The Report Committee for Feng Zheng
Certifies that this is the approved version of the following report:

Hardware Accelerator for the JPEG Encoder

On the Xilinx SPARTAN 3 FPGA

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Jacob A. Abraham

Jason C. Perkey

Hardware Accelerator for the JPEG Encoder

On the Xilinx SPARTAN 3 FPGA

by

Feng Zheng, B.S.E.E.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

The University of Texas at Austin

December, 2010

Dedication

To Franny.

Acknowledgments

The author would like to acknowledge the significant guidance and support from Dr. Jacob Abraham, Mark McDermott as well as the classmates of the Cohort 5 MS-ECD program.

December 2010

Abstract

Hardware Accelerator for the JPEG Encoder

On the Xilinx SPARTAN 3 FPGA

Feng Zheng, M.S.E.

The University of Texas at Austin, 2010

Supervisor: Jacob A. Abraham

The report detailing the Hardware Accelerator for the JPEG encoder is organized into three sections. First, it will review the processes of the Joint Photographic Experts Group (JPEG) encoding and decoding standard. Second, it will review three different implementations of the discrete cosine transform in hardware. This is a very computationally intensive element of the JPEG encoding process and the analysis of these designs covers the benefits and costs of the various approaches for the Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC) implementations. Finally, it will discuss this specific hardware accelerator design for a color state transformation for the standard JPEG encoder. An eight by eight matrix of Red, Green, Blue (RGB) values is passed into the FPGA as well as calculated in

software. The Y Cr Cb results from that of the hardware accelerator implementation are compared with the software implementation for computational accuracy and the differences in computation time are sampled for a comparison. There is a clear 38% improvement in speed from the hardware accelerator.

Table of Contents

List of Figures..... x

PART 1: BACKGROUND..... 1

Chapter 1: Introduction..... 1

Chapter 2: Basic JPEG Encoding Standard..... 3

2.1 Color Space Conversion and Downsampling..... 3

2.2 Block Splitting and Discrete Cosine Transform 4

2.3 Quantization and Entropy Coding..... 5

PART 2: Hardware Accelerators..... 9

Chapter 3: DCT Hardware Designs..... 9

3.1 Hardware Implementation of the Encoder..... 10

3.2 Hardware implementation of the 2-D DCT..... 13

3.3 Seven stage DCT Implementation..... 17

3.4 Implementation Result..... 19

Chapter 4: RGB to Y Cr Cb Conversion..... 21

4.1 Hardware And Software Computations..... 21

4.2 Software Hardware Communication.....22

4.3 Interrupt Handler.....23

Chapter 5: Testing and Results..... 25

Appendices..... 28

Appendix A: Acronym Definitions..... 28

Appendix B: Hardware Accelerator Source Code..... 29

References..... 45

Vita..... 46

List of Figures

Figure 1. JPEG Scalar Quantization Table [4]..... 6

Figure 2. Zig-Zag Ordering [4]..... 7

Figure 3. Kovac's Jaguar Architecture for the hardware JPEG encoder [5].. 11

Figure 4. Entropy encoding logic [5]……..12

Figure 5. Six step 1-D DCT architecture [6]..... 14

Figure 6. Multiplier architecture in the DCT pipeline [6]..... 15

Figure 7. Transpose memory buffer architecture [6]..... 16

Figure 8. 2D-DCT with single seven step pipelined DCT [8]..... 17

Figure 9. Seven step 1D-DCT [8]..... 18

Figure 10. State diagram of *top.v*..... 24

PART 1: BACKGROUND

Chapter 1: Introduction

Data compression is the reduction and elimination of redundancy in data to gain savings in storage and communication costs. In the last two decades, the JPEG codec for image compression has become a popular standard used on the Internet and in many mobile devices. Many sites on the World Wide Web store images in the JPEG format, and JPEG is the primary image format for digital cameras. The popularity is driven by the fact that this codec standard allows for faster writes to memory and is compatible with email and internet pages, allowing for maximization of the number of images which can be stored.

The aim of JPEG standard committee was to propose an image compression algorithm that would be application independent and aid VLSI implementation of data compression [1]. The compression can be lossy or loss-less depending on the application demand. From lossy compression methods, close approximation of the original data are obtained. Loss-less compression methods are typically used in text compression and image compression in environments where exact original data are required for applications such as medical imaging. Computation in hardware can accelerate performance of the compression process which may be required for real-time devices. Many hardware implementations are available for Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Arrays (FPGA) design.

JPEG compression can be divided into five main steps: color space conversion, down sampling, 2-D discrete cosine transform (DCT), quantization and entropy coding. The color space conversion and down sampling steps are only used for color images. The

color space conversion transforms the RGB input image to a luminance and chrominance space color, such as the Y Cb Cr representation. The down sampling operation reduces the sampling ratio of the Cb and Cr color information with little or no noticeable difference for the human eye [2].

Chapter 2, presents a review of the encoding and decoding process for the JPEG standard. Chapter 3 is a literary analysis of three designs for hardware implementation of JPEG compression. This paper will explore the implementation's tradeoff between performance and area enhancement. Chapter 4, describes the author’s implementation of the color space conversion process on a hardware accelerator. Finally, Chapter 5 presents the conclusions of the experiment for the hardware accelerator and thoughts on future work.

Chapter 2: Basic JPEG Encoding Standard

JPEG is a digital-image codec standard that is specified by the Joint Photographic Experts Group, a joint International Organization for Standardization (ISO) and International Telegraph and Telephone Consultative Committee (CCITT) technical committee [3]. JPEG standard was originally developed for use in areas such as desktop publishing, graphic arts, medical imaging, and document imaging, where it has become the common format for storing and transmitting still images [3]. However, the introduction of high performance hardware capable of encoding and decoding JPEG images in real time has enabled the development of full motion video application based on JPEG and made this standard more common for mobile devices and cameras.

2.1 Color Space Conversion and Downsampling

During the encoding process the original image first is converted from RGB (red, green, blue) to Y Cb Cr. Y is the luminous component representing brightness of a pixel while Cb and Cr are both blue and red components representing chrominance. This is one of the most computationally intensive parts of the encoding process. The data is converted from RGB because the Y Cb Cr components allows for greater compression. The brightness is separated from the less visually significant chrominance components [4]. However, when the size of the stored data is not of great significance, the RGB color model is kept and this color space conversion is absent from the encoding process. The resolution of chrominance data can be reduced by a factor of 2 because the human eye is

less sensitive to color details compared to the detail in brightness [1]. This reduction is referred to as down-sampling or Chroma subsampling.

2.2 Block Splitting and Discrete Cosine Transform

The input to the JPEG encoder is split into 8 by 8 pixel blocks of data after subsampling. If the data does not conclude with an integer number of blocks, the encoder will fill the remaining area of the incomplete blocks with dummy data. One disadvantage associated with this method, however, is that it may create ringing artifacts along the border of the image [2].

The compressor encodes each block of data in three steps. First a Discrete Cosine Transform (DCT) is performed on the 8 by 8 block of data. Then, the results from the DCT will be quantized. Finally the encoder entropy code further compresses the data with a loss-less algorithm [3]. The discrete cosine transform is a mathematical operation that takes the 8 by 8 blocks of data as its input and coverts the information from the spatial domain to the frequency domain using a normalized, two dimensional type II discrete cosine transform. Before computing the DCT, the gray scale values are shifted from the positive range to one where the average is around zero. This can be done by subtracting all values by half the number of possible values, or 128. By subtracting each value by 128 the possible range for each value is between -128 to 127. The formula for 2-D DCT is:

$$G_{u,v} = \alpha(u)\alpha(v) \sum_{x=0}^{7} \sum_{y=0}^{7} g_{x,y} \cos\left[\frac{\pi}{8}\left(x + \frac{1}{2}\right) u\right] \cos\left[\frac{\pi}{8}\left(y + \frac{1}{2}\right) v\right]$$

In this equation, the *u* and *v* are horizontal and vertical spatial frequencies, respectively. The value for α*(n)* is sqrt(1/8) if *n* is 0 and sqrt(2/8) otherwise. *G(u,v)* is the DCT coefficient at coordinates (*u,v*). During the direct cosine transform stage the encoder is loss-less. Input values which represent brightness levels are transformed into values which represent spatial frequencies that make up the input data spectrum. Low spatial frequencies far outweigh high spatial frequencies in terms of importance in each block of input. This transformation provides a good opportunity for data compression. In the output matrix, the lowest frequencies are stored in the upper-left corner, and information about the highest frequencies is stored in the lower right corner [4]. Much of the high frequency values can be sacrificed without much noticeable changes to the image in the next step of encoding. This is because human vision is more sensitive to small variations in color or brightness over large areas than the strength of high frequency brightness variations.

2.3 Quantization and Entropy Coding

The quantization step involves dividing each value in the matrix output from the DCT by a corresponding value in the quantization table then rounding to the nearest integer. Figure 1 is a typical quantization matrix, as specified by the original JPEG standard:

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Figure 1. JPEG Scalar Quantization Table [4]

The most important results of the DCT are in the upper right hand corner of each value block. The quantizer is the part of the JPEG baseline sequential encoder that causes the encoder to be lossy. The degree of compression can be adjusted allowing a tradeoff between storage size and image quality. JPEG typically attains a ten to one compression with minimal distinguishable loss in image quality.

