1-1-2008

Implementation of Jpeg compression and motion estimation on Fpga hardware

Ramakrishna Gopalakrishnan
University of Nevada, Las Vegas

IMPLEMENTATION OF JPEG COMPRESSION AND MOTION ESTIMATION ON FPGA HARDWARE

by

Ramakrishna Gopalakrishnan

Bachelor of Engineering
Anna University, Chennai, India
2006

A thesis submitted in partial fulfillment
of the requirement for the

Master of Science Degree in Electrical Engineering
Department of Electrical and Computer Engineering
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2008

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

July 24, 20 08

The Thesis prepared by

Ramakrishna Gopalakrishnan

Entitled

"Implementation of JPEG Compression and Motion Estimation of FPGA Hardware"

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

7/24/08

Graduate College Faculty Representative

ABSTRACT

Implementation of JPEG Compression and Motion Estimation on FPGA Hardware

by

Ramakrishna Gopalakrishnan

Dr. Henry Selvaraj, Examination Committee Chair
Professor of Electrical Engineering
University of Nevada, Las Vegas

A hardware implementation of JPEG allows for real-time compression in data intensivve applications, such as high speed scanning, medical imaging and satellite image transmission. Implementation options include dedicated DSP or media processors, FPGA boards, and ASICs. Factors that affect the choice of platform selection involve cost, speed, memory, size,power consumption, and case of reconfiguration. The proposed hardware solution is based on a Very high speed integrated circuit Hardware Description Language (VHDL) implememtation of the codec with prefered realization using an FPGA board due to speed, cost and flexibility factors.

The VHDL language is commonly used to model hardware impletations from a top down perspective. The VHDL code may be simulated to correct mistakes and subsequently synthesized into hardware using a synthesis tool,such as the xilinx ise suite. The same VHDL code may be synthesized into a number of sifferent hardware architetcures based on constraints given. For example speed was the major constraint when synthesizing the pipeline of jpeg encoding and decoding, while chip area and power

consumption were primary constraints when synthesizing the on-die memory because of large area. Thus, there is a trade off between area and speed in logic synthesis.

ACKNOWLEDGEMENTS

Active support of a wide variety of professors from my department has helped my learning and research on this thesis. They are Dr. Henry Selvaraj, Dr. Muthukumar Venkatesan, Dr. Emma Regentova and Dr. Laxmi Gewali, who are all distinguished professors in Electrical Engineering and Computer Science department at UNLV. They shared their insights and allowed me to explore the concepts towards comprehensive learning while constantly guiding towards focused research. I am fortunate to have Dr. Henry Selvaraj willing to review early drafts of the thesis and offer very constructive criticism for improvement.

Arun Reddy Toomu and Nachiket Jugade, both colleagues and Graduate Assistants at Electrical Department, UNLV provided me detailed feedback and provided essential help. This effort would have been impossible without the active support of my parents. Their unstinting support and willingness to take the burdens and provide support all through made all this possible. My deep thanks to friends and my other family members.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Introduction to JPEG Compression and Motion Estimation

A digital color image is a collection of pixels with each pixel a 3-dimensional (3-D) color vector. The vector elements specify the pixel's color with respect to a chosen color space; for example, RGB, YCbCr, etc [1, 2]. Joint Photographic Experts Group (JPEG) is a commonly used standard to compress digital color images [7]. JPEG is "lossy," meaning that the decompressed image isn't quite the same as the one you started with. (There are lossless image compression algorithms, but JPEG achieves much greater compression than is possible with lossless methods.) JPEG is designed to exploit known limitations of the human eye, notably the fact that small color changes are perceived less accurately than small changes in brightness. Thus, JPEG is intended for compressing images that will be looked at by humans. If you plan to machine-analyze your images, the small errors introduced by JPEG may be a problem for you, even if they are invisible to the eye.

The temporal prediction technique used in MPEG video is based on motion estimation. The basic premise of motion estimation is that in most cases, consecutive video frames will be similar except for changes induced by objects moving within the

frames. In the trivial case of zero motion between frames (and no other differences caused by noise, etc.), it is easy for the encoder to efficiently predict the current frame as a duplicate of the prediction frame. When this is done, the only information necessary to transmit to the decoder becomes the syntactic overhead necessary to reconstruct the picture from the original reference frame. When there is motion in the images, the situation is not as simple. The problem for motion estimation to solve is how to adequately represent the changes, or differences, between these two video frames.

1.2 Why FPGA's?

For many years, electronic hardware used for computation could be divided into two main types, general purpose, and application specific. General-purpose hardware is exemplified by microprocessors such as the Intel 80x86 families and the Motorola 68000 family, which serve as the main processing unit in most personal computers. The architecture of these devices is fixed and includes specific hardware to implement a limited, pre-defined, set of instructions. These microprocessors run programs, which are lists of instructions to be executed that are stored in external memory. New programs can be loaded into memory from disk or other storage as needed. The software program determines the computation to be done, not the hardware. However, general-purpose computers can be very slow at performing certain kinds of operations, such as those involving floating-point calculations or complex mathematical functions.

Application-specific computing hardware performs functions very quickly, but the price of this speed is limited flexibility. As their name implies, this type of hardware can only perform one function, or a group of closely related functions. The hardware

determines the type of computation to be done. They cannot be reprogrammed to perform entirely new functions that were not anticipated and included in the original design. If application specific hardware is needed to perform a new function, then a new hardware design will have to be created. Since this type of computation hardware is generally implemented as carefully designed Application Specific Integrated Circuits (ASICs), creating a new design takes a great deal of effort and knowledge. Since they are custom ICs, they are also very expensive to fabricate, and it takes week or months to design a new ASIC and have it fabricated.

In recent years, a new class of computing hardware has been gaining increasing research interest. Configurable computing hardware has some of the advantages of both general-purpose and application-specific hardware. This type of hardware may be based on commercially available Field Programmable Gate Arrays (FPGAs), or on ICs designed specifically for the purpose. In either case, this type of hardware consists of a relatively large number of functional units with programmable interconnections. The functionality of the hardware is determined by how the interconnections between functional units are configured, and in most, but not all, architectures, how the functional units themselves are configured. By changing the configuration, the hardware can be made to perform a completely different function. Since the configuration is specific to the application at hand, it is in effect a custom computer for the particular design.

In order to map an application to this hardware, we must first design the hardware configuration needed to perform the necessary functions. This is done with either schematic capture or in this case with a Hardware Description Language (HDL) known

as VHDL. In either case, we must understand digital design and be able to separate an application into data processing and control elements. The design must then be partitioned spatially, so that the design is spread across the resources available on the FPGAs. If the design does not fit in the available FPGAs, then it must also be partitioned temporally, by allocating functional units to different configurations of the same FPGA.

1.3 Steps Involved in Hardware Implementation

The major components used in this implementation are

- PC for software interface (Active HDL 7.2 & Xilinx ISE 9.1)
- Virtex-4 FPGA board
- Microcontroller Frame Grabber (uCFG)
- Camera
- RS-232 (9-pin) for serial interface between uCFG and Virtex-4 FPGA board

Firstly, an image frame is captured with the help of a camera. The camera is connected to a microcontroller frame grabber which calls for the image and stores it in its EEPROM. The frame is then segregated into luminance(Y) and Chroma (Cr, Cb) signals. The luminance and chroma values are stored in the FIFO buffer of the frame grabber. The Frame Grabber and the Virtex-4 FPGA board are connected serially with a 9 pin RS-232 null cable. The FPGA communicates with the Microcontroller (MCU) in the uCFG through this serial cable with the help of UART-Transmitter and Receiver modules. When asked for, the microcontroller in the frame grabber communicates with FIFO buffer and transmits bits (ie) the luminance or chrominance values are as requested by the FPGA. The luminance or chrominance values are selected using a series of commands

which increase the address of the read pointer accordingly. All the transmitted bits are stored in a ROM which is instantiated in the FPGA. Once the MCU finishes transmitting all the bits (8 x 8) the code developed in VHDL comes into the picture. These VHDL modules are synthesized using XILINX ISE 9.1. All the necessary place and route operations are done according the requirements and all the inputs and outputs are assigned to the FPGA pins. Then the code is run on the stored pixel values and a compressed bit stream of 32 bits is received as the output. The outputs are seen on the LED's on the FPGA board.

CHAPTER 2

JPEG COMPRESSION MODULE

The proposed JPEG standard aims to be generic and support a wide variety of applications for continuous-tone images [3]. To meet the differing needs of many applications, the JPEG standard includes two basic compression methods, each with various modes of operation. A DCT-based method is specified for "lossy'' compression, and a predictive method for "lossless'' compression. JPEG features a simple lossy technique known as the Baseline method, a subset of the other DCT-based modes of operation.

2.1 Block Diagram

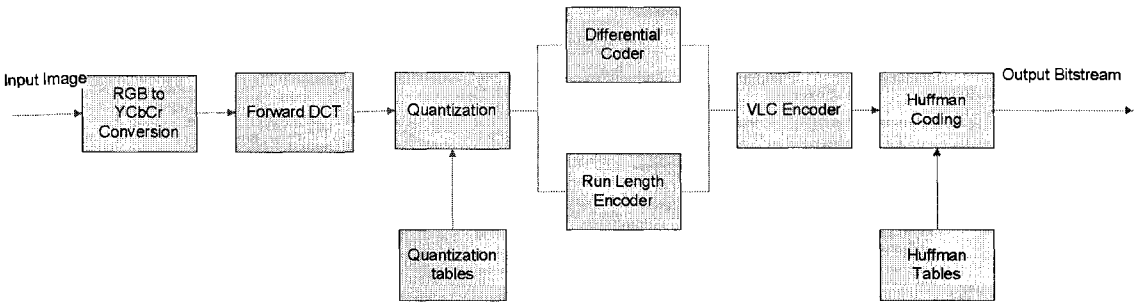

Figure 1 JPEG Compression Module

2.1.1 Discrete cosine transformation

Discrete Cosine Transform (DCT) is a lossy compression scheme where an N x N image block is transformed from the spatial domain to the DCT domain. A related transform, the discrete cosine transform (DCT), does not have complex values. The DCT is a separate transform and not the real part of the DFT. It is widely used in image and video compression applications, e.g., JPEG and MPEG. It is also possible to use DCT for filtering using a slightly different form of convolution called symmetric convolution. DCT decomposes the signal into spatial frequency components called DCT coefficients [5]. The lower frequency DCT coefficients appear toward the upper left-hand corner of the DCT matrix, and the higher frequency coefficients are in the lower right-hand corner of the DCT matrix. The Human Visual System (HVS) is less sensitive to errors in high frequency coefficients than it is to lower frequency coefficients. Because of this, the higher frequency components can be more finely quantized, as done by the quantization matrix. Each value in the quantization matrix is pre-scaled by multiplying by a single value, known as the quantizer scale code. This value can range in value from one to 112 and is modifiable on a macro block basis. Dividing each DCT coefficient by an integer scale factor and rounding the results accomplishes quantization. This sets the higher frequency coefficients (in the lower right corner), that are less significant to the compressed picture, to zero by quantizing in larger steps. The low frequency coefficients (in the upper left corner), are more significant to the compressed picture, and are quantized in smaller steps. The goal of quantization is to force as many of the DCT coefficients to zero, or near zero, as possible within the boundaries of the prescribed bit-

rate and video quality parameters. Thus, since quantization throws away some information, it is a lossy compression scheme.

