_interior_0          => data_in_1OF_interior_0_MB16,
  i_data_toprowww_rightcol_0 => data_in_1OF_toprowww_rightcol_0_MB04,
  i_data_toprowww_rightcol_1 => data_in_1OF_toprowww_rightcol_1_MB04,
  i_data_leftcoll_botomrow_0 => data_in_1OF_leftcoll_botomrow_0_MB04,
  i_data_leftcoll_botomrow_1 => data_in_1OF_leftcoll_botomrow_1_MB04,

  i                   => i,
  --Parameter Inputs
  INTERNAL_CLR_FMT    => INTERNAL_CLR_FMT,
  --Data Outputs
  o_data_interior_0          => data_out_1OF_interior_0_MB16,
  o_data_toprowww_rightcol_0 => data_out_1OF_toprowww_rightcol_0_MB04,
  o_data_toprowww_rightcol_1 => data_out_1OF_toprowww_rightcol_1_MB04,
  o_data_leftcoll_botomrow_0 => data_out_1OF_leftcoll_botomrow_0_MB04,
  o_data_leftcoll_botomrow_1 => data_out_1OF_leftcoll_botomrow_1_MB04
  );


--The number of clocks it takes to perform the SecondLevelInverseTransform
--depends on the parameter "INTERNAL_CLR_FMT":
--                            when YUV422 = 8 Reg,
--                            when YUV420 = 5 Reg,
--                            when Others = 18 Reg
u_SecondLevelInverseTransform : entity work.InversePhotoCoreTransform(rtl)
port map(
  --Control logic
  clk                => clk,            --in
  reset              => reset,          --in
  sync               => sync_2IT,       --in
  data_valid         => data_valid_2IT, --out
  --Data Inputs
  Macroblock16_in    => data_in_2IT_MB16,
  i                  => i,
  --Parameter Inputs
  INTERNAL_CLR_FMT   => INTERNAL_CLR_FMT,
  SCALED_FLAG        => SCALED_FLAG,
  --Data Outputs
  Macroblock16_out   => data_out_2IT_MB16
  );

gen_dpram2 : for gen2_cnt in 0 to 7 generate
  u2_dpram : entity work.dpram_BlockHeight_x_16(rtl)
  port map(
    CLK   => clk,
    WE    => we2_sig,
    WADDR => wr2addr_sig,
    DIN   => wr2data_sig(gen2_cnt),
    RE    => re2_sig,
    RADDR => rd2addr_sig,
    DOUT  => rd2data_sig(gen2_cnt)
    );
end generate;
```

```
--The number of clocks it takes to perform the SecondLevelOverlapFilter is 30
u_SecondLevelOverlapFilter : entity work.InverseOverlapFilter(rtl)
port map(
  --Control logic
  clk                 => clk,            --in
  reset               => reset,          --in
  sync                => sync_2OF,       --in
  data_valid          => data_valid_2OF, --out
  data_valid_m1       => open, --out, not used
  --Data Inputs
  i_data_interior_0          => data_in_2OF_interior_0_MB16,
  i_data_toprowww_rightcol_0 => data_in_2OF_toprowww_rightcol_0_MB04,
  i_data_toprowww_rightcol_1 => data_in_2OF_toprowww_rightcol_1_MB04,
  i_data_leftcoll_botomrow_0 => data_in_2OF_leftcoll_botomrow_0_MB04,
  i_data_leftcoll_botomrow_1 => data_in_2OF_leftcoll_botomrow_1_MB04,

  i                   => i,
  --Parameter Inputs
  INTERNAL_CLR_FMT    => INTERNAL_CLR_FMT,
  --Data Outputs
  o_data_interior_0          => data_out_2OF_interior_0_MB16,
  o_data_toprowww_rightcol_0 => data_out_2OF_toprowww_rightcol_0_MB04,
  o_data_toprowww_rightcol_1 => data_out_2OF_toprowww_rightcol_1_MB04,
  o_data_leftcoll_botomrow_0 => data_out_2OF_leftcoll_botomrow_0_MB04,
  o_data_leftcoll_botomrow_1 => data_out_2OF_leftcoll_botomrow_1_MB04
  );

end architecture;