The entropy coding block creates the actual JPEG bit stream with the quantized output. The values in the quantized block of data are read in a zigzag sequence to ensure that the encoder will encounter all nonzero values in the block as early as possible. Figure 2 below shows the order for the zigzag sequence.

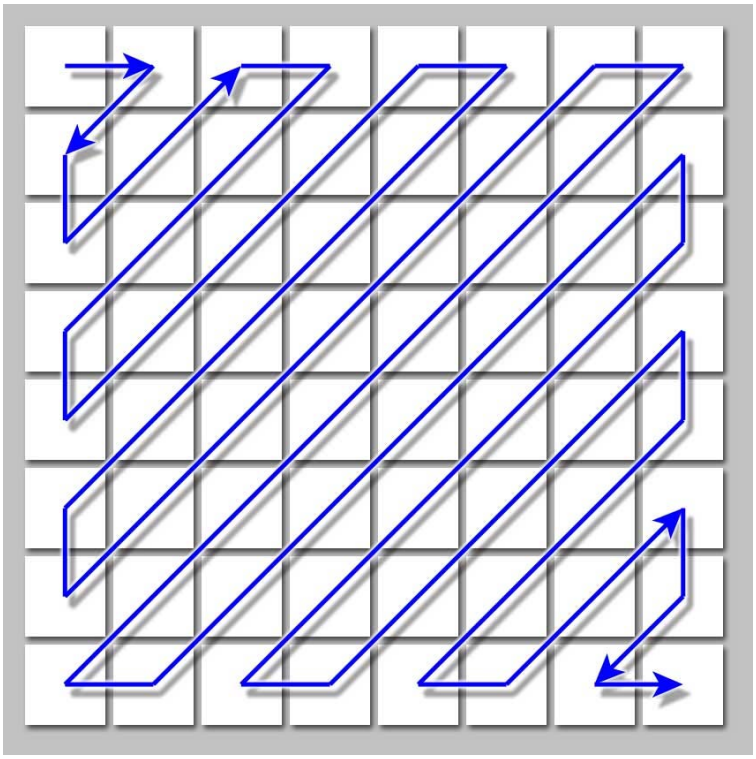

Figure 2. Zig-Zag Ordering [4]

Three pieces of information are collected each time the encoder encounters a nonzero value as it moves through the data matrix. These are the number of zeros it passed over before finding the nonzero value, the number of bits it will take to encode the nonzero value, and the value itself. The encoder then consults a Huffman table to find the bit sequence that represents the first two pieces of information, writes this bit sequence to the JPEG bit stream, then encodes the value using a variable-length code. When the remaining values in the block are all zeros, the encoder writes a special end-of-block bit sequence to the bit stream The JPEG standard also allows the use of arithmetic coding as an alternative to Huffman coding. A JPEG extension replaces the Huffman engine with a binary arithmetic entropy encoder which results in the size of the compressed JPEG data

being 10 to 15 percent smaller than the result achieved by the Huffman coder. However, it is also slower to encode and to decode and protected by patents [11]. The arithmetic coding algorithm is owned by IBM and AT&T and a license must be obtained for the coder to be used. The decoding process to display the image is simply a reverse of all of the encoding steps. First, the differences of the DC coefficients are added back in. Second, an inverse DCT step is carried out. Third, the products of the second step are rounded to whole integer values. Finally, the number 128 is added back to all of the values [7].

PART 2: Hardware Accelerators

Chapter 3: DCT Hardware Designs

Because software implementations are incompatible or too inefficient to compress and/or decompress JPEG images in real time devices, hardware specific implementations of JPEG must be used in devices such as digital cameras or platforms such as Field Programmable Gate Arrays [5]. Various tradeoffs for performance, power, and area exist between different types of hardware implementations of the JPEG decoder/encoder. For example, dedicated digital hardware designs can provide improved speed but take more research and development time which can hurt time to market. Modern FPGAs can handle parallel pipeline processing required in real time signal processing by providing speed, performance, and flexibility but lacks the area efficiency of the dedicated digital hardware. The DCT process is computationally intensive, making it an excellent candidate for optimization with a hardware accelerator design.

Three methods for hardware implementation of the JPEG baseline compression will be reviewed below. First, Kovac presents the architecture implementation of the discrete cosine transform and entropy encoder for a single high speed VLSI chip [5]. The architecture exploits the principles of pipelining and parallelism to achieve high speed and throughput. The design yields a clock rate of 100MHz at input rate of 30 frames per second for 1024x1024 color images [5]. In the second approach, Agostini's paper discusses the architecture and the VHDL synthesis of a 2-D DCT on the Altera Flex10kE

FPGA. The architecture is divided into the familiar two 1-D DCT calculations by using the transpose buffer. The design uses 4,792 logic cells of the FPGA and reached an operating frequency of 12.2 MHz. For one input block of 8 by 8 elements, the block was processed in 25.2us with a pipelined latency of 160 clock cycles [6]. Lastly, the approach from Tumeo propose the use a seven stage 1D-DCT for the same computation. Using the embedded hardware of the FPGA, Tumeo shows a design which is more advanced in not only area but also performance of the encoder [8]. By exploiting symmetries of the algorithm to save area, the seven stage 1D-DCT accelerator alternates computation for the even and odd coefficients in every cycle.

3.1 Hardware Implementation of the Encoder

Kovac organized the entire architecture as a linear stage pipeline in order to achieve high throughput. The image is input at the rate of one pixel per clock cycle into the architecture [5]. The compressed data is output by the system at a variable rate depending on the amount of compression achieved. In the DCT module, the two dimensional DCT computation is separated into a row wise one dimensional DCT operation and a second column wise DCT operation. These one dimensional DCT computations are separated by a transpose buffer.

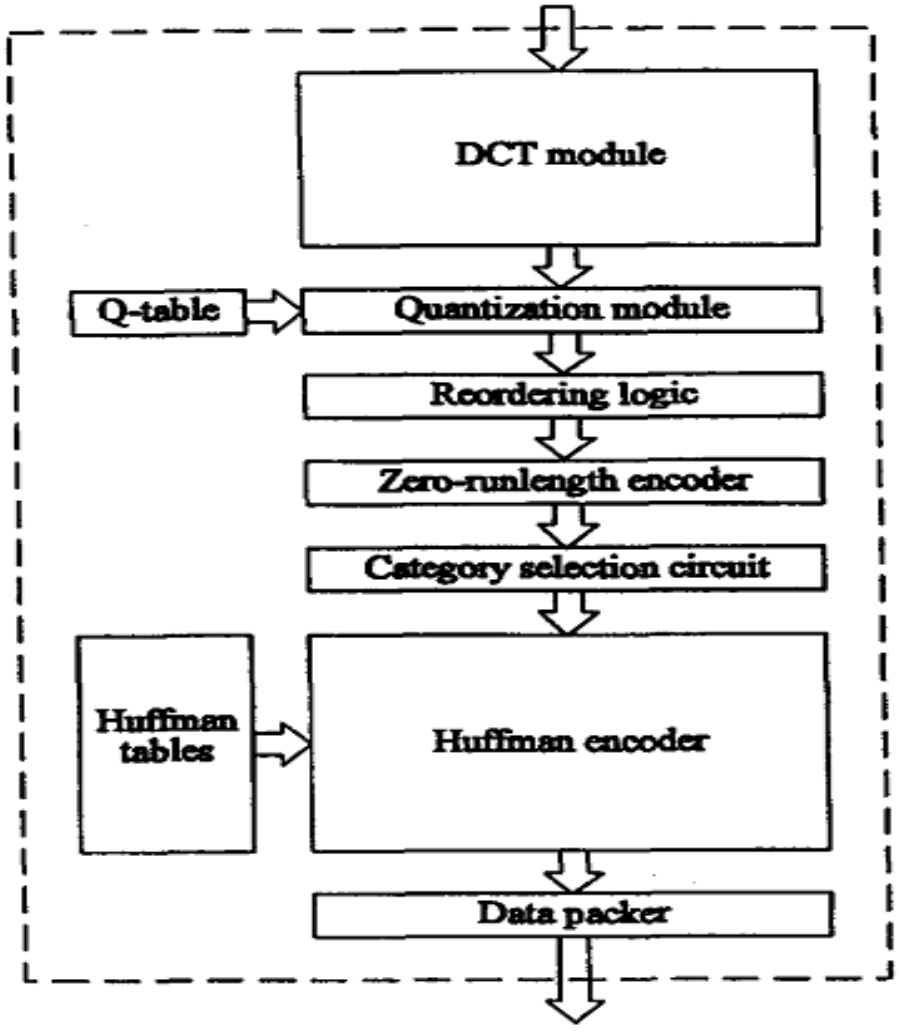

Figure 3. Kovac's Jaguar Architecture for the hardware JPEG encoder [5].

The single 1D-DCT consists of six partitions and accepts one pixel per clock cycle. The adders are single stage units and the multiplier is a six stage Wallace tree multiplier. The algorithm for the 1-D DCT requires 5 multiplications, 29 additions and 16 two's complement additions. The circuit has a latency of 59 clock cycles for each 1-D DCT computation. The transpose buffer, with a latency of 64 clock cycles, consists of an 8 by 8 array of register pairs. Data from the row registers receive values from the DCT

module until all 64 registers are loaded then the registers are copied in parallel into corresponding adjacent registers connected in column-wise fashion. Thus, the output of the row-wise DCT is transposed to the column-wise computation.

