Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

Exploring the Implementation of JPEG Compression on FPGA

A Thesis presented in partial fulfilment of the requirements for the degree of

Masters of Engineering
in
Electronics and Computer Systems Engineering

at Massey University, Palmerston North
New Zealand

Ann Malsha De Silva
2012

SUPERVISORS

Dr Amal Punchihewa

Associate Prof. Donald Bailey

Abstract

This thesis presents an implementation of JPEG compression on a Field Programmable Gate Array (FPGA) as the data are streamed from a camera. The goal was to minimise the usage of logic resources of the FPGA and the latency at each stage of the JPEG compression. The modules of these architectures are fully pipelined to enable continuous operation on streamed data. The designed architectures are detailed in this thesis and they were described in Handel-C. The correctness of each JPEG module implemented on Handel-C was validated using MATLAB.

The software and hardware based algorithms did result in small differences in the compressed images as a result of simplifying the arithmetic in hardware. However, these differences were small, with no discernible difference in image quality between hardware and software compressed images.

The JPEG compression algorithm has been successfully implemented and tested on Altera DE2-115 development board. Improvements were made by minimising the latency, and increasing the performance. Final implementation also showed that implementing a quantisation module in three stage pipeline fashion and using FPGA multipliers for 1D-DCT and 2D-DCT can significantly drop the logic resources and increase the performance speed. The proposed JPEG compressor architecture has a latency of 114 clock cycles plus 7 image rows and has a maximum clock speed of 55.77MHz. The results obtained from this implementation were very satisfactory.

Acknowledgements

I would like to take this opportunity to thank some people who made my research possible. First of all I'd like to express my gratitude towards my supervisors, Dr. Amal Punchihewa and A/P Donald Bailey for giving me guidance, support, encouragement, understanding and patience throughout this research as they have broadened my views of the various technologies available in my work field and provided me with skills I can use in the field later in my career.

I also like to acknowledge the receipt of a Master of Engineering Scholarship from Massey University and Graduate Women Manawatu Postgraduate Scholarship from Graduate Women Manawatu Charitable Trust Inc.

Finally I would like to thank my husband, my parents, sister and my friends for their support and encouragement throughout this experience.

Thank you all!

Table of Contents

List of Figures

List of Tables

1

Introduction

This chapter gives a basic background to this research. It states the problem to be studied, aim and main objectives. The proposed methodology for implementing JPEG compression into FPGA is presented.

1.1 Background

In image processing, image compression can improve the performance of the system by reducing the cost and time in image storage and transmission without a significant reduction of the image quality. A monochrome image can be defined over a matrix of picture elements (pixels), with each pixel represented by 8-bit grey scale value. This representation of image data could demand large storage and bandwidth to transmit. The purpose of image compression is to reduce the size of the representation and, at the same time to preserve most of the information contained in the original image [1]. Image compression can be lossy or lossless. Lossy compression gives a greater reduction in data volume compared to lossless compression; however only an approximation to the original image can be reconstructed.

There are several standards for image compression and decompression (CODEC) such as Joint Photographic Experts Group (JPEG) [2], JPEG2000 [3], Graphic Interchange Format (GIF) [4], Portable Network Graphics (PNG) [5]. JPEG compression is the most widely used form of lossy image compression. It's based on the Discrete Cosine Transform (DCT). A compressed image in JPEG format can be 10% of the original size depending on the information contained within the image and compression quality, which would mean that a 90% decrease in the needed bandwidth [6]. Image and video codecs are implemented mainly in software as signal processors can manage operations without incurring too much overhead in computation. These operations can also be efficiently implemented in hardware [7].

Field Programmable Gate Arrays (FPGAs) are a relatively new technology, which combines the properties of the traditional hardware and software alternatives. They can provide speed, performance and flexibility since they implement a parallel and pipelined version of the algorithm [8]. The latest FPGAs have millions of reconfigurable gates, capable of running at clock speeds of hundreds of megahertz (MHz) and therefore these devices are well-suited for graphics and image processing. FPGA based designs generally comprise a large number of simple processors which all work in parallel and may compete for memory access or other resources [9].