--*********************************************************************************
--    File Name:  InversePhotoCoreTransform_YUV444.vhd
--      Version:  01.00
--         Date:  July 18, 2009
--
--  Description:  Inverse Photo Core Transform (IPCT) for YUV444 format
--
--       Author:  Peter A. Frandina
--       School:  Rochester Institute of Technology
--      Project:  Thesis - HD Photo Decompression ILBT Algorithm
--*********************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use work.data_types_pkg.all;

entity InversePhotoCoreTransform_YUV444 is
  port(
    --Control logic
    clk               : in  std_logic;
    reset             : in  std_logic;
    --Data Inputs
    Macroblock16_in   : in  type_Macroblock16;
    i                 : in integer range 0 to 15;--(0=lumma, 1-15=chroma)
```

```
    --Parameter Inputs
    INTERNAL_CLR_FMT   : in  std_logic_vector(2 downto 0);
    SCALED_FLAG        : in  std_logic;
    --Data Outputs
    Macroblock16_out   : out type_Macroblock16
    );
end entity;

architecture rtl of InversePhotoCoreTransform_YUV444 is
  signal limited_sig    : std_logic_vector(15 downto 0); --detect overflow
  signal MB_sig         : type_Macroblock16;
  signal Macroblock_sig : type_Macroblock16;
begin

--concurrent output assignment
Macroblock16_out <= Macroblock_sig;

--This generated process scales the output when appropriate.
scaleYUV444_gen : for gen_cnt in 0 to 15 generate
  process(i,SCALED_FLAG,MB_sig(gen_cnt))
  begin
    if (i>0) and (SCALED_FLAG='1') then--chroma component and scaled arithmetic
      --multiply each by 2
      if MB_sig(gen_cnt)(MSB) = '1' then --we have a negative value
        if MB_sig(gen_cnt)(MSB-1) = '1' then
          Macroblock_sig(gen_cnt) <= MB_sig(gen_cnt)((MSB-1) downto 0) & '0';
        else
          Macroblock_sig(gen_cnt) <= MAX_NEG_MSB;  --limit to max negative
          limited_sig(gen_cnt) <= '1';
        end if;
      else --we have a positive value
        if MB_sig(gen_cnt)(MSB-1) = '0' then
          Macroblock_sig(gen_cnt) <= MB_sig(gen_cnt)((MSB-1) downto 0) & '0';
        else
          Macroblock_sig(gen_cnt) <= MAX_POS_MSB;  --limit to max positive
          limited_sig(gen_cnt) <= '1';
        end if;
      end if;
    else   --do not multiply by 2 (i.e. do not scale)
      Macroblock_sig(gen_cnt) <= MB_sig(gen_cnt);
    end if;--end multiply by 2
  end process;
end generate;

u_4x4ICT : entity work.funct_4x4ICT(rtl) port map(
  clk   => clk  ,
  reset => reset,
  Coeff0_in   => Macroblock16_in( 0),
  Coeff1_in   => Macroblock16_in( 1),
  Coeff2_in   => Macroblock16_in( 2),
  Coeff3_in   => Macroblock16_in( 3),
  Coeff4_in   => Macroblock16_in( 4),
  Coeff5_in   => Macroblock16_in( 5),
```