The data compressed at the transmitter needs to be decompressed at the receiver. IDCT is used to decompress DCT compressed data in the decoder. DCT and IDCT are two of the most computation intensive functions in compression. Therefore, a fast and optimized DCT/IDCT implementation is essential in improving the performance of the video coder and decoder.

2.1.2 Quantization

The first processing step breaks the image into a stream of 8x8 blocks of pixels and transforms these grayscale values into the frequency domain using a Forward DCT (FDCT) [6]. Transforming the coefficients into the frequency domain causes most of the energy to reside in the DC and low frequency terms. This occurs because pixel values do not vary much within such a small region and in general yields greater compression ratios. The output of the FDCT results in a set of 64 basis-signal amplitudes. These amplitudes, or coefficients, are then uniformly quantized with a 64 element quantization table (QTABLE).

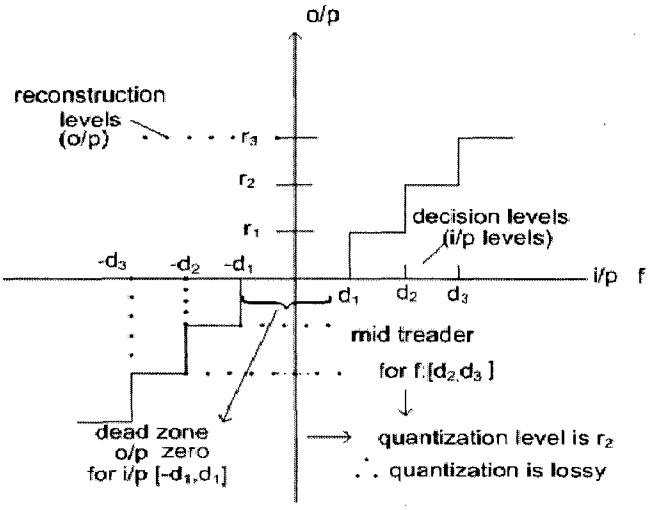

Figure 2 Uniform Quantizer

For 12-bit imagery, each element of the QTABLE can be in the range of 1 to 4095. This value specifies the scale factor, or step size, that is applied to the corresponding coefficient. Scaling the coefficients with the QTABLE results in the greatest source of pixel reconstruction error in JPEG, but also provides the greatest amount of compression. After quantization, the resulting values are rounded to the nearest integer. Finally, the coefficients are entropy encoded using either Huffman coding or arithmetic coding.

The amount of compression is controlled by quantizing the coefficients resulting from the FDCT. Our first attempt at finding an appropriate QTABLE for the cervical image set looked at differences between the distribution of the noise and signal coefficients. This proved futile due to the high resolution of the images relative to the 8x8 DCT block size. That is, since the resolution of the image is so high, anatomical structure is spread among many 8x8 blocks. This results in blocks with low variance and shifts most of the energy

into the lowest frequency component, known as the DC coefficient. This component is labeled DC with due reference to the terminology of direct current. The remaining coefficients are all labeled as AC.

2.1.3 Run Length Encoder

Run-length Encoding or RLE is a technique used to reduce the size of a repeating string of characters. This repeating string is called a run; typically RLE encodes a run of symbols into two bytes, a count and a symbol. RLE can compress any type of data regardless of its information content, but the content of data to be compressed affects the compression ratio. RLE cannot achieve high compression ratios compared to other compression methods, but it is easy to implement and is quick to execute.

Compression is normally measured with the compression ratio:

Compression Ratio = original size / compressed size : 1

In run-length encoding, repetitive source such as a string of numbers can be represented in a compressed form, for example,

1,4,5,1,4,5,1,4,5

can be compressed to form

3(1,4,5)

Thus, giving a compression ratio of = 9/4:1 which is almost 2: 1.

2.1.4 Entropy and Variable Length Codeword

Uniform length codeword assignment is not in general optimal in terms of the required average bit rate. Suppose some message probabilities are more likely to be sent than others. Then by assigning shorter codewords to the more probable message

possibilities and longer codewords to the less probable message possibilities, we may be able to reduce the average bit rate.

Codewords whose lengths are different for different message possibilities are called variable-length codewords. When the codeword is designed based on the statistical occurrence of different message probabilities, the design method is called statistical coding. To discuss the problem of designing codewords such that the average bit rate is minimized, we define an entropy H as:

$$H = -\sum_{i=1}^{L} P_i \log_2 P_i$$

where P_i is the probability that the message will be a_i since $\sum_{i=1}^{L} P_i - 1$ it can be shown that

$0 \leq H \leq \log_2 L$

The entropy H can be interpreted as the average amount of information that a message contains. Suppose L=2, if $P_1 = 0$ and $P_2 = 0$, H is zero and is the minimum possible for L = 2. In this case the message is a_1 with probability of 1; i.e. the message contains no new information. At the other extreme, suppose $P_1 = P_2 = 1/2$. The entropy H is 1 and is the maximum possible for L = 2. In this case the two message possibilities a_1 and a_2 are equally likely. Receiving the message clearly adds new information.

2.1.5 Huffman Encoding

It is a lossless data compression algorithm which uses a small number of bits to encode common characters. Huffman coding approximates the probability for each

character as a power of 1/2 to avoid complications associated with using a non-integral number of bits to encode characters using their actual probabilities.

Huffman coding works on a list of weights $\{w_i\}$ by building an extended binary tree with minimum weighted path length and proceeds by finding the two smallest ws, w_1 and w_2, viewed as external nodes, and replacing them with an internal node of weight $w_1 + w_2$. The procedure is them repeated stepwise until the root node is reached. An individual external node can then be encoded by a binary string of 0s (for left branches) and 1s (for right branches).

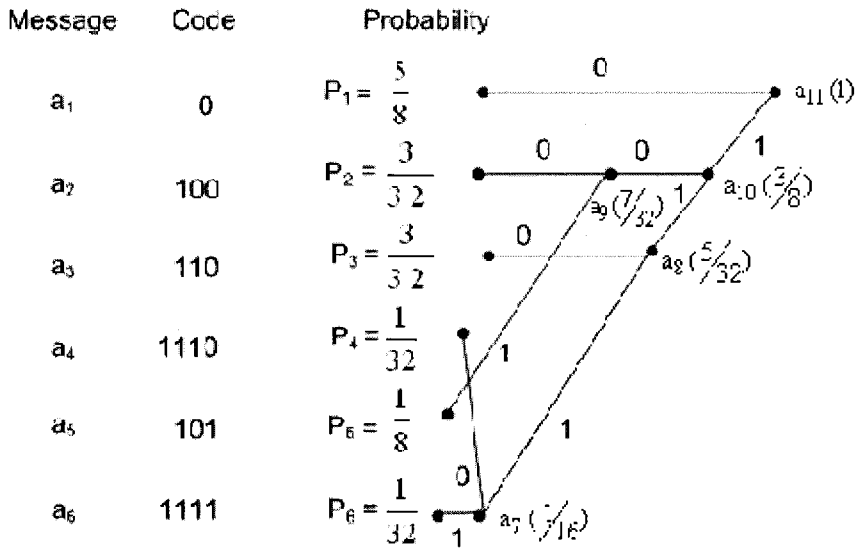

Figure 3 Illustration of codeword generation in Huffman Coding

An example of Huffman coding is shown in Figure 3. In the example L = 6 with the probability for each message possibility noted at each node.

Message	Codeword	Probability
a_1	0	$P_1 = 5/8$
a_2	100	$P_2 = 3/32$
a_3	110	$P_3 = 3/32$
a_4	1110	$P_4 = 1/32$
a_5	101	$P_5 = 1/8$
a_6	1111	$P_6 = 1/32$

In the Ist step of Huffman coding, we select the two message possibilities that have two lowest probabilities. We combine them and form a new node with combined probabilities. We assign '0' to one of the two branches and '1' to other. Reversing this affects the codeword but not the average bit rate. We continue with this process until we are left with one message with probability '1'. To determine the specific codeword assigned to each message possibility, we begin with last node with probability '1', follow the branches that lead to the message possibility of interest and combine the 0's and 1's on the branches.

For example, a_4 has codeword 1110. To compare performance of Huffman coding with the entropy H and uniform length codeword assignment for the above example, we

compute average bit rate achieved by uniform length codeword, Huffman coding and the entropy respectively.