1.2 Problem Description and Motivation

Processing time and power restrictions imposed on dedicated embedded systems make software compression unviable in many applications. Power efficiency and fast compression are often performance critical factors. For most digital image codecs, increasing the compression has been achieved at the cost of increasing the complexity of the techniques and implementations. These restrictions usually mandate a dedicated hardware implementation of a JPEG compressor, especially in applications such as digital cameras, DVD players, traffic controllers, secure ticketing, and many more. As the JPEG compression process is complex, its design in hardware is demanding [10]. These restrictions have motivated this work.

FPGAs are well suited for many embedded systems applications because they have several desirable attributes such as, small size, low power consumption, a large number of I/O ports, and a large number of computational logic blocks [11]. Images have a high degree of spatial parallelism, thus image processing applications are ideally suited to implementation on FPGAs which contain large arrays of parallel logic and registers and can support pipelined algorithms [9].

JPEG is an international standard for still-image compression and it has been widely used since 1987 [2]. This research is concerned about the implementation of JPEG compression for grey-scale images in to FPGA in real time as the images are streamed from the camera. The real time JPEG FPGA is expected to run at 50MHz.

There are many research papers published in conference proceedings and journal papers about JPEG compression using FPGAs [7, 10, 12, 13]. Unfortunately, as a result of page limits and space constraints, many of these papers give the results of the implementation of various sections of the JPEG compression algorithm, but present relatively few design details. Some researchers only focus on implementing either 2D-DCT or Huffman coding onto FPGAs [1, 14-17].

This research focuses on implementing JPEG compression by minimising the usage of logic resources available and minimising the latency at each module of the JPEG

compression. The JPEG algorithm was chosen for this research as it is well established and highly recognisable. The development of design architectures for each module of JPEG is described in later chapters. The modules of these architectures are fully pipelined and target to FPGA device implementation using Handel-C. Each JPEG module was evaluated using MATLAB.

1.3 Aim of the Thesis

The main aim of this thesis was to explore the implementation of JPEG compression on to FPGA device as the data are streamed from the camera while minimising:

- the requirement of logic resources of the FPGA and,
- the latency at each stage of the JPEG compression.

1.4 Scope and Constraints

The scope of this thesis was limited to JPEG compression only, since it is well defined and highly recognisable. This thesis only focuses on grey-scale. It can readily be extended to colour images as the processing on the chrominance components is the same as for the luminance.

JPEG implementation into FPGA presents a number of problems and constraints:

- **Real-time constraints:** The input image is streamed from the camera. This imposes a strict time constraint that depends on the capture frame rate and the image size. Stream processing demands calculations for one pixel value at each clock cycle (given a serial input stream).

- **Memory bandwidth constraints:** In stream processing, memory bandwidth constraints dictate that as much processing as possible be performed on the data as they arrive. Small on-chip memory blocks can be accessed in parallel increasing available bandwidth for temporary storage.

1.5 Objectives

The objectives of this thesis were:

1. Identify the optimum number of bits that need to maximise fidelity while minimising the use of resources and latency at each stage of JPEG compression.
2. Simulate the proposed system architecture using the hardware description language, Handel-C.
3. Compile the program using Quartus II program and port the code to FPGA.
4. Test the JPEG compression on FPGA as data are streaming from a camera.
5. Evaluate hardware implementation and compare the results using MATLAB.
6. Optimise the results obtained.

1.6 Proposed Methodology

The proposed workflow for implementing JPEG compression onto an FPGA is shown in Figure 1.1. The basic idea of the methodology is adopted from Figure 4.3 of [11]. First, a fixed point JPEG algorithm has been developed in MATLAB. This serves two purposes. First, it identifies the minimum number of bits required to represent each stage within the FPGA without introducing significant error. This is important, because an FPGA implementation is not restricted to work with the standard 8, 16, or 32 bit-word lengths used by software. The speed can be increased, and resources required can be reduced by minimising the number of bits. Second, it also provides ground truth data for bench-testing the resulting FPGA algorithm.