```
  Coeff6_in   => Macroblock16_in( 6),
  Coeff7_in   => Macroblock16_in( 7),
  Coeff8_in   => Macroblock16_in( 8),
  Coeff9_in   => Macroblock16_in( 9),
  Coeff10_in  => Macroblock16_in(10),
  Coeff11_in  => Macroblock16_in(11),
  Coeff12_in  => Macroblock16_in(12),
  Coeff13_in  => Macroblock16_in(13),
  Coeff14_in  => Macroblock16_in(14),
  Coeff15_in  => Macroblock16_in(15),
  Coeff0_out  => MB_sig( 0),
  Coeff1_out  => MB_sig( 1),
  Coeff2_out  => MB_sig( 2),
  Coeff3_out  => MB_sig( 3),
  Coeff4_out  => MB_sig( 4),
  Coeff5_out  => MB_sig( 5),
  Coeff6_out  => MB_sig( 6),
  Coeff7_out  => MB_sig( 7),
  Coeff8_out  => MB_sig( 8),
  Coeff9_out  => MB_sig( 9),
  Coeff10_out => MB_sig(10),
  Coeff11_out => MB_sig(11),
  Coeff12_out => MB_sig(12),
  Coeff13_out => MB_sig(13),
  Coeff14_out => MB_sig(14),
  Coeff15_out => MB_sig(15)
  );

end architecture;

--*********************************************************************************
--    File Name:  InverseOverlapFilter.vhd
--      Version:  01.00
--         Date:  July 18, 2009
--
--  Description:  Inverse Photo Overlap Filter (IPOF) Top
--
--       Author:  Peter A. Frandina
--       School:  Rochester Institute of Technology
--      Project:  Thesis - HD Photo Decompression ILBT Algorithm
--*********************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use work.data_types_pkg.all;

entity InverseOverlapFilter is
  port(
    --Control logic
    clk                       : in  std_logic;
    reset                     : in  std_logic;
    sync                      : in  std_logic;
    data_valid                : out std_logic;
    data_valid_m1             : out std_logic;
```

```
    --Data Inputs
    i_data_interior_0          : in  type_Macroblock16;
    i_data_toprowww_rightcol_0 : in  type_Macroblock04;
    i_data_toprowww_rightcol_1 : in  type_Macroblock04;
    i_data_leftcoll_botomrow_0 : in  type_Macroblock04;
    i_data_leftcoll_botomrow_1 : in  type_Macroblock04;
    i                          : in  integer range 0 to 15;--(0=lumma, 1-15=chroma)
    --Parameter Inputs
    INTERNAL_CLR_FMT           : in  std_logic_vector(2 downto 0);
    --Data Outputs
    o_data_interior_0          : out type_Macroblock16;
    o_data_toprowww_rightcol_0 : out type_Macroblock04;
    o_data_toprowww_rightcol_1 : out type_Macroblock04;
    o_data_leftcoll_botomrow_0 : out type_Macroblock04;
    o_data_leftcoll_botomrow_1 : out type_Macroblock04
    );
end entity;

architecture rtl of InverseOverlapFilter is

  signal data_out_sig_reg00 : type_arr16_slvMSBbit;
  signal data_out_sig_reg04 : type_arr16_slvMSBbit;
  signal data_out_sig_reg12 : type_arr16_slvMSBbit;

  signal do_4x4i0_sig : type_Macroblock16;
  signal do_4OFt0_sig : type_Macroblock04;
  signal do_4OFt1_sig : type_Macroblock04;
  signal do_4OFb0_sig : type_Macroblock04;
  signal do_4OFb1_sig : type_Macroblock04;

  signal sync_r1   : std_logic;
  signal sync_r2   : std_logic;
  signal sync_r3   : std_logic;
  signal sync_r4   : std_logic;
  signal sync_r5   : std_logic;
  signal sync_r6   : std_logic;
  signal sync_r7   : std_logic;
  signal sync_r8   : std_logic;
  signal sync_r9   : std_logic;
  signal sync_r10  : std_logic;
  signal sync_r11  : std_logic;
  signal sync_r12  : std_logic;
  signal sync_r13  : std_logic;
  signal sync_r14  : std_logic;
  signal sync_r15  : std_logic;
  signal sync_r16  : std_logic;
  signal sync_r17  : std_logic;
  signal sync_r18  : std_logic;
  signal sync_r19  : std_logic;
  signal sync_r20  : std_logic;
  signal sync_r21  : std_logic;
  signal sync_r22  : std_logic;
  signal sync_r23  : std_logic;
```