The quantization module consists of a RAM to store the quantization table [5]. The output of the DCT is multiplied by a set of predefine values from the quantization table. The latency is six clock cycles. The results of the quantization module are reordered in a zigzag fashion with an 8 by 8 array or registered pairs.

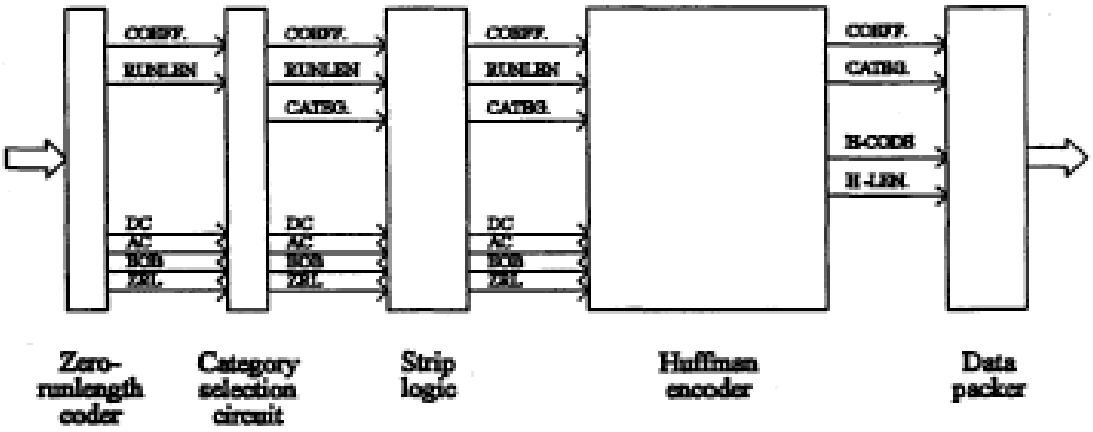

Figure 4. Entropy encoding logic [5].

The entropy encoder stages which follows consists of a zero-run length coder, a category selection circuit, strip logic, the Hoffman encoder and data packer. The zero-run length coder calculates the delta DC which is the difference between the current DC coefficient and the DC coefficient of the previous block. If the sign of the coefficient is negative the DC/AC coefficients are decremented by one. The model has three stages and a latency of three cycles. Each DC and AC coefficient is associated with a corresponding

category depending on the magnitude of the coefficient, within the category selection circuit [10].

Kovac uses a simple combinatorial circuit to convert a given coefficient into the corresponding category in a single clock cycle. The data stream still contains all 64 coefficients. The strip logic strip off the zero valued coefficients and ZRL symbols that precede an EOB symbol [10]. Strip logic consists of four stages with three registers in each stage. This four stage buffer compressed data elements after the removal of zero coefficients.

The Huffman encoder consists of Huffman code tables stored in random access memory modules and logic for replacing the run length count pairs with corresponding Hoffman codes. Finally, the data pacer converts variable length compressed data into fixed length compressed data stream.

3.2 Hardware implementation of the 2-D DCT

Agostini uses the generic 2-D DCT architecture where the algorithm uses two 1-D DCT steps to generate the 2-DCT coefficients [6]. In an 8 by 8 matrix, the first 1-D DCT is performed row-wise and the second 1-D DCT is performed column-wise on the outputs of the first 1-D DCT. The design can reach a high operating frequency and allow the use of pipeline techniques.

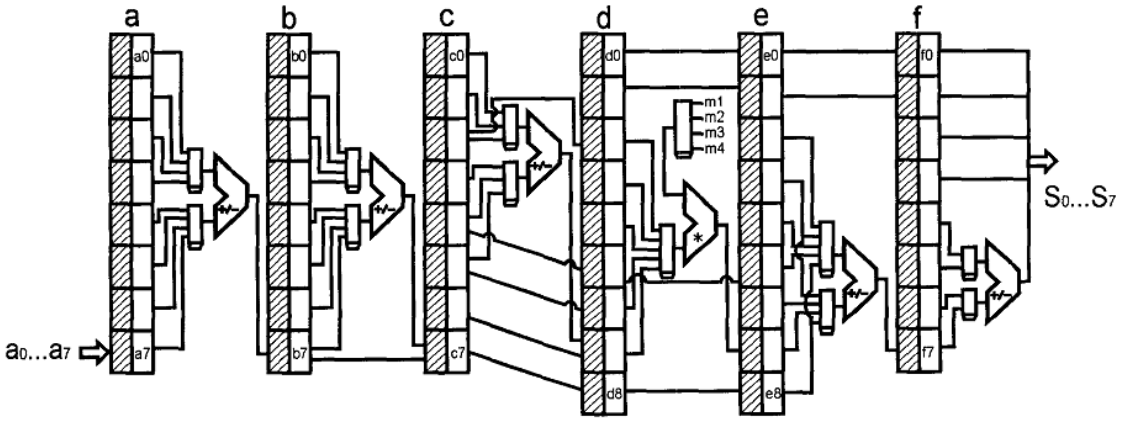

Figure 5. Six step 1-D DCT architecture [6].

Agostini uses the design for 1-D DCT presented in Kovac's paper. The 1-D DCT pipelined has six stages for the six steps of the algorithm. The multiplier architecture diverges from the six stages Wallace tree used by Kovac and instead use shifts and adds to minimize the hardware. The assumption made is that there are not existing multiplier available in the FPGA. The design uses 6 clock cycles saving 8 clock cycles from the Wallace tree 1D-DCT. The number of shifts-adds was restricted to four which saves arithmetic units but generates a 0.6% error in the constant values. All significant bits are considered internally to the multiplier to maximize precision of the calculation however, the outputs are truncated by discarding the fractional components. This loss in procession is viewed as having little significance because the next operation in compression involve the integer division by numbers higher than 10.

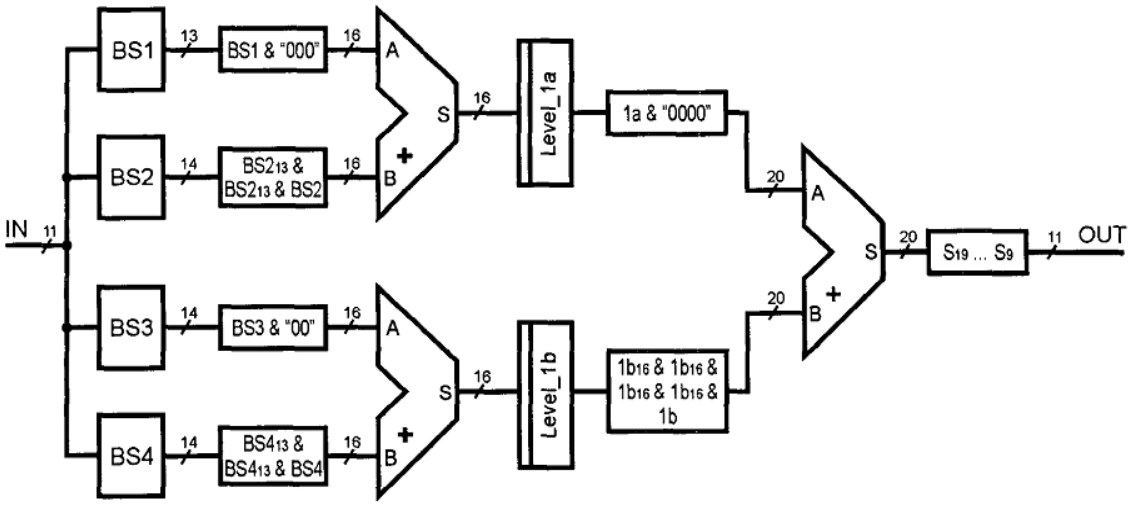

Figure 6. Multiplier architecture in the DCT pipeline [6].

Internal FPGA RAM are used by Agostini instead of conventional RAM cells by Kovac. FPGA has internal RAM macro-blocks which can be used to save logic cells. Registers are also available in FPGAs, however registers consume large amounts of logic cells and their performance is no better than the internal FPGA RAM's performance. The transpose buffer VHDL description used is specific to the Altera device.

The 2-D DCT was synthesized into an Altera Flex 10KE family FPGA. The 2-DCT hardware was designed and described in approximately 5,250 lines of VHDL code. The two 1D-DCT architectures descriptions are structural and device independent so it can be described in a variable number of FPGA however the transpose buffer uses an Altera specific library that allows the use of internal memory.