CHAPTER 3

MOTION ESTIMATION MODULE

3.1 Block Diagram

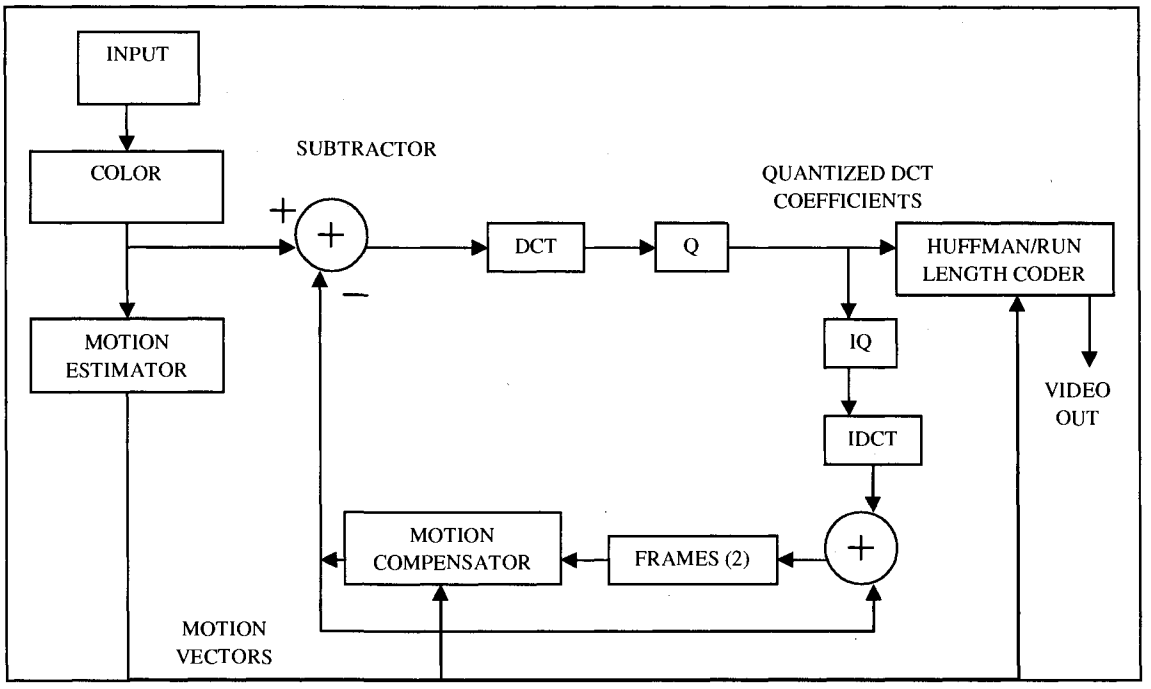

Figure 4 MPEG Module

3.2 What is Motion Estimation?

Motion estimation is the processes which generates the motion vectors that determine how each motion compensated prediction frame is created from the previous frame. Block Matching (BM) is the most common method of motion estimation. Typically each macroblock (16×16 pixels) in the new frame is compared with shifted regions of the same size from the previous decoded frame, and the shift which results in the minimum error is selected as the best motion vector for that macroblock. The motion compensated prediction frame is then formed from all the shifted regions from the previous decoded frame.

BM can be very computationally demanding if all shifts of each macroblock are analyzed. For example, to analyze shifts of up to ±15 pixels in the horizontal and vertical directions requires $31 \times 31 = 961$ shifts, each of which involves $16 \times 16 = 256$ pixel difference computations for a given macroblock. This is known as exhaustive search BM. Significant savings can be made with hierarchical BM, in which an approximate motion estimate is obtained from exhaustive search using a low pass sub sampled pair of images, and then the estimate is refined by a small local search using the full resolution images. Sub sampling 2:1 in each direction reduces the number of macroblock pixels and the number of shifts by 4:1, producing a computational saving of 16:1.

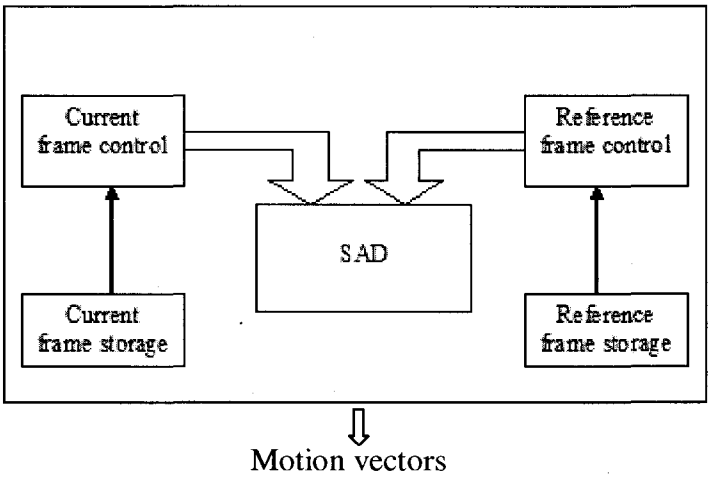

Figure 5 Motion Estimator Module

3.2.1 Reference frame storage

This is the very important part of this module. The current macroblocks are 16x16 blocks which contain the current frame information and have to be compared with the reference macroblocks which are already stored.

3.2.2 Current frame storage

The current frame storage is done in a similar way as the reference frame storage. The only difference is segregation has to be done for 16x16 macroblocks. It requires less memory and is faster as each time only 256 bytes of luminance pixels have to be read.

3.2.3 Reference frame control

This module is a sliding window controller which sweeps across the 32x32 search area i.e. 1024 pixels. The Macroblock at each specific point in the sliding window is latched and fed to the SAD module for further computation. The sliding window gives an

accurate account of overall search area. The probability of finding the best match increases in this case.

3.2.4 Current Frame Control

This module is a 8-byte Shift Register (SR) which shifts the 256 different luminance values of the current macroblock. As soon as all the 256 values have been clocked in the shift register, all the values are latched into the 256in-256out structure, which then feeds concurrently into the input of SAD block. The synchronization is a bit complex but not as complex as in the case of logarithmic algorithms.

3.2.5 SAD Module

SAD module has inputs as reference and current macroblocks and outputs the motion vectors XY-coordinates. An important metric used in motion estimation is the sum of absolute differences (SAD). $\sum_{k=0}^{M-1} \sum_{l=0}^{N-1} | (C(x+k, y+l) - R(x+i+k, y+j+l)) |$. The absolute difference operation can be implemented in several ways: serial, per column in parallel, per row in parallel, and fully parallel. The implementation described in [5] focuses on the SAD16 operation that performs the SAD on one row of a macroblock (16x1). All the input values are 8-bit unsigned binary numbers. By iteration or parallel execution of the SAD16 operation, the complete SAD operation for the 16x16 macroblock can be performed. First, the steps necessary to perform the 16x1 SAD operations in more detail:

- Determine the smaller of the two operands: As suggested in [3, 4], it is only necessary to determine whether (A' + B) produces a carry or not.
- Invert the smallest operand: If no carry was produced then B must be inverted; otherwise, A must be inverted. This is done by utilizing an EXOR operation.

- Pass both operands to an adder tree: After inverting either A or B, the operands must be passed to an adder tree. Thus, the values (A', B) or (A, B') are passed further.
- Add a correction term to the adder tree: Also an additional correction term must be added to the adder tree which is 16 in this case i.e. adding 1 to each of the 16 blocks.
- Reduce the 33 addition terms to 2: All 33 addition terms must be reduced to 2 terms before the final addition can be applied. This can be done using an 8-stage carry save adder tree using 243 carry save adders.
- Add the remaining two terms using an adder: The final two addition terms are added using an 8-bit carry look ahead adder for the most significant bits. The result is a 13-bit unsigned binary number. However, as stated in [4, 5], the most significant bit of this result can be disregarded resulting in a final 12-bit unsigned binary number.

CHAPTER 4

HARDWARE IMPLEMENTATION SETUP

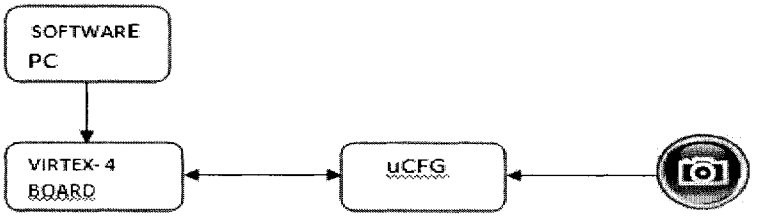

Figure 6 Hardware Implementation Setup

4.1 Virtex-4 Board

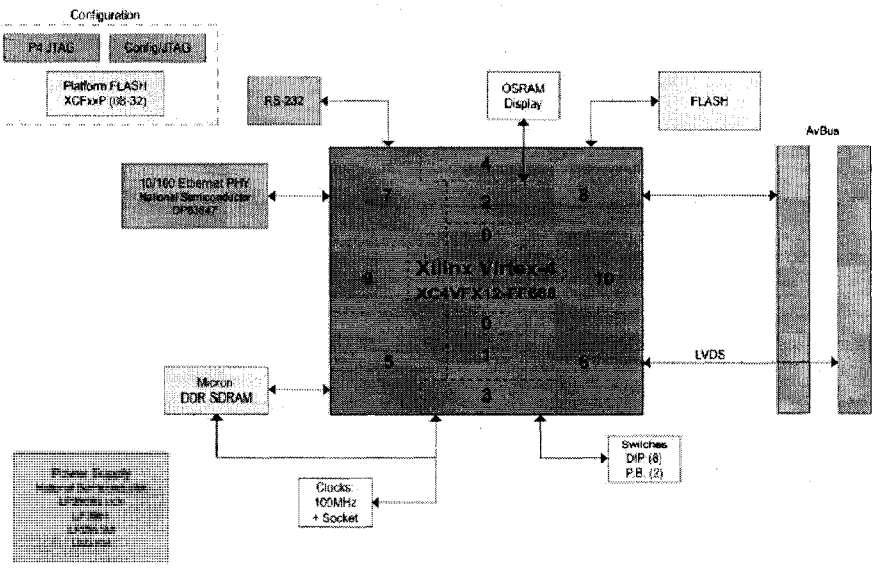

Figure 7 Virtex-4 FX12 Evaluation Board Block Diagram

4.1.3 Virtex -4 FPGA

The FPGA used in our implementation is a Virtex™-4 FPGA produced by Xilinx. Virtex™-4 family FPGAs offer the functionality and performance to address the widest range of demanding applications. It has added enhancements that accelerate productivity by simplifying system design and providing the margin that makes it easy to achieve design targets. The major elements of the Virtex™-4 FPGA are

- Configurable Logic Blocks
- PowerPC® Processor
- Smart RAM

The Virtex™-4 FPGA features arrays of CLBs arranged in columns surrounded on all sides by input/output blocks (IOBs). The CLB is optimized for area and speed for compact high performance design. There are four slices per CLB which implement any combinatorial and sequential circuit and each slice has 4-input look-up tables (LUT), flip-flops, multiplexors, arithmetic logic, carry logic, and dedicated internal routing

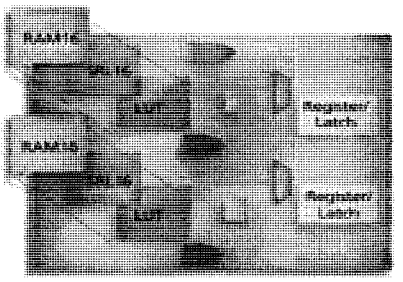

Figure 8 Configuration Logic Block

The Virtex™-4 FPGAs provides up to two PowerPC 405, 32-bit RISC processor cores in a single device. It has flexible system partitioning into hardware and software which supports custom hardware acceleration and co-processing (Control plane processing.)