```
  signal sync_r24  : std_logic;
  signal sync_r25  : std_logic;
  signal sync_r26  : std_logic;
  signal sync_r27  : std_logic;
  signal sync_r28  : std_logic;
  signal sync_r29  : std_logic;
  signal sync_r30  : std_logic;


begin
  data_out_sig_reg00(15) <= do_4OFt0_sig(3);
  data_out_sig_reg00(14) <= do_4OFt0_sig(2);
  data_out_sig_reg00(13) <= do_4OFt0_sig(1);
  data_out_sig_reg00(12) <= do_4OFt0_sig(0);
  data_out_sig_reg00(11) <= do_4OFt1_sig(3);
  data_out_sig_reg00(10) <= do_4OFt1_sig(2);
  data_out_sig_reg00( 9) <= do_4OFt1_sig(1);
  data_out_sig_reg00( 8) <= do_4OFt1_sig(0);
  data_out_sig_reg00( 7) <= do_4OFb0_sig(3);
  data_out_sig_reg00( 6) <= do_4OFb0_sig(2);
  data_out_sig_reg00( 5) <= do_4OFb0_sig(1);
  data_out_sig_reg00( 4) <= do_4OFb0_sig(0);
  data_out_sig_reg00( 3) <= do_4OFb1_sig(3);
  data_out_sig_reg00( 2) <= do_4OFb1_sig(2);
  data_out_sig_reg00( 1) <= do_4OFb1_sig(1);
  data_out_sig_reg00( 0) <= do_4OFb1_sig(0);

o_data_interior_0 <= do_4x4i0_sig;
o_data_toprowww_rightcol_0 <= type_Macroblock04(data_out_sig_reg12(15 downto 12));
o_data_toprowww_rightcol_1 <= type_Macroblock04(data_out_sig_reg12(11 downto  8));
o_data_leftcoll_botomrow_0 <= type_Macroblock04(data_out_sig_reg12( 7 downto  4));
o_data_leftcoll_botomrow_1 <= type_Macroblock04(data_out_sig_reg12( 3 downto  0));

data_valid <= sync_r30;
data_valid_m1 <= sync_r29;

--This process registers the sync pulse for the max number of cycles
-- for this module to complete.
process(clk,reset)
begin
  if reset = '1' then
    sync_r1  <= '0';
    sync_r2  <= '0';
    sync_r3  <= '0';
    sync_r4  <= '0';
    sync_r5  <= '0';
    sync_r6  <= '0';
    sync_r7  <= '0';
    sync_r8  <= '0';
    sync_r9  <= '0';
    sync_r10 <= '0';
    sync_r11 <= '0';
    sync_r12 <= '0';
```