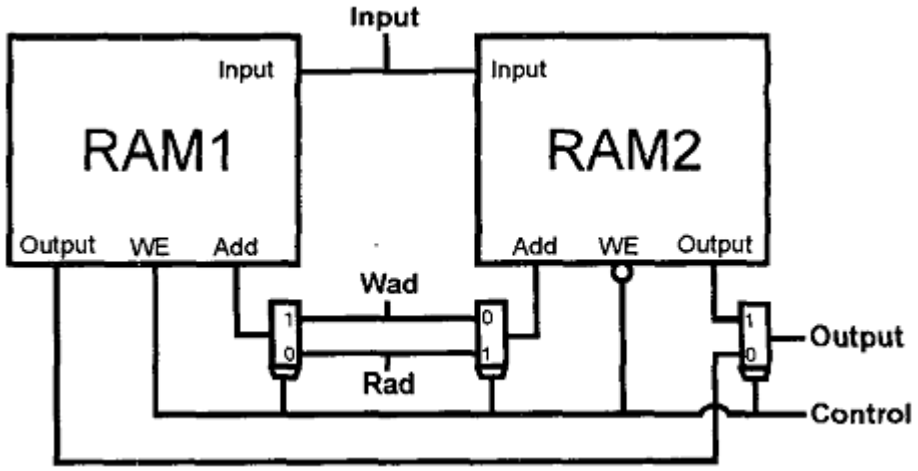

Figure 7. Transpose memory buffer architecture [6].

This makes the description language Altera dependent although the code can be modified for other FPGA families [6]. The complete synthesis results for the Altera FPGA is the use of 4792 logic cells, and 1536 memory bits for the transpose buffer. Each 8 by 8 input block is processed in 5.6us when the pipeline is full. An empty pipeline can process a 640 x 480 pixel gray level image in just 25.2ms with a processing image rate of 39 images per second. A color image in the same setting can be processed in 75.7ms with a processing image rate of 13 images per second [6]. The results suggest that the entire JPEG compressor can be implemented in hardware including the I/O and bus control functions.

3.3 Seven stage DCT Implementation

The goal of the Tumeo design is to create a pipelined, fast 2D-DCT accelerator for the FPGA which focus on both performance and area optimization [8]. Both this design as well as Agostini's design focus only on the 2D-DCT phase of the encoding process. This work assume the implementation will work as a component of a complete HW/SW implementation of the JPEG encoding algorithm while the previous two papers assume a full hardware implementation solution [8].

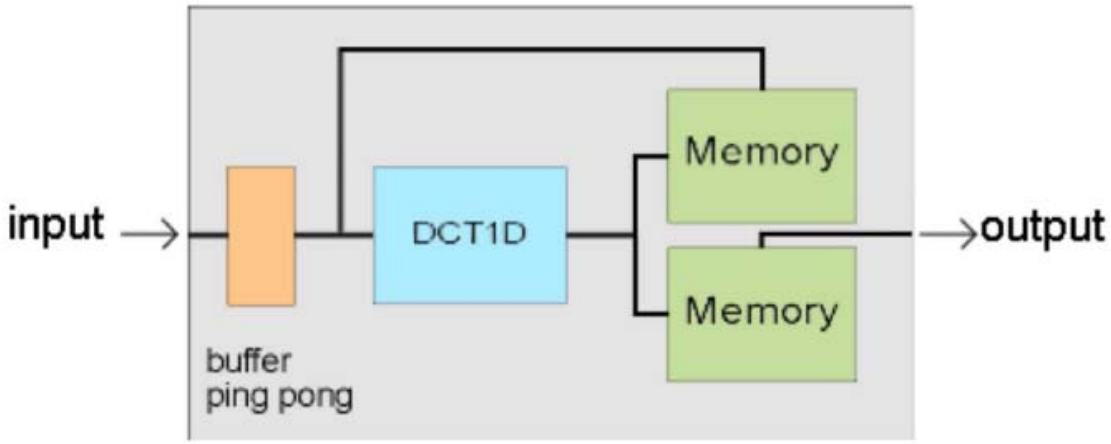

Figure 8. 2D-DCT with single seven step pipelined discrete cosine transform [8].

The implementation uses a single 1D-DCT pipeline that is fed by a master Fast Simplex Links (FSL) port and a transpose memory. The transpose memory feeds the transposed results from the first DCT computation back to the 1D-DCT so that the second 1D-DCT can be eliminated. The transpose memory is written in rows-wise but read from in column-wise format. When the second 1D-DCT is performed, the results are stored in another buffer before being transposed again where the values are output to the slave FSL.

By organizing the pipeline into seven stages, the number of adders is reduced from 29 sums and 5 multiplications to 19 sums and 4 multiplications. This is due to that the odd and even coefficients of the transformed vector requires different types of computations. Every cycle, the pipeline alternates the needed values and computes the odd then the even coefficients of the resulting vector separately.

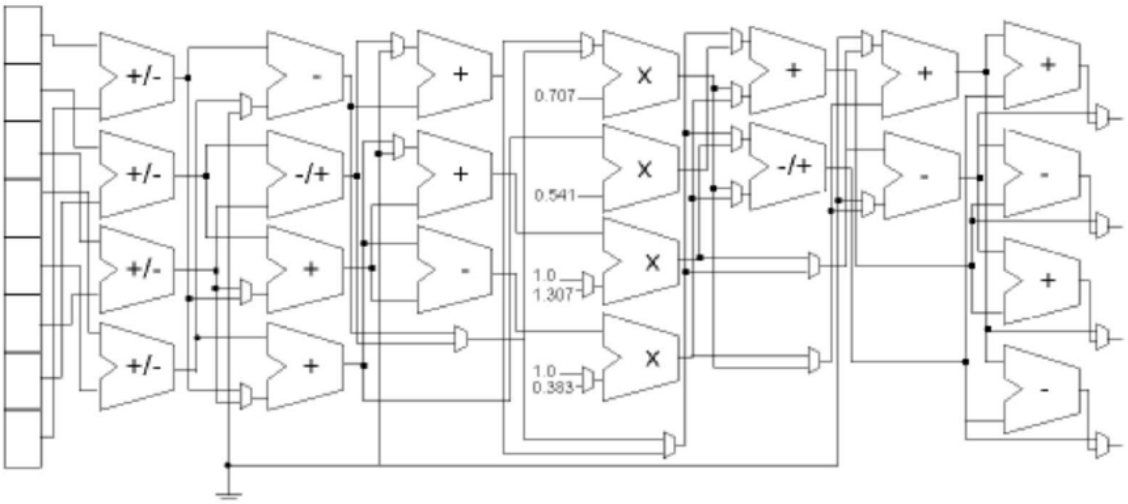

Figure 9. Seven step 1D-DCT [8]

A pseudo ping pong buffer maintain the same values for two consecutive clock cycles so that all 8 values of the input samples are available to the 1D-DCT. This is done because the FSL connection can only feed four 8 bit values per clock cycle [8]. The initial 8 bit values are expended by the DCT computation. Within the 2D-DCT process the values are computed with at least 24 bit floating point precision. The output values of the 2D-DCT are truncated to 16 bit integers.

The software first sends the data from the master port to the ping-pong buffer. The results are sent to the processor as two 16 bits values. The entire process takes 80 cycles (48 cycles for interface, 32 cycles for computation).

3.4 Implementation Result

Kovac simulated his design using Verilog and Veritime. He reasoned that since the entire architecture requires only a few major components a few adders, multipliers, and random access memory modules, the architecture can be realized in a single chip. The critical path of the chip implementation depends on a 14bit adder circuit which can be achieved with a 10ns clock period using 1um CMOS process. The operating frequency was proposed to be 100MHz. However the design can also be implemented on a single FPGA chip without the need to created a ASIC for the encoder.

Agostini's paper focuses only in a pipelined hardware implementation of the 2-D DCT process for JPEG image compression because of its computational complexity. The RTL level VHDL design can be reused for an ASIC implementation. His project was a attempt to carry out the entire encoder on a single FPGA. Although his design was an improvement on the work of Kovac and his colleagues, since modern FPGAs have embedded multipliers, the tactic of decomposing the multiplier in shift and add operations no longer offer the area optimization it would in an ASIC.

The Tumeo design offered a performance improvement that was two orders of magnitude faster than the software implementation while reducing the number of multipliers and adders from the Agostini implementation. Overall these improvements resulted in a 20% increase in performance with this implementation [8]. This is the ideal use of a hardware accelerator for SOC solutions. The paper points out that although the DCT process is a very good algorithm for hardware acceleration, the most

computationally intensive component may still be the color space conversion. The rest of this report will describe a hardware accelerator that optimizes the color space conversion for the encoder. As the trend for larger and crisper images increases, the reward of hardware acceleration for the encoder also increases.

Chapter 4: RGB to Y Cr Cb Conversion

The goal of this project is to implement the conversion of RGB to Y Cb Cr in Linux running on the The Learning Labs 5000 (TLL5000) platform using the I/O ports on the FPGA. The projects reviewed in this report thus far have all concentrated on the implementation of the discrete cosine transform in hardware. While this has been shown to successfully optimize the encoder by reduction of process time, optimization of the color space transformation has the potential of giving even more performance improvement because of its computational complexity.

4.1 Hardware and Software Computations

Y Cb Cr can be computed directly from 8-bit RGB as follows [3]:

$$Y = 0.299R + 0.587G + 0.114B$$
$$Cb = -0.1687R - 0.3313G + 0.5B + 128$$
$$Cr = 0.5R - 0.4187G - 0.0183B + 128$$

Commonly, the image file format store image samples in the order R0, G0, R0, to Rn, Gn, Bn. Each RGB data sequence is stored as one 24 bit memory or register. The red, green, and blue color values are represented by 8 bits each. For ease of testing and debugging, the test code have separated the 24 bit sequence. Three 8 by 8 integer matrix will will represent the red, green, and blue values respectively.