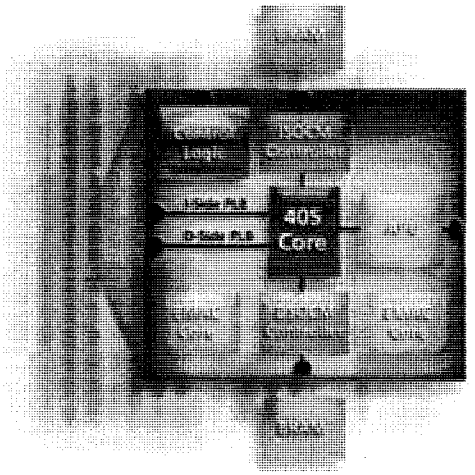

Figure 9 PowerPC Processor

The Virtex™-4 Smart RAM hierarchies not only enables us to achieve compact utilization and highest performance but can also configure any CLB Look-Up Table (LUT) to work as a fast, compact, 16-bit shift register and implement pipeline registers, buffers for video and wireless.

Virtex-4 FPGAs provide up to 960 user I/Os supporting over 20 single-ended and differential electrical I/O standards to enable several parallel system interface standards on one device. New ChipSync™ technology built into every I/O block makes source-synchronous interfacing to the latest high-speed components easy. Plus, powered with XCITE technology, each I/O block deliver on-chip active I/O termination eliminating external termination resistors to increase signal integrity, and save board space, and reduce system cost.

To ensure reliable data transfer between a new generation of high-speed devices, hardware designers are turning to source-synchronous design techniques, in which the component sending the data generates and issues its own clock signal along with the data that it transmits. ChipSync technology simplifies component interface design with critical built-in circuitry that is available in every Virtex-4 I/O. /O termination is required to maintain signal integrity. With hundreds of I/Os and advanced package technologies, external termination resistors are no longer viable. All Virtex-4 I/O structures include third-generation Xilinx Controlled Impedance Technology (XCITE) on-chip active I/O termination. These built-in circuits dynamically eliminate drive strength variation resulting from process, temperature, and voltage fluctuations.

4.1.3 Clocks

The available clock sources on the Virtex-4 FX12 Evaluation board are shown below.

• Single-ended, 100 MHz Oscillator – FPGA pin "AD11"

• 8-pin DIP Clock Socket – FPGA pin "AD12"

The on-board 100 MHz oscillator is used as the clock source for all designs. The 8-pin DIP clock socket allows the user to supply their oscillator of choice.

4.1.3 Memory

The Virtex-4 FX12 Evaluation board is populated with both high-speed RAM and non-volatile ROM to support various types of applications. The board has 32 Megabytes (MB) of DDR SDRAM and 4 MB of FLASH.

XCITE DCI TECHNOLOGY ADVANTAGES	
ADVANTAGE	DETAILS
2nd generation technology	Proven in the field sand used extensively by customers
Lowers cost	Fewer resistors, fewer PCB traces and smaller board area, result in lower PCB costs.
Absolute I/O Flexibility	Any termination on any I/O bank. Non-XCITE technology alternatives deliver limited functionality.
Maximum I/O Bandwidth	Less ringing and reflections maximize I/O bandwidth.

Immunity to temperature and voltage changes	Temperature and voltage variations lead to significant impedance mismatches. XCITE technology dynamically adjusts on-chip impedance to such variations reducing and improving reliability.
Eliminates stub reflection	Improves discrete termination techniques by eliminating the distance between the package pin and resistor.
Increases system reliability	Fewer components on board, deliver higher reliability

Figure 10 XCITE DCI Technology Advantages

4.1.4 DDR SDRAM

Two Micron DDR SDRAM devices make up the 32-bit data bus. Each device provides 16 MB of memory on a single IC and is organized as 2 Megabits x 16 x 4 banks (128 Megabit). The Virtex-4 FX12 Evaluation Board can support larger devices with addressing support for up to 256 MB (two 1-Gigabit devices). The device has an operating voltage of 2.5V and the interface is JEDEC Standard SSTL_2 (Class I for unidirectional signals, Class II for bidirectional signals). The -75 speed grade supports 7.5 ns cycle times with a 2 ½ clock read latency (DDR266B).

4.1.5 Flash Memory

Non-volatile data storage is provided in the form of Flash memory. A single Intel Strata Flash® device makes up the 16-bit data bus. This device provides 4 MB of

memory on a single IC and is organized as 2Megabits x 16 (32 Megabit). The device has an operating voltage of 3.0V

4.1.6 User I/O

Basic user I/O is provided for on the Virtex-4 FX12 Evaluation Board in the form of switches and LED indicators. These peripherals are used to display the compressed bit stream which is the output of the JPEG compression module.

4.1.7 Push Buttons

Two momentary closure push buttons have been installed on the board and attached to the FPGA. These buttons for logic reset and push functions. Pull down resistors hold the signals low (0) until the switch closure pulls it high (1).

Part #	Signal Name	FPGA pin#
SW1 (Pushing compressed bit stream in to led's)	SWITCH_PB1	Y19
SW2	SWITCH_PB2	Y20

Figure 11 Switch Configuration

4.1.8 DIP-switch

An eight-position DIP-switch (SPST) has been installed on the board and attached to the FPGA. These switches provide digital inputs to user logic as shown below. The signals are pulled low (0) by 10K ohm resistors when the switch is open and tied to 3.3V (1) when the switch is closed.

Switch #	Signal Name	FPGA pin#
S1-1 ACDC	SWITCH0	AB24
S1-2 ACDC1	SWITCH1	AB23
S1-3 LAST	SWITCH2	AC25
S1-4 LOAD	SWITCH3	AC24
S1-5 READ_EN	SWITCH4	AD26
S1-6 YCbCr 0	SWITCH5	AD25
S1-7 YCbCr 1	SWITCH6	AC23
S1-8	SWITCH7	G10

Figure 12 Pin Configuration – Switches

4.2 Microcontroller Frame Grabber™ (uCFG™)

The uCFG system block diagram is shown in Figure 3. The uCFG provides 4 separate analog video inputs. All inputs accept either NTSC or PAL composite color video signals depending on the board's configuration settings in non-volatile memory. Black & white (RS-170) video signals are also supported. The uCFG can be setup to operate in high quality S-Video mode where the luma and chroma components are separated out into two discrete signals (Y and C). In this mode, two channels of S-Video are available by using input pairs 1&3 and 2&4 respectively.

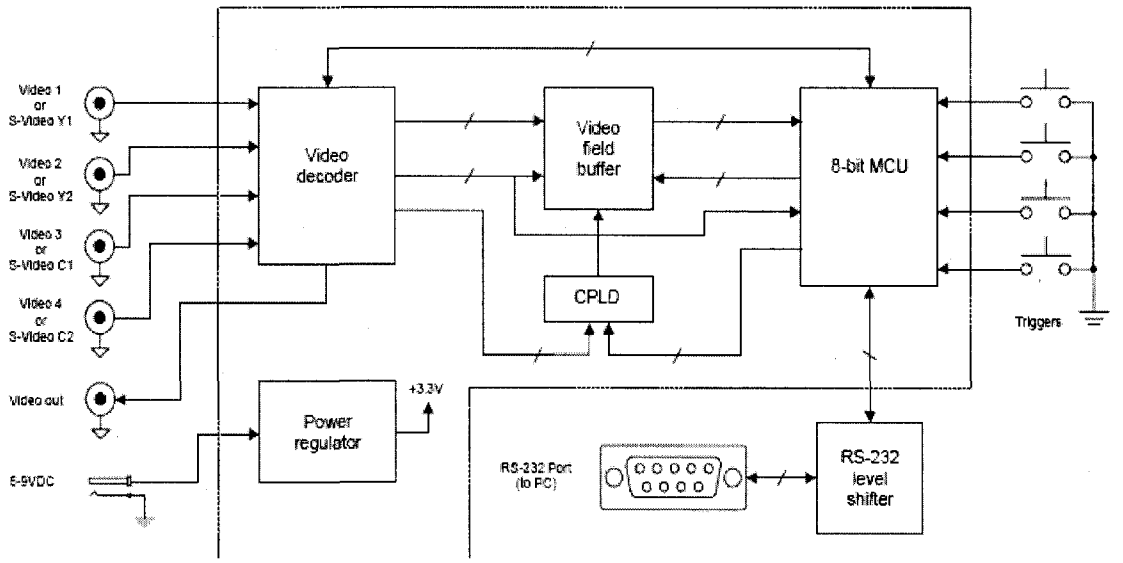

Figure 13 uCFG High Level Diagram

The video inputs are fed to an internal video multiplexer used to select the active video channel for digitization. The output of the analog multiplexer is provided at the uCFG's video out terminal for live video preview. This signal is also fed to an internal video analog-to-digital converter (ADC) and is then converted to an industry standard ITU-BT656 4:2:2 digital video streams. The digital video stream is written to a video field buffer under the control of an 8-bit microcontroller (MCU) as well as a complex programmable logic device (CPLD). Once a single field of color video is stored into the buffer, the 8-bit MCU can read out the data and transmit it through its serial port (LVTTL, +3.3V level). 4 external trigger inputs with programmable polarity are provided to trigger acquisition of a video field from the corresponding video channel. Software triggering is also possible through ASCII commands. For serial interface

compatibility with RS-232 levels, an external level shifter such as the Sipex SP3232 must be added.