```
    sync_r13 <= '0';
    sync_r14 <= '0';
    sync_r15 <= '0';
    sync_r16 <= '0';
    sync_r17 <= '0';
    sync_r18 <= '0';
    sync_r19 <= '0';
    sync_r20 <= '0';
    sync_r21 <= '0';
    sync_r22 <= '0';
    sync_r23 <= '0';
    sync_r24 <= '0';
    sync_r25 <= '0';
    sync_r26 <= '0';
    sync_r27 <= '0';
    sync_r28 <= '0';
    sync_r29 <= '0';
    sync_r30 <= '0';
  elsif rising_edge(clk) then
    sync_r1  <= sync;
    sync_r2  <= sync_r1;
    sync_r3  <= sync_r2;
    sync_r4  <= sync_r3;
    sync_r5  <= sync_r4;
    sync_r6  <= sync_r5;
    sync_r7  <= sync_r6;
    sync_r8  <= sync_r7;
    sync_r9  <= sync_r8;
    sync_r10 <= sync_r9;
    sync_r11 <= sync_r10;
    sync_r12 <= sync_r11;
    sync_r13 <= sync_r12;
    sync_r14 <= sync_r13;
    sync_r15 <= sync_r14;
    sync_r16 <= sync_r15;
    sync_r17 <= sync_r16;
    sync_r18 <= sync_r17;
    sync_r19 <= sync_r18;
    sync_r20 <= sync_r19;
    sync_r21 <= sync_r20;
    sync_r22 <= sync_r21;
    sync_r23 <= sync_r22;
    sync_r24 <= sync_r23;
    sync_r25 <= sync_r24;
    sync_r26 <= sync_r25;
    sync_r27 <= sync_r26;
    sync_r28 <= sync_r27;
    sync_r29 <= sync_r28;
    sync_r30 <= sync_r29;
  end if;
end process;

--interior YUV444 only
```

```
u_4x4OverlapFilter : entity work.funct_4x4OverlapFilter(rtl) port map(
  clk   => clk  ,
  reset => reset,
  a_in  => i_data_interior_0(0),
  b_in  => i_data_interior_0(1),
  c_in  => i_data_interior_0(2),
  d_in  => i_data_interior_0(3),
  e_in  => i_data_interior_0(4),
  f_in  => i_data_interior_0(5),
  g_in  => i_data_interior_0(6),
  h_in  => i_data_interior_0(7),
  i_in  => i_data_interior_0(8),
  j_in  => i_data_interior_0(9),
  k_in  => i_data_interior_0(10),
  l_in  => i_data_interior_0(11),
  m_in  => i_data_interior_0(12),
  n_in  => i_data_interior_0(13),
  o_in  => i_data_interior_0(14),
  p_in  => i_data_interior_0(15),
  a_out => do_4x4i0_sig(0),
  b_out => do_4x4i0_sig(1),
  c_out => do_4x4i0_sig(2),
  d_out => do_4x4i0_sig(3),
  e_out => do_4x4i0_sig(4),
  f_out => do_4x4i0_sig(5),
  g_out => do_4x4i0_sig(6),
  h_out => do_4x4i0_sig(7),
  i_out => do_4x4i0_sig(8),
  j_out => do_4x4i0_sig(9),
  k_out => do_4x4i0_sig(10),
  l_out => do_4x4i0_sig(11),
  m_out => do_4x4i0_sig(12),
  n_out => do_4x4i0_sig(13),
  o_out => do_4x4i0_sig(14),
  p_out => do_4x4i0_sig(15)
  );
--YUV444 top row / right col 0
ut0_4OverlapFilter : entity work.funct_4OverlapFilter(rtl) port map(
  clk   => clk  ,
  reset => reset,
  a_in  => i_data_toprowww_rightcol_0(0),
  b_in  => i_data_toprowww_rightcol_0(1),
  c_in  => i_data_toprowww_rightcol_0(2),
  d_in  => i_data_toprowww_rightcol_0(3),
  a_out => do_4OFt0_sig(0),
  b_out => do_4OFt0_sig(1),
  c_out => do_4OFt0_sig(2),
  d_out => do_4OFt0_sig(3)
  );
--YUV444 top row / right col 1
ut1_4OverlapFilter : entity work.funct_4OverlapFilter(rtl) port map(
  clk   => clk  ,
  reset => reset,
```