The software will process an entire 64 sample sequence of RGB values according to the formula listed above and store the Y Cb Cr values in three different matrices. This is done so that individual value computation times and average computation time can be recorded for debugging and comparison with hardware. The software will input each set

of RGB values from an 8 by 8 matrix into the ARM processor and the hardware will return each sequence of values back to the software to build the matrix of Y Cb Cr.

4.2 Software Hardware Communication

The software components consist of the user application and the device driver. The 'make' file uses tools from Codesourcery.com to compile the code for the ARM processor. When compiled with the '*make*' file, the *'user_application'* executable can interact with hardware code executed in the Linux operating system running on the ARM platform. In the *user_application.c* file, the code calls to the device driver. The device driver is opened as a file with a specific matrix to be converted. The *ioctl* calls direct the read-write commands and the corresponding values to the device driver. The conversion.c file implements the device driver. In the conversion device driver, calls are received from the *user_application* and redirected to specific addresses in the FPGA. These addresses are memory mapped in the FPGA. For this code they are xD3000004 to xD30000B memory locations.

The hardware receives individual Rx Gx Bx data from the software and gives the Yx Cbx Crx values as the output. The conversion is implemented in the Xilinx SPARTAN 3 FPGA. The hardware will read the values in xD3000004 to xD300000B, perform the conversion and return the result whenever addresses xD3000000 are read by the software. The file *top.v* performs this functionality with a seven-state finite state machine. The author used Xilinx ISE for the synthesis and the generation of the bit file which is then loaded to the FPGA.

4.3 Interrupt Handler

A *sighandler* function is called when an interrupt signal is read by the program. The code uses asynchronous notification. It set a *FASYNC* flag in the event of an interrupt via *fcntl()*. The signal handler is *signal()*. To obtain the current file control flags the program use *fcntl(fd, F_GETFL)*. The *fd* represents file descriptor. The asynchronous flag is set by *fcntl(fd, F_SETFL, oflag | FASYNC)*.

For the hardware implementation time, the difference between the start and stop time is the interrupts latency. The latency value is then calculated in microseconds. The *start_time* is recorded before the first write command is called. This is because the state machine in *top.v* goes into a state which triggers the rising edge of the interrupt as soon as the write command is recognized. The interrupt activates the *sighander*. If the "*signo*" is equal to "*SIGIO*", then the *det_int* flag is set to exit the while loop and the time is recorded in the *stop_time* array.

In the *top.v* file the counter register is utilized. An *add_int* and a *countloop* register are used. In state one, the count register accumulates as the Write command 1, write command 2 and write command 3 are called in the main program. When State 3 is reached, if count equals 3 then the interrupt is called and the state machine goes to State 6 which now computes the Y Cr and Cb results and stores them in a number of register and then set up the interrupt register *func_int* for a brief amount of time. The interrupt register, *count*, and *countloop* is cleared before the machine exits state 6. When the read

command is issued, the machine goes to state 7 where the results are read into *latch_data*. That will repeat three times for each pixel processed.

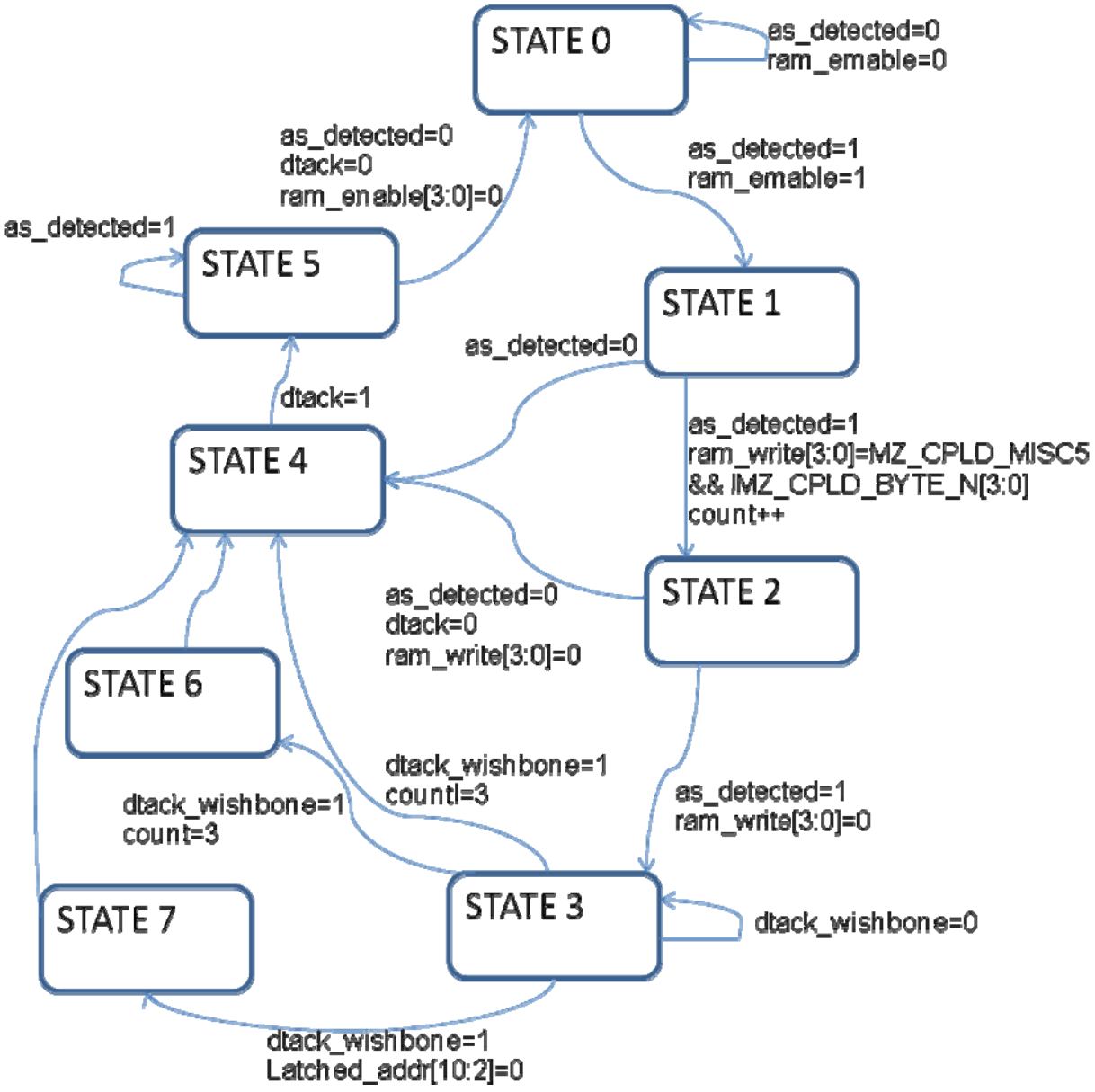

Figure 10. State diagram of *top.v*

Chapter 5: Testing and Results

Three 8 by 8 matrix of integers for the RGB values are created. The elements of the matrices correspond to specific colors for RGB tables. The start and stop times of both the software and the hardware implementation will be recorded and compared. First the *user_application* computes the software implementation. Two *for loops* cycle through all the elements of the three 8 by 8 matrix and computes the Y Cr and Cb results one set at a time. The *timeofday* function in C records the start time before the outer for loop is entered. Then after the compiler exits the outer *for* loop, the stop time is computed and the latency is measured and printed. Placing the bit of code here allows tester to obtain the average computation time per 24 bits of data.

The *user_application* then uses two *for* loops to compute the information again but this time it sends the data to be computed in the FPGA. The code record the start and stop time of the interrupt signal. The matrix elements use the *ioctrl* function to input data into the hardware. State six of the *top.v* code is the state where the computation is performed and also where the interrupt is generated upon completion. After the pixel data is sent to the hardware, the program goes into a wait-condition loop which frees up CPU cycles for other work. While inside this loop, the code waits for the interrupt which will set the flag *det_int* so that the program counter can exit the while loop. The *iter* is reset after that and the main while loop is executed again.

The initial goal of this project was to build a hardware accelerator that will accept input of 24 bit values from the software [11]. The hardware would mask out each 8 bit

value and compute the results. The results are then concatenated into another 24 bit value and returned to the software using the same interrupt approach. A copy of the JPEG software can be downloaded from *www.downloadfreescript.com*. The code was written to do this before any color state conversion code had been applied. However, because it was a time-intensive task to debug the code for computational accuracy, the color conversion process was set aside so that the values can be directly sent to the FPGA for computation and comparison. This step can be temporarily sacrificed because the masking of values, bit rotations, and sequencing has much lesser affect compared to multiplication and addition functions. Many mismatches between the hardware implementation results and the software implementation results appeared. In this experiment the software data severs as the control group for the color state values.

The hardware successfully receives data from the software. Using a number of print statements that author was able to step through each state of the state machine. The results are passed back to the software at the end of the computation. The end of computation works to activate the interrupt flag as well as to capture the amount of time spent by the hardware.

The error of the code lies in the computation process. If given additional time, one should rewrite the code to calculate each step of the conversion in software. The software results can be passed into the hardware and be compared with the FPGA computation to locate the error in the hardware code.