4.1.5 FIFO Read Pointer Control

When a video field grab, is commanded either through the software GRAB/TRIG or a hardware trigger, the uCFG board digitizes the very next video field (odd or even depending on request) in the ITU-R BT656 4:2:2 digital video format. This data stream is stored into a long First-In/First-Out (FIFO) memory field buffer. A FIFO is simply a memory storage element which is linearly accessible through a read pointer. This operation essentially freezes the digital data stream for the entire field as-is into the FIFO. Once the capture is complete, the MCU can read out the contents of the FIFO one byte at a time starting from the beginning of the stored digital stream. The linearly accessible FIFO allows the MCU to read out one byte of data at a time and then increments the read pointer to the next data byte in the stored data stream. It is also possible to directly increment the read pointer in the FIFO by an arbitrary amount without reading the data. This is useful to skip some data samples for horizontal decimation, or even skip an entire row's worth of data for vertical decimation. The read pointer can also be reset to point back to the beginning of the stream with the RRST command. The only restriction is that the pointer cannot be decremented directly. Figure below shows the physical layout of the uCFG's internal FIFO buffer. As shown, the FIFO is a byte wide uninterrupted memory buffer accessible with the help of a read pointer (used to index the memory location to read). For simplicity, the FIFO in Figure has been split up into separate video lines even though all the data is actually continuous inside the FIFO. As can be seen, one

line (720 pixels wide) of video data actually occupies 1440 bytes of physical FIFO space. Note that every second luma (Y) sample is surrounded by its chroma (Cb and Cr) color components which are co sited in space (i.e. belong to the same pixel). Every other luma sample is alone (due to the 4:2:2 chroma decimation).

The host controller is in full control of the FIFO access operations and readout. In fact, the host is responsible for issuing the appropriate commands to skip data samples when it requires image decimation. A number of example commands are illustrated in Figure as well as their effect on the read pointer. (a) Different commands increment the read pointer directly. RRST resets the pointer to 0, the start of the FIFO. SEND 111 simply reads the current byte and increments the pointer by 1. RINC with no parameters assumes 1 by default and increments the pointer. RINC with parameters increments the pointer by the specified amount. (b) To download a black & white image only (the luma component of the color image) at the full 720 resolution, simply position the read pointer on the first luma sample of interest with RINC and issue a SEND 2 n 1 where n is the number of bytes to read. Here the command will read and return every 2nd sample until a total of n bytes have been returned on the serial port. To read a ½ decimated version, use the SEND 4 n 1 which skips every 2nd luma (i.e. return every 4th sample). The possibilities are quite numerous. (c) & (d) Illustrates color component download. Simply position the pointer to the appropriate starting position and issue a SEND 4 n 1 to read out every 4th sample.

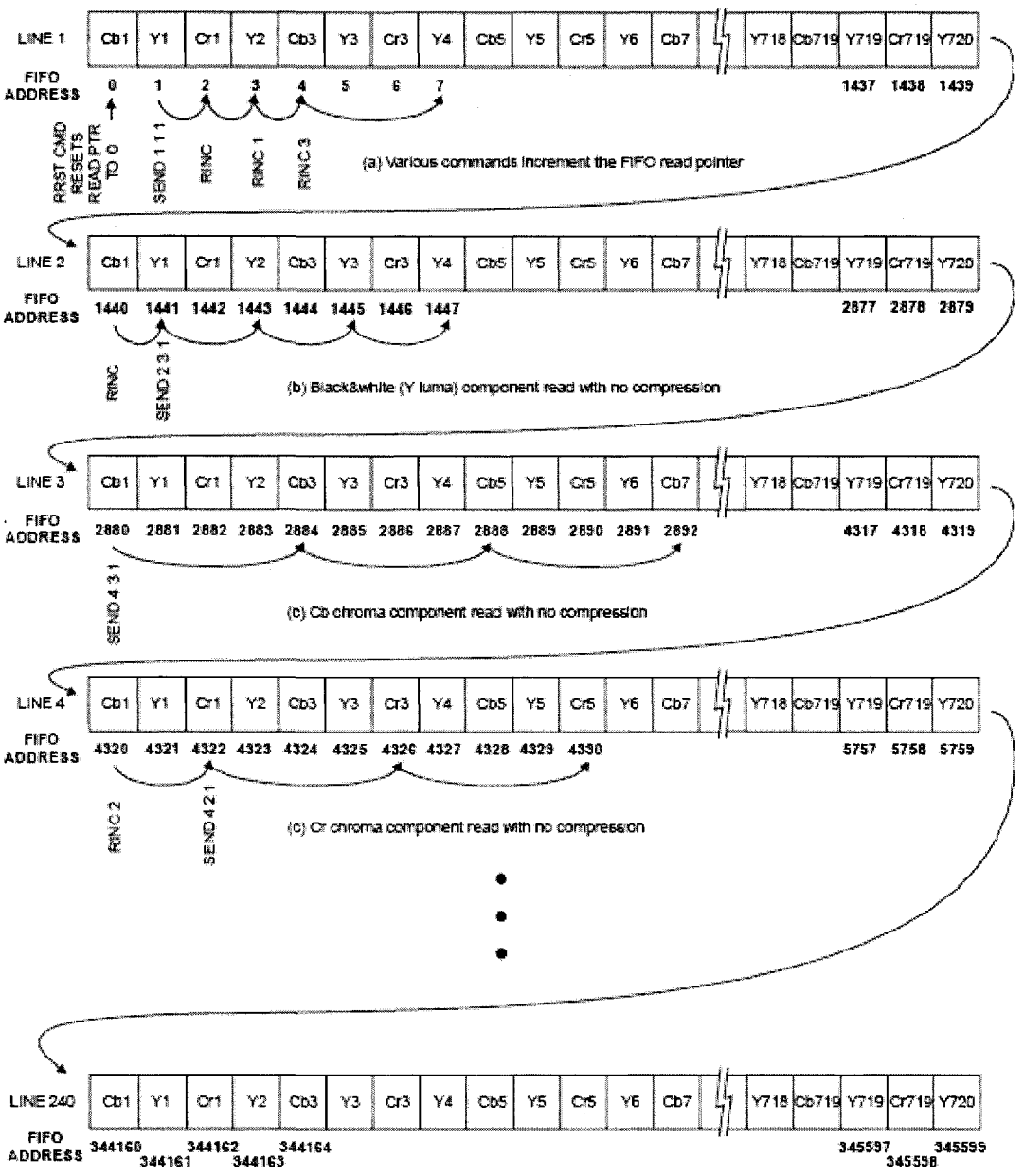

Figure 14 FIFO Read Point Buffer

To read a full color image, first read all the luma values (Y), then issue a read pointer reset command RRST. Then, read all the Cb components and issue another read pointer reset. Finally, read out all the Cr components.

Hopefully, it now becomes clear how powerful the different FIFO commands are. They give full control over the position of the read pointer, the skip factor, and the amount of data to be downloaded. As an example, reading out a sub image (region of interest) would be quite simple to do by directing the read pointer to the correct locations and selectively downloading the data. This feature could be used to perform a quick coarse thumbnail image preview, then a high resolution area of interest download. Another example could be downloading a black and white image only, but with color info only for a small region of interest within the master image...all these techniques reduce the downloaded image data size. To use the compression feature, simply set the 3rd parameter of the SEND command (compression ratio) to 2, 4 or 8 and a compressed image data stream is returned instead of the raw sample values. For example, SEND 2 3 2 would return 3 compressed bytes at 2:1 compression. This means that 3 compressed bytes * 2 compressed pixels / byte = 6 image samples in total skipping every 2nd sample in the FIFO (i.e. Y samples only). The number of bytes returned from this call would be 1 integrator reset byte (for decompression) as header + 3 compressed data bytes = 4 bytes. Simply feed this compressed data vector of 4 bytes into the decompression routine, and the original vector of 6 raw image samples will be regenerated. Please note that the compression is done on-the-fly as data is returned on the serial port. The original raw image data in the FIFO remains unaltered regardless of the way in which the data is read out (i.e. decimated, compressed or not). The only time the data is altered is when the next field grab is commanded (or when the power is removed). It is therefore possible to read

out the data multiple times with different settings. This can be useful to first obtain a thumbnail, followed by a full download.

4.1.5 Grabbing and Downloading an Uncompressed Field

A very common operation to be performed with the uCFG is to grab a field of video and download it. Figure below demonstrates one possible sequence of commands to grab and downloading a full resolution field of NTSC color video (720x240).

First the video channel select command CSEL is issued (this is optional, otherwise the last selected channel will be used). The GRAB command with the odd (O) field parameter is then specified (use E for even field). After execution of the GRAB command, the return value is verified. If an error occurred, then either the selected channel has no valid video signal connected, or the signal is of the wrong standard (PAL/NTSC). Upon a successful field grab, the triggers are temporarily disabled with the TREN 0 command. This will prevent (in the event of a trigger) any new images from overwriting the FIFO data while downloading. Next, three passes are performed through the FIFO. The first pass will read out all the Y luma samples. The second pass will read out the Cb chroma samples, and the third, the Cr chroma samples. Of course, the samples could all be read in one pass, but it is simpler on the host side to split into three passes. As well, to read black & white data only, a single pass would be performed to collect the luma samples.

On the first pass, the FIFO read pointer is reset (RRST). As seen in Figure 9, FIFO address 0 actually points to the Cb1 sample (Cb value of first pixel of line 1). The read pointer therefore needs to be shifted by 1 to point to the first Y1 sample. This is done

with the RINC 1 command. Next, for each line, a SEND 2 720 1 command is issued to send out every 2nd sample in the FIFO from the start position with no compression. This actually reads out all the Y samples of the first line. The process is then repeated for all the 240 lines of the image.