Then, most of the effort has gone into mapping the algorithm into a form suitable for FPGA implementation. The aim was to make the implementation as resource and time efficient as possible. The resulting algorithm was implemented using the hardware description language, Handel-C [18]. Two software suites were used during the hardware implementation: Mentor Graphics' DK design suite, and Altera's Quartus II design suite. Each stage of the algorithm was validated through simulation by comparing the results with the MATLAB results.

Finally the implementation was targeted to a Cyclone IV FPGA on an Altera DE2-115 development board.

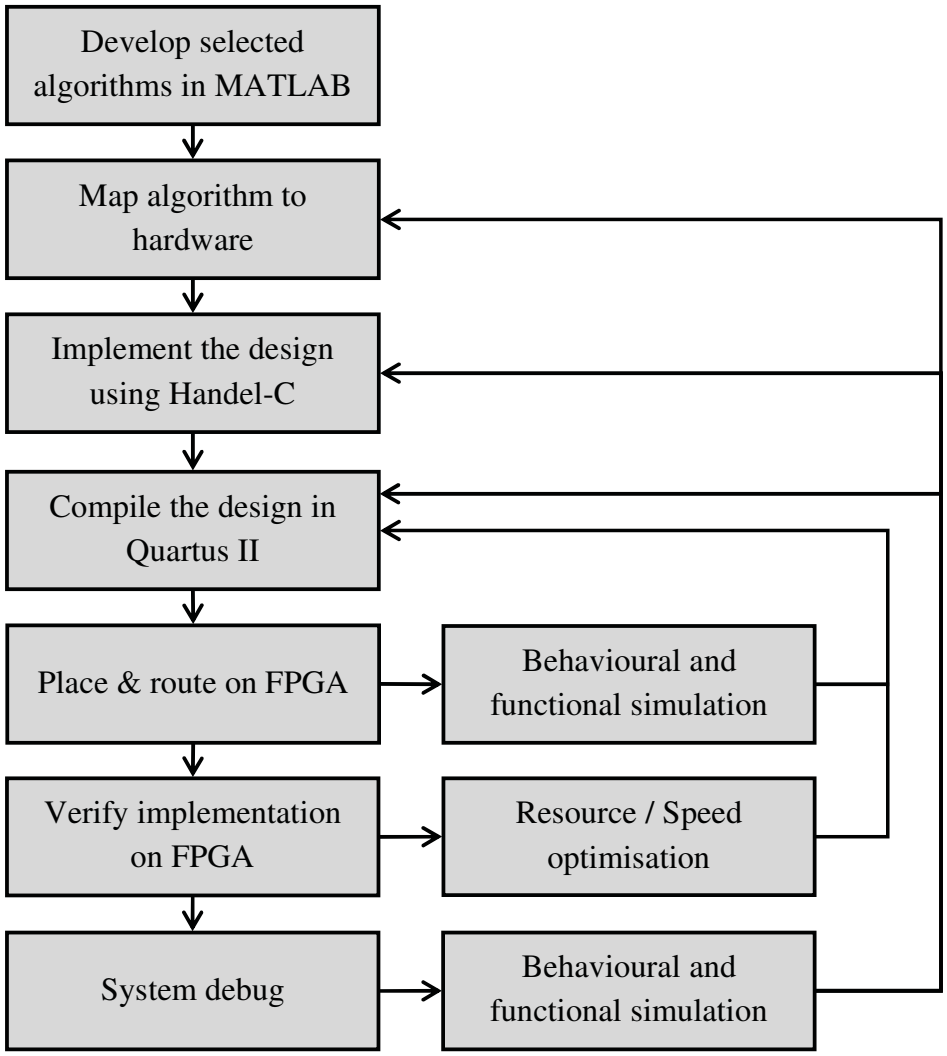


Figure 1.1: Proposed implementation of JPEG compression on FPGA adopted from [11].