Successive innovations in FPGA and ADIC technology offers engineers new ways to improve the JPEG video and imaging encoding process. The papers discussed in

this report shows different ways to implement the JPEG encoding process. The rewards of each approach depend on the device being used and the specs required by the end users. Despite a number of errors in computation, the implementation of this design showed the benefit of using mixed software and hardware approach to improve the complicated data encoding process. The function of the interrupt handler and the loop counter were all successful. The initial test showed a 38% improvement in computation time using the hardware accelerator vs. the use of the full software color state conversion. In future work, the author plans to integrate the hardware code with existing color JPEG encoder software using at 24 bit data transfer process.

Appendices

Appendix A: Acronym Definitions

ASIC - Application Specific Integrated Circuit

FPGA- Field Programmable Gate Arrays

ZRL- Zero-Run-Length

EOB- End-of-Band

FSL - Fast Simplex Links

DCT- Discrete Cosine Transform

RGB- Red, Green, and Blue additive color model

SOC- System on Chip

VHDL- VHSIC Hardware Description Language

Appendix B: Hardware Accelerator Source Code

```
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/time.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>

#define READ_CMD1 (0x0 << 31)
#define READ_CMD2 (0x0 << 31)
#define READ_CMD3 (0x0 << 31)
#define WRITE_CMD1 (0x1 << 31)
#define WRITE_CMD2 (0x1 << 30)
#define WRITE_CMD3 (0x1 << 30)
#define INT_CMD (0x1 << 29)
#define ADDER_BASE 0xd3000000
#define ADDER_MASK 0x00001fff
#define ADDER_SIZE 0x20000
#define COMMAND_MASK 0x80000000

//////////////////////////
int num_int = 0;

int flag;
double delta, tmp_delta;
float time_buf;
int fd, i, j;
float Y_fpgaresult[8][8],Cr_fpgaresult[8][8],Cb_fpgaresult[8][8];
float Y_swresult[8][8],Cr_swresult[8][8],Cb_swresult[8][8];
i = 0;
struct timeval fpga_start_time;
struct timeval fpga_stop_time;
struct timeval fpga_latency;
struct timeval sw_start_time;
struct timeval sw_stop_time;
```

```
struct timeval sw_latency;
unsigned long target_addr;
unsigned int value;

int det_int = 0;

void sighandler(int signo)
{

  if (signo == SIGIO)
  {
        det_int++;

    gettimeofday(&fpga_stop_time, NULL);
  }

//   printf("\nMAIN: Interrupt captured by SIGIO\n");
  return;  /* Return to main loop */

}

char buffer[4096];

int main(int argc, char * argv[]) {

  float R[8][8] = {{192, 227, 195, 126, 164, 158, 163, 151 },
            {151, 216, 249, 151, 125, 212, 149, 185 },
            {208, 268, 0, 195, 186, 197, 229, 186 },
            {199, 159, 165, 248, 189, 255, 208, 123 },
                  {258, 158, 158, 0, 205, 218, 268, 228 },
            {139, 255, 182, 189, 205, 208, 221, 258 },
                  {199, 139, 195, 255, 255, 238, 139, 205 },
            {198, 264, 228, 238, 178, 186, 258, 238 }},
     G[8][8] = {{0, 91, 58, 150, 35, 82, 108, 64 },
            {198, 205, 130, 76, 84, 125, 169, 54 },
            {79, 111, 155, 244, 246, 142, 255, 157 },
            {188, 128, 255, 246, 134, 35, 139, 111 },
                  {66, 79, 0, 183, 118, 178, 250, 238 },
            {92, 186, 170, 118, 133, 128, 78, 205 },
                  {64, 51, 94, 99, 205, 147, 64, 64 },
            {94, 66, 128, 105, 51, 180, 102, 119 }},
     B[8][8] = {{51, 78, 66, 38, 100, 66, 51, 43 },
            {66, 40, 29, 0, 33, 0, 35, 140 },
```

```
                {55, 79, 0, 98, 100, 97, 80, 50 },
                {40, 150, 53, 161, 66, 119, 215, 36 },
                      {77, 25, 74, 46, 51, 51, 43, 69 },
                {82, 120, 35, 220, 180, 20, 181, 220 },
                      {213, 0, 37, 117, 0, 37, 205, 0 },
                {191, 130, 137, 186, 0, 150, 0, 102 }};

long int t1, t2;
long int fpga_time_diff;
long int sw_time_diff;

 int count;
 struct sigaction action;
 int rc, fc;

 sigemptyset(&action.sa_mask);
 sigaddset(&action.sa_mask, SIGIO);

 action.sa_handler = sighandler;
 action.sa_flags = 0;

 sigaction(SIGIO, &action, NULL);

 //Open the adder as a file
 fd = open("/dev/adder", O_RDWR);

 printf("fd %d\n",fd);
 if(!fd)
 {
  printf("Unable to open /dev/adder.");
  return -1;
 }

 printf("\nMAIN: /dev/adder opened successfully\n");

 fc = fcntl(fd, F_SETOWN, getpid());

 if (fc == -1)
 {
   perror("MAIN: SETOWN failed\n");
   rc = fd;
   exit (-1);
 }
```

```
 fc= fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_ASYNC);

 if (fc == -1)
 {
   perror("SETFL failed\n");
   rc = fd;
   exit (-1);
 }
 int flag2 = 1;

gettimeofday(&sw_start_time, NULL);
for(i = 0; i < 8; i++)
  for( j = 0; j < 8; j++)
  {
    Y_swresult[i][j] = 0.299 * R[i][j] + 0.587 * G[i][j] + 0.114 * B[i][j];
    Cb_swresult[i][j] = 0.5 * B[i][j] + 128 - 0.1687 * R[i][j] - 0.3313 * G[i][j];
    Cr_swresult[i][j] = 0.5 * R[i][j] - 0.4187 * G[i][j] - 0.0183 * B[i][j] + 128;
  }

gettimeofday(&sw_stop_time, NULL);

sw_latency.tv_sec = sw_stop_time.tv_sec - sw_start_time.tv_sec;
sw_latency.tv_usec = sw_stop_time.tv_usec - sw_start_time.tv_usec;
if (sw_latency.tv_usec < 0){
  sw_latency.tv_usec += 1000000, --sw_latency.tv_sec;  }
  sw_time_diff = sw_latency.tv_sec*1000000.0  + sw_latency.tv_usec;

printf("\n Computation Time of the software is %d \n", sw_time_diff);

gettimeofday(&fpga_start_time, NULL);
for(i = 0; i < 8; i++)
  for( j = 0; j < 8; j++)
  {
     flag = 0;
        ioctl(fd, WRITE_CMD1, &R[i][j]);
        ioctl(fd, WRITE_CMD2, &G[i][j]);
        ioctl(fd, WRITE_CMD3, &B[i][j]);

        ioctl(fd, READ_CMD1, &Y_fpgaresult[i][j]);
        Y_fpgaresult[i][j] = Y_fpgaresult[i][j]/10000;
        ioctl(fd, READ_CMD2, &Cr_fpgaresult[i][j]);
        Cr_fpgaresult[i][j] = Cr_fpgaresult[i][j]/10000;
```

```
            ioctl(fd, READ_CMD3, &Cb_fpgaresult[i][j]);
            Cb_fpgaresult[i][j] = Cb_fpgaresult[i][j]/10000;

            while(det_int == 0){

            }
            det_int = 0;
          }

        fpga_latency.tv_sec = fpga_stop_time.tv_sec - fpga_start_time.tv_sec;
        fpga_latency.tv_usec = fpga_stop_time.tv_usec - fpga_start_time.tv_usec;
        if (fpga_latency.tv_usec < 0){
          fpga_latency.tv_usec += 1000000, --fpga_latency.tv_sec;  }
          fpga_time_diff = fpga_latency.tv_sec*1000000.0  + fpga_latency.tv_usec;

        printf("\n Computation Time of FPGA is %d \n", fpga_time_diff);

        for(i = 0; i < 8; i++)
          for( j = 0; j < 8; j++)
          {
            if(Y_swresult[i][j] != Y_fpgaresult[i][j]){
              printf(" Computation for Y failed!\n");
            }
            if(Cb_swresult[i][j] != Cb_fpgaresult[i][j]){
              printf(" Computation for Cb failed!\n");
            }
            if(Cr_swresult[i][j] != Cr_fpgaresult[i][j]){
              printf(" Computation for Cr failed!\n");
            }
          }
          close(fd);
        return 0;
      }


module top (
            SYS_CLK,          // 24 MHz FPGA clock
            MZ_CPLD_CLKO,     // Clock from CPLD
            FPGA_CLK4,        // 27 MHz clock
            FPGA_CLK3,        // 100 MHz clock
            SYS_RST_N,
            MZ_CPLD_RESET_OUT, // Reset from CPLD
            MZ_BUF_DATA,
```

```
            MZ_BUF_ADDR,
            MZ_CPLD_AS,      // Address strobe - Active High
            MZ_CPLD_RW,
            FPGA_MZ_DTACK,   // DTACK FPGA->MZ Active High
            MZ_CPLD_BYTE_N,
            MZ_CPLD_MISC0,   // CPLD: State[0]
            MZ_CPLD_MISC1,   // CPLD: State[1]
            MZ_CPLD_MISC2,   // CPLD: State[2]
            MZ_CPLD_MISC3,   // CPLD: State[3]
            MZ_CPLD_MISC4,   // CPLD: oe_b
            MZ_CPLD_MISC5,   // CPLD: rw_b
            MZ_CPLD_MISC6,   // CPLD: cs0_b
            MZ_CPLD_MISC7,   // CPLD: cs1_b
            MZ_CPLD_MISC8,   // CPLD: cs2_b
            MZ_CPLD_MISC9,   // CPLD: cs3_b
            MZ_CPLD_MISC10,  // CPLD: cs5_b
            MZ_CPLD_MISC11,  // CPLD: nfio4
            MZ_CPLD_MISC12,  // CPLD: nfio5
            MZ_CPLD_MISC13,  // Undefined output to CPLD
            MZ_CPLD_MISC14,  // Interrupt from FPGA to CPLD
            LED,
            DIP_SW
          );

// ------------------------------------------------------------------
// INPUTS
// ------------------------------------------------------------------


input           SYS_CLK;
input           MZ_CPLD_CLKO;
input           FPGA_CLK4;
input           FPGA_CLK3;
input           SYS_RST_N;
input           MZ_CPLD_RESET_OUT;
input [23:0]      MZ_BUF_ADDR;
input           MZ_CPLD_AS;
input           MZ_CPLD_RW;
input [3:0]       MZ_CPLD_BYTE_N;
input [7:0]       DIP_SW;
input           MZ_CPLD_MISC0;
input           MZ_CPLD_MISC1;
input           MZ_CPLD_MISC2;
```

```
input           MZ_CPLD_MISC3;
input           MZ_CPLD_MISC4;
input           MZ_CPLD_MISC5;
input           MZ_CPLD_MISC6;
input           MZ_CPLD_MISC7;
input           MZ_CPLD_MISC8;
input           MZ_CPLD_MISC9;
input           MZ_CPLD_MISC10;
input           MZ_CPLD_MISC11;
input           MZ_CPLD_MISC12;

// -------------------------------------------------------------------
//   INOUTS
// -------------------------------------------------------------------

inout [31:0]      MZ_BUF_DATA;    // 32 bit data bus

// -------------------------------------------------------------------
//    OUTPUTS
// -------------------------------------------------------------------


output          FPGA_MZ_DTACK;  // DTACK back to Mezzanine State Machine
output [7:0]      LED;

output          MZ_CPLD_MISC13;
output          MZ_CPLD_MISC14;

// -------------------------------------------------------------------
//    REGISTERS
// -------------------------------------------------------------------

reg  [3:0]      count;          // Use to determine how long DTACK is asserted
reg  [3:0]      countloop;          // Use to pulse a interrupt signal in State 6
reg             dtack;          // Internal DTACK register

reg  [23:0]     latched_addr;
reg  [31:0]     latched_data;
reg  [31:0]     feedback_addr;

reg  [2:0]      state;
reg  [1:0]      as_sync;       // Sync Address Strobe from CPLD
```

```
reg               ram_enable;     // Based on one hot enable signals
reg   [3:0]         ram_write;

reg [31:0] operand_R;
reg [31:0] operand_G;
reg [31:0] operand_B;
reg [31:0] result_Y;
reg [31:0] result_Y1;
reg [31:0] result_Y2;
reg [31:0] result_Y3;
reg [31:0] Ycoffa;
reg [31:0] Ycoffb;
reg [31:0] Ycoffc;
reg [31:0] result_Cr;
reg [31:0] result_Cr1;
reg [31:0] result_Cr2;
reg [31:0] result_Cr3;
reg [31:0] Crcoffa;
reg [31:0] Crcoffb;
reg [31:0] Crcoffc;
reg [31:0] result_Cb;
reg [31:0] result_Cb1;
reg [31:0] result_Cb2;
reg [31:0] result_Cb3;
reg [31:0] Cbcoffa;
reg [31:0] Cbcoffb;
reg [31:0] Cbcoffc;
reg [3:0]  func_state;
reg [2:0]  out_state;
reg        add_req;
reg        add_ack;
reg          func_int;
reg [31:0] cval;

// -------------------------------------------------------------------
//  DATA BUS ASSIGNMENTS
// -------------------------------------------------------------------

      // This assignment puts the latched address back out on the data lines.

//wire [31:0]     MZ_BUF_DATA = (MZ_CPLD_RW && MZ_CPLD_AS) ?
latched_addr : 32'bz;
```

```
    // This assignment puts the latched address back out on the data lines.

wire [31:0]     MZ_BUF_DATA = (MZ_CPLD_RW && MZ_CPLD_AS) ? latched_data
: 32'bz;


    // This assignment puts the SWITCHES back out on the data bus lines.

//wire [31:0]     MZ_BUF_DATA = (MZ_CPLD_RW && MZ_CPLD_AS) ?
{4{DIP_SW}} : 32'bz;


// -------------------------------------------------------------------
//  Continuous assignments and wires
// -------------------------------------------------------------------

assign          FPGA_MZ_DTACK = dtack;    // Drive DTACK to CPLD

wire            dtack_wishbone = 1'b1;    // DTACK from components on the Wishbone Bus.

assign          ram_wr = ram_write[0] && ram_write[1] && ram_write[2] &&
ram_write[3];

assign          buffered_clk = MZ_CPLD_CLKO;     // Clock from CPLD