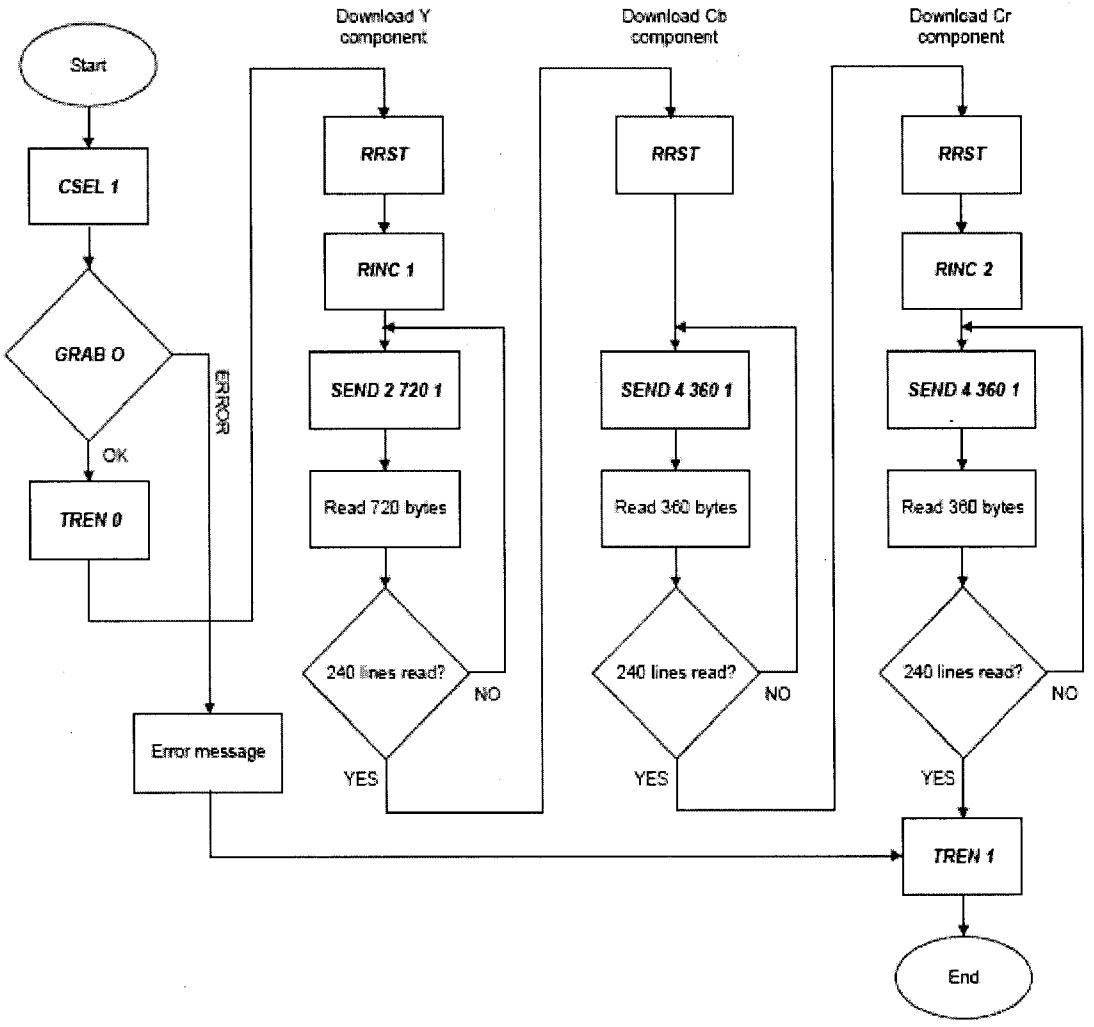

Figure 15 Grabbing Uncompressed Field

In this example, the host computer has enough buffer space to buffer 720 bytes of downloaded data. However, in the case of a resource-limited host (8-bit MCU), it may be desirable to instead download a few bytes at a time only. This is easy to do by changing the parameters of the SEND command. Of course, the increased protocol overhead cost will result in a longer download time. The process is repeated for the 2nd and 3rd pass by first positioning the read pointer on the first Cb or Cr sample and then downloading every 4th data sample for a total of 360 per line (remembering that the chroma information is sub-sampled by a factor 2 in the 4:2:2 standard). Finally, once done the triggers are re-enabled with TREN 1.

4.1.5 Downloading a Decimated Field

Figure below shows a possible sequence of command to download an uncompressed color decimated field of half the full NTSC resolution (360x120). As before, most of the steps are the same. The main differences are that the SEND command now skips every 4 samples (i.e. every 2nd Y) only. This results in a horizontally decimated image by a factor 2. The second difference is that following the download of the decimated data for a full line, a RINC 1440 command is introduced. This command effectively skips an entire row (or line) of raw video data in the FIFO. This vertically decimates the number of lines in the downloaded field by a factor 2.

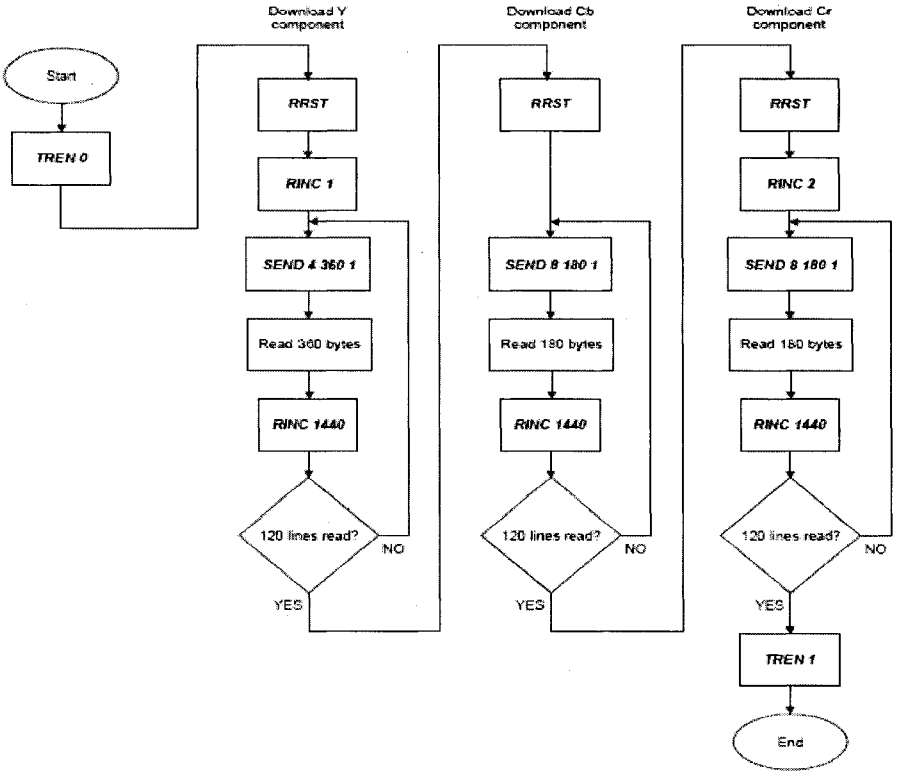

Figure 16 Downloading Decimated Field

4.2.4 Converting 4:2:2 YCbCr to RGB for PC Display

The ultimate goal of the image downloading is to display the images to the user on a PC monitor. To perform this operation correctly, a few things must be explained. First off, PC monitors work in the RGB color space. This means that every pixel has an 8 bit red, green and a blue component. The downloaded data from the uCFG is in the YCbCr color space and, as well, the color components are decimated by a factor 2 (4:2:2). Before displaying an image on a PC, this data must therefore be up sampled and converted to the RGB domain.

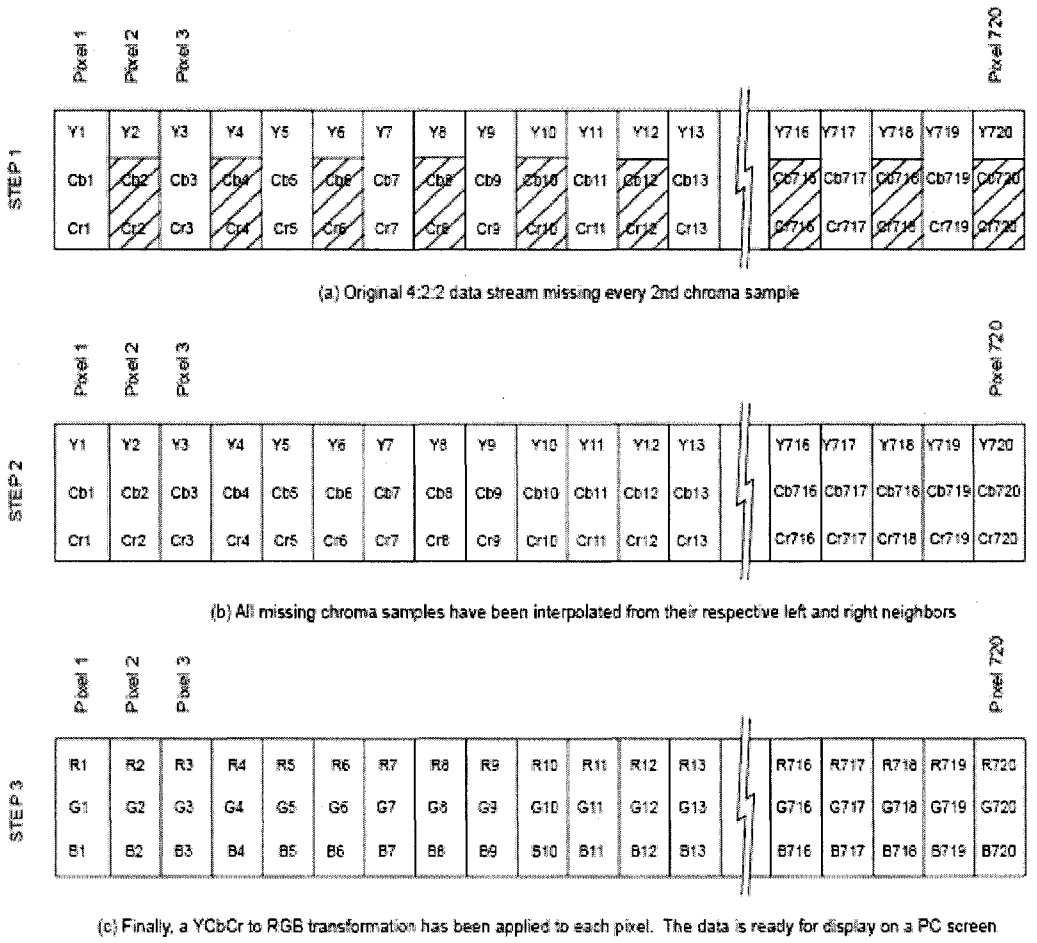

Figure 17 YCbCr to RGB Conversion

Figure illustrates the different steps required to convert a 4:2:2 YCbCr data stream into a non-decimated RGB stream suitable for display on the PC monitor. Step 1 shows a single line of downloaded 4:2:2 YCbCr color data samples shown overlaid over a row of the 720 pixels forming the image line. It can be seen that some of the decimated chroma samples are missing.

In Step 2, all the missing chroma values are filled in with a simple linear interpolation performed by taking the average of the previous and next respective samples. For example, Cr2 = (Cr1 + Cr3)/2. After this step, each pixel has a Y, a Cb, and a Cr value. Step 3 performs a color space conversion from YCbCr to RGB. Each pixel's YCbCr values a fed through equation 1 and the resulting RGB output vector is obtained, ready for display on a PC monitor.