1.7 Thesis Structure

This thesis explores the implementation of JPEG compression into an FPGA device as the data are streaming from the camera. The contents of the thesis are structured as follows.

Chapter 1: Introduction

This chapter introduces several important concepts that are necessary to understand the content of this thesis. This gives a basic background on image compression and states the problem to be studied. The proposed methodology for implementing JPEG compression into FPGA is presented.

Chapter 2: Introduction to FPGAs

This chapter describes the basic architecture of FPGA and presents the advantages and disadvantages of FPGAs. This chapter also gives details on the hardware and software development environments.

Chapter 3: Introduction to JPEG Compression

This chapter gives an introduction to JPEG compression and presents the basic architecture of JPEG compression. It also presents a basic outline of each block of JPEG compression and lists the advantages and disadvantages of image compression. The previous work on FPGA implementation of JPEG compression is presented in some detail.

Chapter 4: Implementation of Block Processing and Discrete Cosine Transform

This chapter investigates the implementation of a low complexity and more elegant pipelined DCT architecture for JPEG compression. This chapter presents the basic background to DCT transform. Then it presents the proposed architecture and compares it with alternative approaches. It also analyses the fixed point word length optimization. Finally, it discusses the results obtained through the proposed DCT architecture.

Chapter 5: Implementation of Quantisation and Zigzag Coding

This chapter gives the basic overview of zigzag coding and quantisation. It discusses the hardware implementation; and presents the results after zigzag coding and quantisation obtained through the proposed approach.

Chapter 6: Implementation of Entropy Coding

Background of entropy coding is presented in this chapter. It presents the detailed implementation and discusses the proposed method of Huffman coding and results obtained through this approach.

Chapter 7: Implementation of JPEG Header

This chapter gives the details of JPEG headers and their hardware implementation.

Chapter 8: Simulation Results and Discussion

This presents the overall system architecture and gives the details of the obtained results from the Handel-C simulation and gives the details of testing and validation methods. It also compares the results with MATLAB compression.

Chapter 9: Final Implementation

This describes the mapping of the Handel-C into the DE2-115 board and presents the initial and optimisation results.

Chapter 10: Overall Conclusion

This final chapter presents the overall conclusion of this research and suggests possible future work.

2

Field Programmable Gate Arrays

This chapter describes the basic architecture of FPGAs. It presents the advantages and disadvantages of using FPGAs in applications. It also gives details on the hardware and software development environments used in this research.

2.1 Background

FPGAs are semiconductor devices that can be programmed after manufacturing. Instead of being restricted to any predetermined hardware function, an FPGA allows you to program product features and functions, adapt to new standards, and reconfigure hardware for specific applications even after the product has been installed in the field, hence the name "field-programmable".

Generally, we either implement computations in hardware such as custom VLSI, application specific integrated circuits (ASIC), gate arrays or in software running processors like DSPs, microcontrollers, embedded microprocessors. However, FPGAs combine the speed of hardware with the flexibility of software programming. It has brought about something of a revolution in hardware design. Machines based on FPGAs have achieved impressive performance [12, 19].

Xilinx was the first to introduce FPGAs in 1985. Xilinx and Altera are the current FPGA market leaders and long-time industry rivals. Together, they control over 80% of the market [20]. Other competitors include Lattice Semiconductor, Actel, SiliconBlue Technologies, Achronix, and QuickLogic.

Commercially there are four main classes of FPGAs available: symmetrical array, row-based, hierarchical PLD, and sea-of-gates. In all of these FPGAs the interconnections and how they are programmed vary. Currently there are seven technologies in use: static RAM cells, anti-fuse, PROM, EPROM transistors, EEPROM transistors, and flash and fuse [21].