```
  a_in  => i_data_toprowww_rightcol_1(0),
  b_in  => i_data_toprowww_rightcol_1(1),
  c_in  => i_data_toprowww_rightcol_1(2),
  d_in  => i_data_toprowww_rightcol_1(3),
  a_out => do_4OFt1_sig(0),
  b_out => do_4OFt1_sig(1),
  c_out => do_4OFt1_sig(2),
  d_out => do_4OFt1_sig(3)
  );
--YUV444 bottom row / left col 0
ub0_4OverlapFilter : entity work.funct_4OverlapFilter(rtl) port map(
  clk   => clk  ,
  reset => reset,
  a_in  => i_data_leftcoll_botomrow_0(0),
  b_in  => i_data_leftcoll_botomrow_0(1),
  c_in  => i_data_leftcoll_botomrow_0(2),
  d_in  => i_data_leftcoll_botomrow_0(3),
  a_out => do_4OFb0_sig(0),
  b_out => do_4OFb0_sig(1),
  c_out => do_4OFb0_sig(2),
  d_out => do_4OFb0_sig(3)
  );
--YUV444 bottom row / left col 1
ub1_4OverlapFilter : entity work.funct_4OverlapFilter(rtl) port map(
  clk   => clk  ,
  reset => reset,
  a_in  => i_data_leftcoll_botomrow_1(0),
  b_in  => i_data_leftcoll_botomrow_1(1),
  c_in  => i_data_leftcoll_botomrow_1(2),
  d_in  => i_data_leftcoll_botomrow_1(3),
  a_out => do_4OFb1_sig(0),
  b_out => do_4OFb1_sig(1),
  c_out => do_4OFb1_sig(2),
  d_out => do_4OFb1_sig(3)
  );

out_reg4_gen : for index in 0 to 15 generate
  ur1_0_reg_data : entity work.register_data_chain(rtl)
    generic map(num_registers => 4)
    port map(
      clk     => clk,
      reset   => reset,
      i_pulse => data_out_sig_reg00(index),
      o_pulse => data_out_sig_reg04(index)
      );
end generate;

out_reg8_gen : for index in 0 to 15 generate
  ur1_0_reg_data : entity work.register_data_chain(rtl)
    generic map(num_registers => 8)
    port map(
      clk     => clk,
      reset   => reset,
```

```
      i_pulse => data_out_sig_reg04(index),
      o_pulse => data_out_sig_reg12(index)
      );
end generate;

end architecture;
```

Appendix B

QuartusII Reports

ILBT Fitting

```
Fitter Status :  Successful - Sat Jul 25 13:03:40 2009
Quartus II Version :  9.0 Build 132 02/25/2009 SJ Full Version
Revision Name :  InverseLBT
Top-level Entity Name :  InverseLBT
Family :  Stratix II
Device :  EP2S180F1508C3
Timing Models :  Final
Logic utilization :  13 %
   Combinational ALUTs :  12,792 / 143,520 ( 9 % )
   Dedicated logic registers :  17,653 / 143,520 ( 12 % )
Total registers :  17950
Total pins :  1,161 / 1,171 ( 99 % )
Total virtual pins :  0
Total block memory bits :  141,258 / 9,383,040 ( 2 % )
DSP block 9-bit elements :  0 / 768 ( 0 % )
Total PLLs :  0 / 12 ( 0 % )
Total DLLs :  0 / 2 ( 0 % )
```

ILBT Timing

```
TimeQuest Timing Analyzer Summary
Type :  Setup 'clk'
```

```
Slack :  0.666
TNS : 0.0
Type :  Hold 'clk'
Slack :  0.212
TNS : 0.0
Fmax :  116.47 (MHz)
```

ILBT Power

```
PowerPlay Power Analyzer Status :  Successful - Thu Jul 23 08:29:09
2009
Quartus II Version :  9.0 Build 132 02/25/2009 SJ Full Version
Revision Name :  InverseLBT
Top-level Entity Name :  InverseLBT
Family :  Stratix II
Device :  EP2S180F1508C3
Power Models :  Final
Total Thermal Power Dissipation :  1445.25 mW
Core Dynamic Thermal Power Dissipation :  0.00 mW
Core Static Thermal Power Dissipation :  1366.96 mW
I/O Thermal Power Dissipation :  78.29 mW
Power Estimation Confidence :  Low:  user provided insufficient toggle
rate data
```

IPCT Fitting