4.1.9 RS-232 Interface

An RS-232 Interface is used for communication between the FPGA board and the Microcontroller Frame Grabber (uCFG).

An RS-232 interface has the following characteristics:

- Uses a 9 pins connector "DB-9" (older PCs use 25 pins "DB-25").
- Allows bidirectional full-duplex communication (the PC can send and receive data at the same time).
- Can communicate at a maximum speed of roughly 10KBytes/s.

It has 9 pins, but the 3 important ones are:

- pin 2: RxD (receive data).
- pin 3: TxD (transmit data).
- pin 5: GND (ground).

Using just 3 wires, you can send and receive data.

4.1.9.1 Serial communication

Data is sent one bit at a time; one wire is used for each direction. Since computers usually need at least several bits of data, the data is "serialized" before being sent. Data is

commonly sent by chunks of 8 bits. The LSB (data bit 0) is sent first, the MSB (bit 7) last.

4.1.9.2 Asynchronous communication

This interface uses an "asynchronous" protocol. That means that no clock signal is transmitted along the data. The receiver has to have a way to "time" it to the incoming data bits.

In the case of RS-232, that's done this way:

- Both side of the cable agree in advance on the communication parameters (speed, format...). That's done manually before communication starts.
- The transmitter sends a "1" when and as long as the line is idle.
- The transmitter sends a "start" (a "0") before each byte transmitted, so that the receiver can figure out that data is coming.
- After the "start", data comes in the agreed speed and format, so the receiver can interpret it.
- The transmitter sends a "stop" (a "1") after each data byte.

The speed is specified in baud, i.e. how many bits-per-seconds can be sent. For example, 1000 bauds would mean 1000 bits-per-seconds, or that each bit lasts one millisecond. Common implementations of the RS-232 interface (like the one used in PCs) don't allow just any speed to be used. If you want to use 123456 bauds, you're out of luck. You have to settle to some "standard" speed. Common values are:

- 1200 bauds.
- 9600 bauds.
- 38400 bauds.
- 115200 bauds (usually the fastest you can go).

At 115200 bauds, each bit lasts (1/115200) = 8.7µs. If you transmit 8-bits data, that lasts 8 x 8.7µs = 69µs. But each byte requires an extra start and stop bit, so you actually need 10 x 8.7µs = 87µs. That translates to a maximum speed of 11.5KBytes per second. At 115200 bauds, some PCs with buggy chips require a "long" stop bit (1.5 or 2 bits long...) which make the maximum speed drop to around 10.5KBytes per second.

4.1.9.3 Physical layer

The signals on the wires use a positive/negative voltage scheme.

- "1" is sent using -10V (or between -5V and -15V).
- "0" is sent using +10V (or between 5V and 15V).a

So an idle line carries something like -10V.

A VHDL UART is used to communicate between the FPGA and the FIFO Buffer of the Frame Grabber.

4.1.10 VHDL-UART

The VHDL-UART used to transfer bits from the FIFO buffer of the uCFG to the ROM of the FPGA is given below.

4.1.10.1 UART-Receiver

```
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_arith.all;
entity UARTReceiver is
        generic
        (
                frequency               : integer;
                baud                    : integer;
                oversampling            : integer
        );
        (
                clk                     : in std_logic;
                rxd                     : in std_logic;
                rxd_data                : out std_logic_vector(7 downto 0);
                rxd_data_ready                  : out std_logic
        );
end entity UARTReceiver;
architecture UARTReceiverArch of UARTReceiver is
-- defining constant
                constant BIT_SPACE          : integer := 10;  -- 8 to 11 are common
                constant DIVISOR    : integer := 1600;
                constant FREQ_INC   : integer := (oversampling + 1) * baud / DIVISOR;
                constant FREQ_DIV   : integer := frequency / DIVISOR;
                constant FREQ_MAX : integer := FREQ_DIV + FREQ_INC - 1;
```

```
-- defining types
type state_type is (idle, bit0, bit1, bit2, bit3, bit4, bit5, bit6, bit7, stop);
-- defining signals
signal state            : state_type := idle;    -- receiver's state
signal rxd_sync_inv     : std_logic_vector(1 downto 0);
signal rxd_cnt_inv      : std_logic_vector(1 downto 0);
signal rxd_bit_inv      : std_logic;
signal baud_divider     : integer range 0 to FREQ_MAX := 0;
signal data             : std_logic_vector(7 downto 0);
signal baudover_tick    : std_logic := '0';
signal bit_spacing      : integer range 0 to 15;
signal next_bit         : std_logic := '0';
begin
        -- assignments
        next_bit <= '1' when bit_spacing = BIT_SPACE else '0';
        -- processes
        baud_gen : process(clk)
        begin
                if clk'event and clk = '1' then
                        baud_divider <= baud_divider + FREQ_INC;
                        if baud_divider >= FREQ_DIV then
                                baud_divider <= 0;
```

```
                    baudover_tick <= '1';
                else
                    baudover_tick <= '0';
                end if;
            end if;
        end process baud_gen;
rxd_sync_inverted : process(clk) -- inverted to suppress phantom character
        begin
            if clk'event and clk = '1' then
                if baudover_tick = '1' then
                    rxd_sync_inv <= rxd_sync_inv(0) & not rxd;
                end if;
            end if;
        end process rxd_sync_inverted;
        --
        rxd_counter_inverted : process(clk)
        begin
            if clk'event and clk = '1' then
                if baudover_tick = '1' then
                    if rxd_sync_inv(1) = '1' and rxd_cnt_inv /= "11" then
                        rxd_cnt_inv <= unsigned(rxd_cnt_inv) + 1;
                    elsif rxd_sync_inv(1) = '0' and rxd_cnt_inv /= "00" then
```

```
                    rxd_cnt_inv <= unsigned(rxd_cnt_inv) - 1;
                end if;
                if rxd_cnt_inv = "00" then
                rxd_bit_inv <= '0';
                elsif rxd_cnt_inv = "11" then
                    rxd_bit_inv <= '1';
                end if;
            end if;
        end if;
    end process rxd_counter_inverted;
state_proc : process(clk)
    begin
        if clk'event and clk = '1' then
            if baudover_tick = '1' then
                case state is
                    when idle =>
                        if rxd_bit_inv = '1' then
                            state <= bit0;
                        end if;
                    when bit0 =>
                        if next_bit = '1' then
                            state <= bit1;
```

```
            end if;
        when bit1 =>
            if next_bit = '1' then
                state <= bit2;
            end if;
        when bit2 =>
            if next_bit = '1' then
                state <= bit3;
            end if;
        when bit3 =>
            if next_bit = '1' then
                state <= bit4;
            end if;
        when bit4 =>
            if next_bit = '1' then
                state <= bit5;
            end if;
        when bit5 =>
            if next_bit = '1' then
                state <= bit6;
            end if;
        when bit6 =>
```

```
                        if next_bit = '1' then
                            state <= bit7;
                        end if;
                    when bit7 =>
                        if next_bit = '1' then
                            state <= stop;
                        end if;
                    when stop =>
                        if next_bit = '1' then
                            state <= idle;
                        end if;
                end case;
            end if;
        end if;
end process state_proc;
        bit_spacing_proc : process(clk)
begin
        if clk'event and clk = '1' then
            if state = idle then
                bit_spacing <= 0;
            elsif baudover_tick = '1' then
                if bit_spacing < 15 then
```

```
                    bit_spacing <= bit_spacing + 1;
                else
                    bit_spacing <= 8;
                end if;
            end if;
        end if;
    end process bit_spacing_proc;
    shift_data_proc : process(clk)
    begin
        if clk'event and clk = '1' then
            if baudover_tick = '1' and next_bit = '1' and
            state /= idle and state /= stop then
                data <= not rxd_bit_inv & data(7 downto 1);
            end if;
        end if;
    end process shift_data_proc;
    --
    output_data_proc : process(clk)
    begin
        if clk'event and clk = '1' then
            if baudover_tick = '1' and next_bit = '1' and
            state = stop and rxd_bit_inv = '0' then
```

```
                    rxd_data <= data;
                    rxd_data_ready <= '1';
                else
                    rxd_data_ready <= '0';
                end if;
            end if;
        end process output_data_proc;
end UARTReceiverArch;
```

4.2.6.2 UART-Transmitter

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

entity UARTTransmitter is
    generic
    (
        frequency       : integer;
        baud            : integer
    );
    port
    (
        clk             : in std_logic;
```

```
        txd             : out std_logic;
        txd_data        : in std_logic_vector(7 downto 0);
        txd_start       : in std_logic;
        txd_busy        : out std_logic
    );
end entity UARTTransmitter;
architecture UARTTransmitterArch of UARTTransmitter is
-- defining types
type state_type is (idle, start, bit0, bit1, bit2, bit3, bit4, bit5, bit6, bit7, stop1, stop2);
-- defining signals
signal state        : state_type := idle; -- transmitter's state
signal data         : std_logic_vector(7 downto 0);
signal baud_tick    : std_logic;
signal busy         : std_logic := '0';
signal baud_divider : integer range 0 to (frequency/100 + baud/100 - 1) := 0;
begin
    -- assignments
    txd_busy <= busy; busy <= '0' when state = idle else '1';
    -- processes
    baud_gen : process(clk)
    begin
        if clk'event and clk = '1' then
```

```
            if busy = '1' then
                baud_divider <= baud_divider + (baud/100);
                if baud_divider > (frequency/100) then
                    baud_tick <= '1';
                    baud_divider <= 0;
                else
                    baud_tick <= '0';
                end if;
            end if;
        end if;
end process baud_gen;
--
state_proc : process(clk)
begin
        if clk'event and clk = '1' then
            case state is
                when idle =>
                    if txd_start = '1' then
                        state <= start;
                    end if;
                when start =>
                    if baud_tick = '1' then
```

```
                state <= bit0;
            end if;
        when bit0 =>
            if baud_tick = '1' then
                state <= bit1;
            end if;
        when bit1 =>
            if baud_tick = '1' then
                state <= bit2;
            end if;
        when bit2 =>
            if baud_tick = '1' then
                state <= bit3;
            end if;
        when bit3 =>
            if baud_tick = '1' then
                state <= bit4;
            end if;
        when bit4 =>
            if baud_tick = '1' then
                state <= bit5;
            end if;
```