//assign          buffered_clk = FPGA_CLK4;       // 27 MHz clock
//assign          buffered_clk = FPGA_CLK3;       // 100 MHz clock
//assign          buffered_clk = SYS_CLK;         // 24 MHz FPGA clock

// -------------------------------------------------------------------
//
//  BLOCK RAM instantiations
//
// -------------------------------------------------------------------

wire   [31:0]  ram_data;
wire   [3:0]   parity_out;


RAMB16_S36 U_RAMB16_S36 (
            .DI(MZ_BUF_DATA[31:0]),   // 32-bit data_in bus ([31:0])
            .DIP(4'b0),               // 4-bit parity data_in bus ([35:32])
            .ADDR(latched_addr[10:2]), // 9-bit address bus
            .EN(ram_enable),          // RAM enable signal
```

```
            .WE(ram_wr),            // Write enable signal
            .SSR(1'b0),             // set/reset signal
            .CLK(!buffered_clk ),   // clock signal
            .DO(ram_data),          // 32-bit data_out bus ([31:0])
            .DOP(parity_out)        // 4-bit parity data_out bus ([35:32])
            );
// ----------------------------------------------------------------------------
//  Calculation
// ----------------------------------------------------------------------------

assign  result_Y1 = Ycoffa*operand_R;
assign  result_Y2 = Ycoffb*operand_G;
assign  result_Y3 = Ycoffc*operand_B;
assign  result_Cr1 = Crcoffa*operand_R;
assign  result_Cr2 = Crcoffb*operand_G;
assign  result_Cr3 = Crcoffc*operand_B;
assign  result_Cb1 = Cbcoffa*operand_R;
assign  result_Cb2 = Cbcoffb*operand_G;
assign  result_Cb3 = Cbcoffc*operand_B;

// ----------------------------------------------------------------------------
//
// Detect AS from CPLD - Synchronize through 2 flipflops. The CPLD is clocked
// at 64MHz and the FPGA is being clocked somewhere between 24MHz and 100MHz.
//
// ----------------------------------------------------------------------------

always @(negedge buffered_clk or negedge SYS_RST_N) begin
        if (!SYS_RST_N)  begin
            as_sync[1:0] <= 2'b0;
        end

        else if (SYS_RST_N) begin
            as_sync[1] <= as_sync[0];       // sync
            as_sync[0] <= MZ_CPLD_AS;       // Sample the input pin
        end
end

// ----------------------------------------------------------------------------
// AS from CPLD is detected when both FF's have a high signal. Negated immediately.
// ----------------------------------------------------------------------------

assign      as_detected = as_sync[1] &&
```

```
                    as_sync[0] &&
                    MZ_CPLD_AS;

// -----------------------------------------------------------------
//
// These assignments check that the switch and LEDS are working
// and then muxes various bus signals onto the LEDS for probing.
//
// -----------------------------------------------------------------

assign          LED = DIP_SW |
                  {buffered_clk,
                   MZ_CPLD_AS,
                   MZ_CPLD_MISC8,       // Chip Select 1
                   MZ_CPLD_MISC11,      // Chip Select 5
                   MZ_CPLD_RW,
                   MZ_CPLD_MISC5,       // Raw RW signal
                   ram_enable,
                   MZ_CPLD_MISC4
                   };

assign          MZ_CPLD_MISC14 = func_int;  // Interrupt from FPGA to CPLD
assign          MZ_CPLD_MISC13 = 1'b0;  // Undefined output to CPLD

// -------------------------------------------------------------------------
//          STATE MACHINE
// -------------------------------------------------------------------------

always @(posedge buffered_clk or negedge SYS_RST_N) begin

        if (!SYS_RST_N)  begin              // In RESET
                state           <= 3'b0;    // Start in State 0
                dtack           <= 1'b0;    // Negate dtack to CPLD
                func_state      <= 4'b0;
                out_state       <= 3'b0;
                func_int        <= 1'b0;
                count           <= 4'h0;
                Ycoffa          <= 3'hbae;  //2990;
                Ycoffb          <= 4'h16ee; //5870;
                Ycoffc          <= 3'h474;  //1140;

                Crcoffa         <= 3'h697;  //1687;
                Crcoffb         <= 3'hcf1;  //3313;
```

```
            Crcoffc         <= 4'h1388; //5000;

            Cbcoffa         <= 4'h1388; //5000;
            Cbcoffb         <= 4'h105b; //4187;
            Cbcoffc         <= 2'hb7;   //183;

            cval            <= 6'h138800;//1280000

    end

// ------------------------------------------------------------
// MAIN CONTROL LOOP
// ------------------------------------------------------------

    else if (SYS_RST_N) begin
        // ------------------------------------------------------------
        //   STATE 0
        // ------------------------------------------------------------

          if ((as_detected) && (state == 3'b000)) begin
             state         <= 3'b001;      // GOTO STATE 1
             ram_enable    <= 1'b1;        // Assert RAM enable
             latched_addr  <= MZ_BUF_ADDR;  // Latch address bus
                      func_int <= 1'b0;                 // clear interrupt
           end
          else if ((!as_detected) && (state == 3'b000)) begin
             state         <= 3'b000;       // Stay in STATE 0
             ram_enable    <= 1'b0;
             dtack         <= 1'b0;         // Negate dtack
                      func_int <= 1'b0;
          end

        // ------------------------------------------------------------
        //   STATE 1
        // ------------------------------------------------------------

          else if ((as_detected) && (state == 3'b001)) begin
             state        <= 3'b010;     // GOTO STATE 2
             ram_write[0]  <= !MZ_CPLD_MISC5 && !MZ_CPLD_BYTE_N[0];
             ram_write[1]  <= !MZ_CPLD_MISC5 && !MZ_CPLD_BYTE_N[1];
             ram_write[2]  <= !MZ_CPLD_MISC5 && !MZ_CPLD_BYTE_N[2];
             ram_write[3]  <= !MZ_CPLD_MISC5 && !MZ_CPLD_BYTE_N[3];
                              if(latched_addr[10:2] == 9'h01) begin
```

```
                        count <= count + 1;
                    end
                    if(latched_addr[10:2] == 9'h02) begin
                        count <= count + 1;
                    end
  end
  else if ((!as_detected) && (state == 3'b001)) begin
     state        <= 3'b100;    // GOTO  STATE 4
  end

// -------------------------------------------------------------
//    STATE 2
// -------------------------------------------------------------

  else if ((as_detected) && (state == 3'b010)) begin
     state         <= 3'b011;     // GOTO STATE 3
     ram_write[3:0] <= 4'b0;       // Disable writes to RAM

  end
  else if ((!as_detected) && (state == 3'b010)) begin
     state        <= 3'b100;    // Go to STATE 4
     dtack        <= 1'b0;      // Negate dtack to CPLD
     ram_write    <= 4'b0;       // Disable writes to RAM
   end

// -------------------------------------------------------------
//    STATE 3
// -------------------------------------------------------------

  else if ((!dtack_wishbone) && (state == 3'b011)) begin
      state        <= 3'b011;     // STAY IN STATE 3
  end

  else if ((dtack_wishbone ) && (state == 3'b011)) begin
     //state         <= 3'b100;    // GOTO  STATE 4

     latched_data   <= (ram_data | {4{DIP_SW}});  // Read the data

                    if((latched_addr[10:2] == 9'b0)) begin
                        state <= 3'b111;         // GOTO  STATE 7
                    end
                    if(count == 4'h3) begin
                        state <= 3'b110;         // GOTO  STATE 6
```

```
                        end
                        else begin
                                state <= 3'b100;        // GOTO  STATE 4
                        end
  end

// ------------------------------------------------------------
//   STATE 4
// ------------------------------------------------------------

  else if ( state == 3'b100) begin
      state          <= 3'b101;       // GOTO STATE 5
      dtack          <= 1'b1;         // Assert dtack
  end

// ------------------------------------------------------------
//   STATE 5
// ------------------------------------------------------------

   else if (as_detected & ( state == 3'b101)) begin
      state         <= 3'b101;       // Stay in STATE 5
  end

  else if (!as_detected & ( state == 3'b101)) begin
      state          <= 3'b000;       // GOTO STATE 0
      dtack          <= 1'b0;         // Negate dtack
      ram_enable     <= 1'b0;

  end
// ------------------------------------------------------------
//   STATE 6
// ------------------------------------------------------------

      else if(state == 3'b110) begin
                if(func_state == 4'b0000) begin
                ram_enable     <= 1'b1;// Assert RAM enable
                latched_addr[10:2]   <= 9'h001;//Latch address bus
                ram_write      <= 4'b0;
                func_state     <= 4'b0001;
      end
      else if(func_state == 4'b0001) begin
                operand_R      <= ram_data;
                func_state <= 4'b0010;
```

```
end
else if(func_state == 4'b0010) begin
        ram_enable    <= 1'b1; // Assert RAM enable
        latched_addr[10:2]   <= 9'h002;//Latch address bus
        ram_write     <= 4'b0;
        func_state    <= 4'b0011;
end
else if(func_state == 4'b0011) begin
        operand_G <= ram_data;
        func_state <= 4'b0100;
end
else if(func_state == 4'b0100) begin
        ram_enable    <= 1'b1; // Assert RAM enable
        latched_addr[10:2]   <= 9'h003;// Latch address bus
        ram_write      <= 4'b0;
        func_state    <= 4'b0011;
end
        else if(func_state == 4'b0101) begin
        operand_B     <= ram_data;
        func_state      <= 4'b0111;
end
else if(func_state == 4'b0111) begin
        result_Y <= result_Y1+result_Y2+result_Y3;
        result_Cr <= result_Cr3-result_Cr1-result_Cr2+cval;
        result_Cb <= result_Cb1-result_Cb2-result_Cb3+cval;
        func_state <= 4'b1000;
        ram_enable <= 1'b0;
end
else if(func_state == 4'b1000) begin
        func_state <= 4'b1001;
        ram_enable <= 1'b1;
        ram_write  <= 4'hF;
end
else if(func_state == 4'b1001) begin
        ram_write     <= 4'h0;
        if(countloop == 4'h4) begin
                func_state <= 4'b0;
                state <= 3'b100;
                count <= 4'h0;
                countloop <= 4'h0;// clear interrupt loop counter
        end
else begin
        func_int <= 1'b1;// set interrupt after addition
```

```
                    func_state <= 4'b1000;
                    countloop <= countloop+1;
                end
        end
end

            // ------------------------------------------------------------
            //   STATE 7
            // ------------------------------------------------------------

              else if ( state == 3'b111) begin

                                if(out_state == 3'b0) begin
                                        latched_data <= result_Y;
                                        ram_enable <= 1'b0;
                                        state <= 3'b100;
                                        out_state <= 3'b01;
                                end
                                else if(out_state == 3'b01) begin
                                        latched_data <= result_Cr;
                                        ram_enable <= 1'b0;
                                        state <= 3'b100;
                                        out_state <= 3'b10;
                                end
                                else (out_state == 3'b10) begin
                                        latched_data <= result_Cb;
                                        ram_enable <= 1'b0;
                                        state <= 3'b100;
                                        out_state <= 3'b00;
                                end
                end
        end         //  END of MAIN CONTROL LOOP
    end          // END of State Machine LOOP
endmodule
```