```
Fitter Status :  Successful - Mon Jul 20 21:16:08 2009
Quartus II Version :  9.0 Build 132 02/25/2009 SJ Full Version
Revision Name :  hd_photo
Top-level Entity Name :  InversePhotoCoreTransform
Family :  Stratix II
Device :  EP2S180F1508C3
Timing Models :  Final
Logic utilization :  2 %
   Combinational ALUTs :  1,883 / 143,520 ( 1 % )
```

```
   Dedicated logic registers :  2,364 / 143,520 ( 2 % )
Total registers :  2364
Total pins :  524 / 1,171 ( 45 % )
Total virtual pins :  0
Total block memory bits :  2,410 / 9,383,040 ( < 1 % )
DSP block 9-bit elements :  0 / 768 ( 0 % )
Total PLLs :  0 / 12 ( 0 % )
Total DLLs :  0 / 2 ( 0 % )
```

IPCT Timing

```
TimeQuest Timing Analyzer Summary
Type :  Setup 'clk'
Slack :  -2.844
TNS : 0.0
Type :  Hold 'clk'
Slack :  .357
TNS : 0.0
Fmax :  361.79 (MHz)
```

IPCT Power

```
PowerPlay Power Analyzer Status :  Successful - Thu Jul 23 08:39:12
2009
Quartus II Version :  9.0 Build 132 02/25/2009 SJ Full Version
Revision Name :  InverseLBT
Top-level Entity Name :  InversePhotoCoreTransform
Family :  Stratix II
Device :  EP2S180F1508C3
Power Models :  Final
Total Thermal Power Dissipation :  1421.54 mW
Core Dynamic Thermal Power Dissipation :  0.00 mW
Core Static Thermal Power Dissipation :  1366.55 mW
I/O Thermal Power Dissipation :  55.00 mW
Power Estimation Confidence :  Low:  user provided insufficient toggle
```

rate data

IPOF Fitting

```
Fitter Status :  Successful - Mon Jul 20 20:45:54 2009
Quartus II Version :  9.0 Build 132 02/25/2009 SJ Full Version
Revision Name :  hd_photo
Top-level Entity Name :  InverseOverlapFilter
Family :  Stratix II
Device :  EP2S180F1508C3
Timing Models :  Final
Logic utilization :  4 %
   Combinational ALUTs :  4,017 / 143,520 ( 3 % )
   Dedicated logic registers :  5,902 / 143,520 ( 4 % )
Total registers :  5902
Total pins :  1,036 / 1,171 ( 88 % )
Total virtual pins :  0
Total block memory bits :  8,563 / 9,383,040 ( < 1 % )
DSP block 9-bit elements :  0 / 768 ( 0 % )
Total PLLs :  0 / 12 ( 0 % )
Total DLLs :  0 / 2 ( 0 % )
```

IPOF Timing

```
TimeQuest Timing Analyzer Summary
Type :  Setup 'clk'
Slack :  0.373
TNS : 0.0
Type :  Hold 'clk'
Slack :  .212
TNS : 0.0
Fmax :  311.53 (MHz)
```

IPOF Power

PowerPlay Power Analyzer Status : Successful - Thu Jul 23 08:46:26 2009
Quartus II Version : 9.0 Build 132 02/25/2009 SJ Full Version
Revision Name : InverseLBT
Top-level Entity Name : InverseOverlapFilter
Family : Stratix II
Device : EP2S180F1508C3
Power Models : Final
Total Thermal Power Dissipation : 1440.48 mW
Core Dynamic Thermal Power Dissipation : 0.00 mW
Core Static Thermal Power Dissipation : 1366.87 mW
I/O Thermal Power Dissipation : 73.61 mW
Power Estimation Confidence : Low: user provided insufficient toggle rate data