```
                when bit5 =>
                        if baud_tick = '1' then
                                state <= bit6;
                        end if;
                when bit6 =>
                        if baud_tick = '1' then
                                state <= bit7;
                        end if;
                when bit7 =>
                        if baud_tick = '1' then
                                state <= stop1;
                        end if;
                when stop1 =>
                        if baud_tick = '1' then
                                state <= stop2;
                        end if;
                when stop2 =>
                        if baud_tick = '1' then
                                state <= idle;
                        end if;
        end case;
end if;
```

```
end process state_proc;
--
data_load_proc : process(clk)
begin
        if clk'event and clk = '1' then
                if txd_start = '1' then
                        data <= txd_data;
                end if;
        end if;
end process data_load_proc;
        txd_proc : process(clk)
begin
        if clk'event and clk = '1' then
                case state is
                        when idle => txd <= '1';
                        when start => txd <= '0';
                        when bit0 => txd <= data(0);
                        when bit1 => txd <= data(1);
                        when bit2 => txd <= data(2);
                        when bit3 => txd <= data(3);
                        when bit4 => txd <= data(4);
                        when bit5 => txd <= data(5);
```

```
                    when bit6 => txd <= data(6);
                    when bit7 => txd <= data(7);
                    when stop1 => txd <= '1';
                    when stop2 => txd <= '1';
                end case;
            end if;
        end process txd_proc;
--      busy_proc : process(clk)
--      begin
--          if clk'event and clk = '1' then
--              if state = idle then
--                  busy <= '0'; txd_busy <= '0';
--              else
--                  busy <= '1'; txd_busy <= '1';
--              end if;
--          end if;
--      end process busy_proc;
end UARTTransmitterArch;
```

The other alternative approach is to use MATLAB to capture the frames directly from the frame grabber and to convert the information into RGB values so that the VHDL code for Jpeg compression could use those vales to give the compressed bit stream. The compressed bit stream is displayed using the 8 led's on the Vitex-4 FPGA board. The list

of inputs is as mentioned earlier. Since the output is a compressed bit stream of 32 bits and we have only 8 led's on the board we use SW1 as a push button to push all the bits serially seven bits at a time into the led's.

CHAPTER 5

RESULTS

The process was implemented on a Vrtex-4 FPGA board. An image containing (8x8) pixels was segregated into RGB and given as input. The implementation results are as follows

5.1 JPEG Compression

Map Report

Target Device	:	xc4vfx12
Target Package	:	ff668
Target Speed	:	-10
Mapper Version	:	virtex4 -- $Revision: 1.34 $
Mapped Date	:	Sat Jul 12 16:05:03 2008

Design Summary

Number of warnings	:	4

Logic Utilization:

Total Number Slice Registers	:	3,265 out of 10,944 29%
Number used as Flip Flops	:	3,241
Number used as Latches	:	24
Number of 4 input LUTs	:	3,272 out of 10,944 29%

Logic Distribution:

Number of occupied Slices	:	2,984 out of 5,472 54%
Number of Slices containing only related logic	:	2,984 out of 2,984 100%
Number of Slices containing unrelated logic	:	0 out of 2,984 0%

*See Notes below for an explanation of the effects of unrelated logic

Total Number 4 input LUTs	:	3,323 out of 10,944 30%
Number used as logic	:	3,272
Number used as a route-thru	:	50
Number used as Shift registers	:	1
Number of bonded IOBs	:	42 out of 320 13%
Number of BUFG/BUFGCTRLs	:	2 out of 32 6%
Number used as BUFGs	:	2
Number used as BUFGCTRLs	:	0
Number of DSP48s	:	4 out of 32 12%
Total equivalent gate count for design	:	52,278
Additional JTAG gate count for IOBs	:	2,016

Notes:

Related logic is defined as being logic that shares connectivity - e.g. two LUTs are "related" if they share common inputs. When assembling slices, Map gives priority to combine logic that is related. Doing so, results in the best timing performance. Unrelated logic shares no connectivity. Map will only begin packing unrelated logic into a slice once 99% of the slices are occupied through related logic packing.

Note that once logic distribution reaches the 99% level through related logic packing, this does not mean the device is completely utilized. Unrelated logic packing will then begin, continuing until all usable LUTs and FFs are occupied. Depending on your timing budget, increased levels of unrelated logic packing may adversely affect the overall timing performance of your design.

Delay Summary Report

The Number Of Signals Not Completely Routed For This Design Is	:	0
The Average Connection Delay For This Design Is	:	1.279
The Maximum Pin Delay Is	:	4.318
The Average Connection Delay On The 10 Worst Nets Is	:	3.580

Listing Pin Delays by value: (nsec)

5.2 Motion Estimation

Map Report

Target Device	:	xc4vfx20
Target Package	:	ff672
Target Speed	:	-12
Mapper Version	:	virtex4 -- $Revision: 1.34 $
Mapped Date	:	Sat Jul 12 13:02:03 2008

Design Summary

Number of Slices	:	5313 out of 8544	62%
Number of Slice Flip Flops	:	6378 out of 17088	37%
Number of 4 input LUTs	:	10162 out of 17088	59%

Number used as logic	:	10034	
Number as Shift registers	:	128	
Number of IOs	:	24	
Number of FIFO16/RAMB16s	:	2 out of 68	2%
Number used as RAMB16s	:	2	
Number of GCLKs	:	1 out of 32	3%

CHAPTER 6

CONCLUSION AND FUTURE RECOMMENDATIONS

This paper presented the implementation of JPEG compression and motion estimation on Virtex-4 FPGA hardware. The modules of the jpeg architecture were designed and synthesized. The control block hardware design is also completed. The detailed pipeline design, operators, and the final results of the synthesis of the modules were also presented, resulting in an architecture containing 3,272 logic cells, including the control block with device utilization of 29% and average timing delay of 9.821 ns. The designed architecture performs the JPEG compression of a 640 x 480 pixels gray level image in 23.8ms, allowing its use in a JPEG compressor in hardware.

In future this implementation can be extended to Microblaze technology. The MicroBlaze soft processor includes several configurable interfaces that allow us to connect our own custom peripherals and coprocessors, as well as Xilinx provided peripherals. The MicroBlaze Debug Module (MDM) allows debugging of eight MicroBlazes at a time. An automated partitioning system is under development. The aim is to provide an automated system which can take advantage of processor parameterization with custom instructions, variable width registers, and multiple execution units, as well as assigning operations to hardware or software. FPGA-based codesigns allow a very large design space to be explored, and the opportunity to provide

a very high communication bandwidth between the processors and the hardware will mean that co-design solutions have a good chance of producing more efficient designs.

BIBLIOGRAPHY

1. A Technical Introduction to Digital Video, C. Poynton. New York: Wiley, 1996.
2. "Digital color imaging IEEE Trans. Image Processing, vol. 6, pp. 901–932, July 1997", by G. Sharma and H. Trussell.
3. "The JPEG Still Picture Compression Standard", by Gregory K. Wallace, Multimedia Engineering, Digital Equipment Corporation, Maynard, Massachusetts page 1 jpeg comp.
4. "A System for the Implementation of Image Processing Algorithms On Configurable Computing Hardware" by Benjamin Alexander Levine
5. Wong, S.; Vassiliadis, S.; Cotofana, S., "A sum of absolute differences implementation in FPGA hardware," Euromicro Conference, 2002. Proceedings. 28th, vol., no., pp. 183-188, 2002
6. "Distributions of the Two-Dimensional DCT coefficients for Images", IEEE Trans. Commun, vol. 31, pp. 835-839, 1983, by J. D. Gibson and R. C. Reininger.
7. "JPEG, Still Image Data Compression Standard" Van Nostrand Reinhold, 1993.
8. "Effects of Quantization Table Manipulation on JPEG Compression of Cervical adiographs" by L. E. Berman, R. Long, S. R. Pillemer Society for Information Display International Symposium May 18-20, 1993.
9. http://www.cs.cf.ac.uk/Dave/Multimedia/node259.html.

10. Weisstein, Eric W. "Huffman Coding." From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/HuffmanCoding.html

11. D. Buell, J. Arnold, and W. Kleinfelder, Splash 2: FPGAs in a Custom Computing Machine. Los Alamitos, CA: IEEE Computer Society Press, 1996.

12. "Programmable Active memories: Reconfigurable Systems Come of Age," IEEE Trans. On VLSI Systems, vol. 4, no. 1, pp. 56-69, March 1996 by J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard.

13. "JPEG Compression History Estimation for Color Images", Ramesh Neelamani, Ricardo de Queiroz, Zhigang Fan, Sanjeeb Dash, and Richard G. Baraniuk

14. http://www.eee.bham.ac.uk/WoolleySI

15. "Integrated Digital Architecture for JPEG Image Compression" by Luciano Agostini and Sergio Bampi.

16. "Image and video comoression standards – Second Edition, Kluwer Academic Publishers, USA, 1999 by vasudev bhaskaran, Konstantinos Konstantinides.

17. The International Telegraph and Telephone Consultative Committee (CCITT), "Information Technology – Digital Compression and Coding of Continuous-Tone Still Images – Requirements and Guidelines". Rec. T.81, 1992.

18. V. Bhaskaran, K. Konstantinides. Image and Video Compression Standards Algorithms and Architectures – Second Edition, Kluwer Academic Publishers, USA, 1999.

19. J.M. Saul. Hardware/Software Codesign for FPGA-Based Systems. In proceedings of the 32nd Hawaii International Conference on System Sciences – 1999

20. http://www.xilinx.com/support/documentation/application_notes/xapp610.pdf.

VITA

Graduate College
University of Nevada Las Vegas

Ramakrishna Gopalakrishnan

Address:
1555 E Rochelle Ave Apt 268
Las Vegas, NV 89119

Degree:

- Bachelor of Engineering, Electronics and Communication Engineering, 2006
  Anna University, India

Thesis Title:

Implementation of JPEG Compression and Motion Estimation on FPGA Hardware

Thesis Examination Committee:

Chairperson, Dr. Henry Selvaraj, Ph.D.
Committee Member, Dr. Emma Regentova, Ph.D.
Committee Member, Dr. Muthukumar Venkatesan, Ph.D.
Graduate College Representative, Dr. Laxmi Gewali, Ph.D.