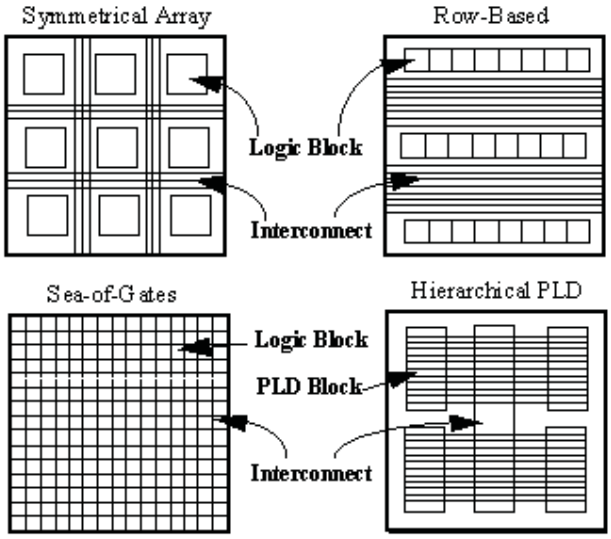


Figure 2.1: Category of different FPGA [21].

2.2 FPGA Architecture

Figure 2.2 shows the basic FPGA architecture and it has three major configurable elements: configurable logic blocks, input/output blocks, and interconnectors. The configurable logic blocks are arranged in a two-dimensional array, and the interconnection wires are organised as horizontal and vertical routing channels between rows and columns of logic blocks. The routing channels contain wires and programmable switches that allow the logic blocks to be interconnected in many ways. Each logic gate is connected by user defined routing to each other or input/outputs, which are the FPGA's connection to the exterior world.

The FPGA products on the market feature different types of logic blocks. The most commonly used logic block is a lookup table (LUT), which contains storage cells that are used to implement a small logic function. Each cell is capable of producing a single logic value, either 0 or 1 as a function of 3 to 6 input bits (depending on the device). When a circuit is implemented in an FPGA, the logic blocks are programmed to realise the necessary functions and the routing channels are programmed to make the required interconnections between logic blocks.

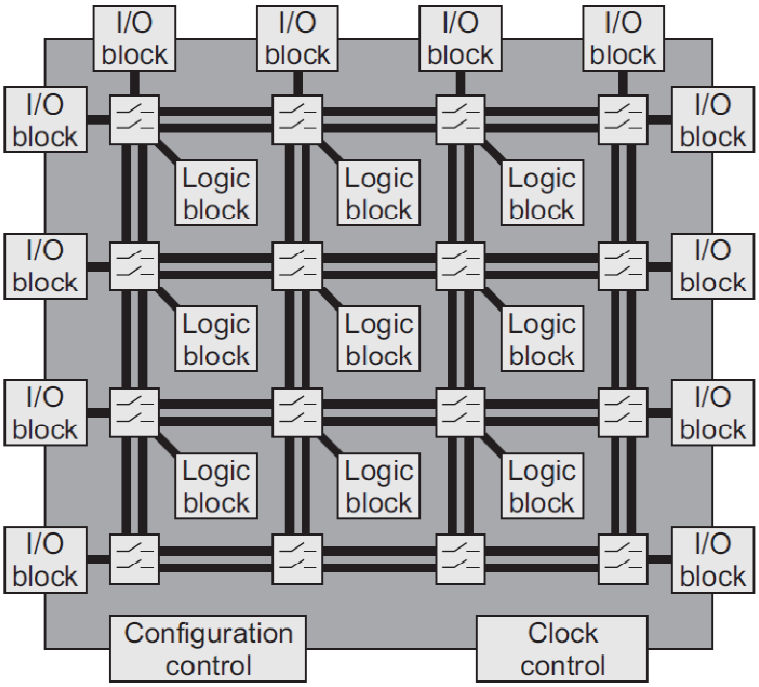


Figure 2.2: Basic FPGA architecture [11].