References

[1] G. K. Wallace, “The JPEG still picture compression standard,” CACM, vol. 34, no. 4, pp.31-44, 1991.

[2] W. Penneker, J. Mitchell. JPEG Still Image Data Compression Standard, Van Nostrand Reinhold, USA, 1992.

[3] “Home site of the JPEG and JBIG committees” <http://www.jpeg.org/> (2010-07-28).

[4] Léger, A., Omachi, T., and Wallace, G. The JPEG still picture compression algorithm. In Optical Engineering, vol. 30, no. 7 (July 1991), pp. 947-954.

[5] M. Kovac, N. Ranganathan. “JAGUAR: A Fully Pipeline VLSI Architecture for JPEG Image Compression Standarded”. Proceedings of the IEEE, vol. 83, no. 2, 1995, pp. 247-258.

[6] L. Agostini, S. Bampi, " Integrated Digital Architectures for JPEG Image Compression, " European Conference on Circuit Theory and Design, Vol 3, pp. 181-184, 2001.

[7] “JPEG Compression” <http://www.fileformat.info/mirror/egff/ch09_06.htm> (2010-09-10).

[8] A. Tumeo, M. Monchiero, " Pipelined Fast 2D-DCT Accelerator for FPGA-based SoCs, " IEEE Computer Society Annual Symposium on VLSI, pp. 331 – 336, 2007.

[9] ISO/IEC, Int. Standard DIS 10918, “Digital compression and coding of continuous-tone still images.”

[10] Y. Arai, T. Agui, and M. Nakajima, “A fast DCT-SQ scheme for images,” Trans. IEICE, vol. E71, no. 11, pp. 1095-1097, 1988.

[11] Altera Digital Library, Altera Corporation, 2000

Vita

Feng Zheng was born in Beijing, China. After completing his work at Austin High School, Sugar Land, Texas in 2001, he entered University of Texas at Austin, Texas. He received the degree of Bachelor of Science from The University of Texas at Austin in 2005. During the following years, he was employed as a facilities engineer at Samsung Austin Semiconductor. In January, 2008, he entered the Graduate School at the University of Texas at Austin.

Permanent Email Address: fengsymphony@gmail.com

This report was typed the author.