Complex circuitry can be mapped onto FPGA devices without the requirement for any expensive machinery or manual effort [22]. In the early days FPGAs only had relatively small number of gates, so they were only used as a bridge or flexible interconnect between other parts of a hardware design. However, now there are FPGA chips with many millions of gates, enabling entire complex systems to be implemented using reconfigurable logic alone in a single IC. Depending on the particular device, the program is either burned in permanently or semi-permanently, or is loaded from an external memory each time the device is powered up.

There are many different FPGAs with different processes. However the basic structure consists of a semi-regular matrix of logic units. Each unit is one of programmable logic devices (PLDs), logic gates, RAM blocks or several other types of component.

2.3 FPGA Applications

Since FPGA implements the logic required for an application by building separate hardware for each function, FPGAs are inherently parallel. This gives them the speed

that results from a hardware design while retaining the reprogrammable flexibility of software at a relatively low cost. These advantages have made FPGAs very popular for image processing [11], digital signal processing [23], ASIC prototype development, custom computing [24], software-defined radio [25], aerospace and defence systems [26], medical imaging [27], speech recognition [28], cryptography [29], bioinformatics [30], computer hardware emulation, radio astronomy [31] and a growing range of other areas.

FPGAs are also widely used for system validation including pre-silicon validation, post-silicon validation, and firmware development. This allows chip companies to validate their design before the chip is produced in the factory, reducing the time to market.

2.4 Advantages and Disadvantages of using FPGAs

Compared to software implementations on DSPs or microcontrollers and discrete hardware implementation on VLSI, FPGAs have following advantages and disadvantages.

Advantages:

1. High processing speed comparing to software implementation.
2. Costs of components can be reduced.
3. FPGAs enable rapid prototyping through reprogramming the hardware.
4. Long product life cycle through the ability to change the program to mitigate the risk of obsolescence.
5. The ability to re-programme in the field while debugging.
6. Shorter time to market.

Disadvantages:

1. Parallel programming is harder to implement complex algorithms compared to sequential programming.

2. In the case of single applications, FPGAs are more expensive than microcontrollers.

3. Limited library availability, i.e. technology is dependent on FPGA type, vendor and the hardware platform.

2.5 Hardware Development

All of the hardware design and implementation for this work was performed on a Windows 7 workstation equipped with a 3.6GHz Intel core processor and 4GB of RAM. The workstation uses an Altera DE2-115 development board to implement the hardware designs.

2.5.1 Altera DE2-115 board

A photograph of DE2-115 development board is shown in the Figure 2.3. It represents the layout of the board and indicates the location of the connectors and key components.

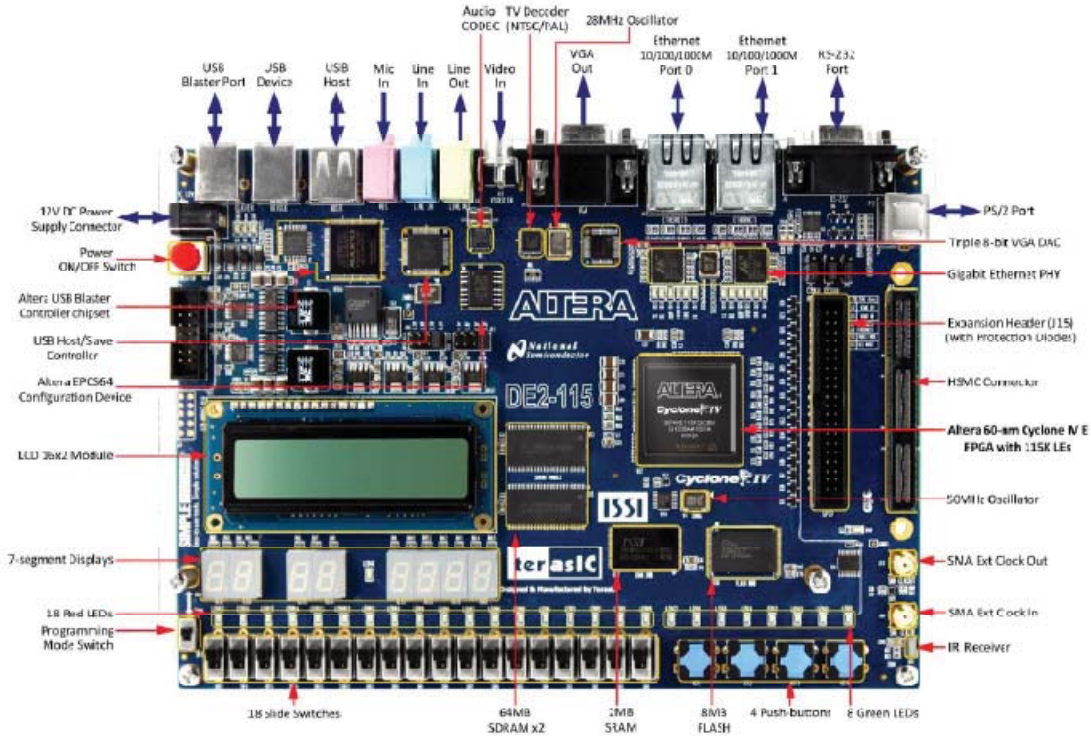


Figure 2.3: The DE2-115 board [32].

The ALTERA DE2-115 board has number of devices that can be used to implement wide range of projects. According to ALTERA [32] the following hardware is provided on the DE2-115 board:

- Altera Cyclone® IV 4CE115 FPGA device
- Altera Serial Configuration device – EPCS64
- USB Blaster (on board) for programming; both JTAG and Active Serial (AS) programming modes are supported
- 2MB SRAM
- Two 64MB SDRAM
- 8MB Flash memory
- SD Card socket
- 4 Push-buttons
- 18 Slide switches
- 18 Red user LEDs
- 9 Green user LEDs
- 50MHz oscillator for clock sources
- 24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jacks
- VGA DAC (8-bit high-speed triple DACs) with VGA-out connector
- TV Decoder (NTSC/PAL/SECAM) and TV-in connector
- 2 Gigabit Ethernet PHY with RJ45 connectors
- USB Host/Slave Controller with USB type A and type B connectors
- RS-232 transceiver and 9-pin connector
- PS/2 mouse/keyboard connector
- IR Receiver
- 2 SMA connectors for external clock input/output
- One 40-pin Expansion Header with diode protection
- One High Speed Mezzanine Card (HSMC) connector
- 16x2 LCD module

The Cyclone IV FPGA device is the centrepiece of the board and is the reconfigurable logic that user can target. So to provide the maximum flexibility for the user, all connections are made through the Cyclone IV FPGA device. Figure 2.4 gives the block diagram of the DE2-115 board.

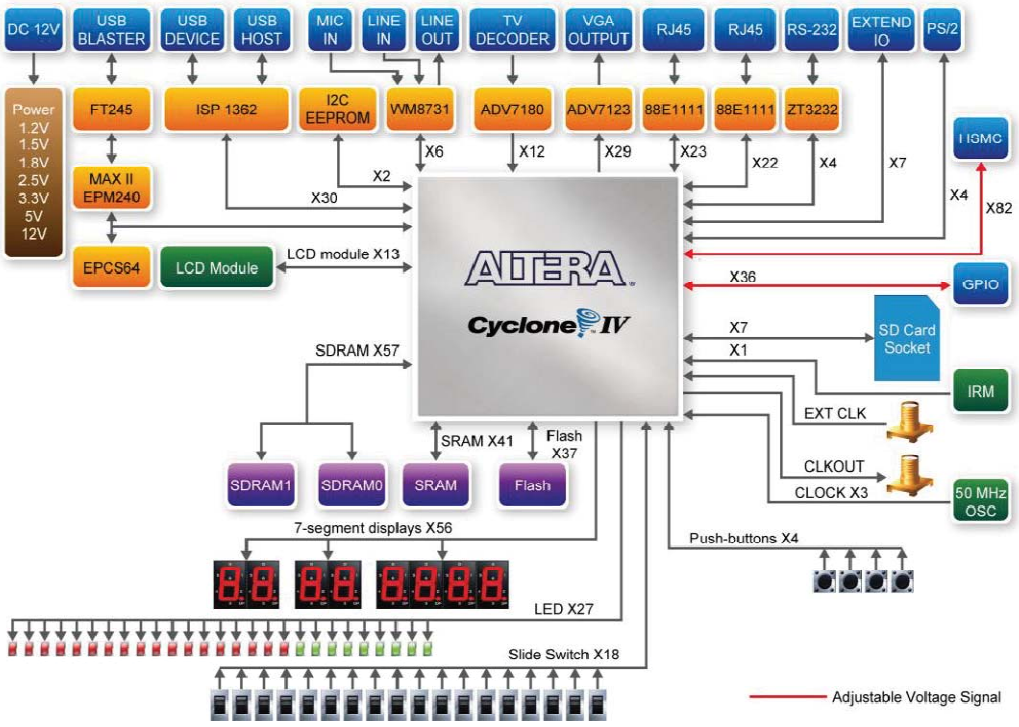


Figure 2.4: Block diagram of DE2-115 [32].

In order to use the DE2-115 board, we have to be familiar with the Quartus II software.

2.5.1.1 Motivation of using DE2-115

This was used mainly because of the availability and the low cost and it does what it required to do.

2.6 Software Development

Two software suites were used during software development: Mentor Graphics DK design suite, and Altera's Quartus II design suite.

2.6.1 DK design suite

The DK development suite supports the development of hardware designs in Handel-C. It provides a software flow for compilation of algorithms onto FPGA boards. Handel-C enables FPGA programming using software-like tools and flows. Users can verify systems in cycle-accurate simulations, and compile code directly into FPGA logic, creating configuration files to program FPGA boards. This enables rapid system implementation from software to FPGA hardware platforms.

2.6.1.1 Handel-C

Handel-C is a high-level hardware description language that allows user control of sequential and parallel processing, and also operator size and widths [19]. Compile time support is provided by designing various interfaces which enable the Handel-C program to communicate with the different parts of the FPGA board, such as the onboard memory.

Handel-C was originally developed by Celoxica, which was formed out of the University of Oxford in 1996 to commercialise its research into Handel-C. Handel-C is a C based language with additional language features for specifying parallel processes, hardware connections and clock functions [18]. It aims at compiling high-level algorithms directly to synchronous hardware [33]. Because standard C is a sequential language, Handel-C has additional constructs to support the parallelisation of code and to allow fine control over what hardware is generated.

Handel-C can be used to design sequential programs but to gain speed improvement parallel constructs need to be used. Handel-C generates the required logic gates from the source code; however it works at the register transfer level. This means that each assignment is clocked into a register after calculation. Consequently, in Handel-C each assignment is performed in a single clock cycle.

The main language extension in Handel-C is the par statement. This enables several statements to be run in parallel. There is also the inclusion of I/O pin constructs, port,

and channel construct to enable communication between external interfaces and parallel process respectively. Channels also allow parallel processes to synchronise with each other. Data types such as the signal have been added which act like wires in a hardware design. There are also extensions for bit manipulations including bit selection and concatenation of variables.

By using fixed point numbers in Handel-C the bit length of each step can be defined to be different. This gives a great deal of flexibility in design and can save on the amount of hardware used for registered outputs, as smaller registers can be constructed.

2.6.1.2 Electronic Design Interchange Format (EDIF)

The Electronic Design Interchange Format (EDIF) is a standardised representation of circuit netlist data, which is independent of specific manufacturers and is designed to allow the transfer of information between incompatible systems.

In this research, EDIF acts as a link between Handel-C and DE2-115 board. This is because the Handel-C compiler is able to generate EDIF files, which can be compiled using Quartus II to give the FPGA configuration file.

2.6.1.3 Motivation for using Handel-C

The usual languages for programming FPGAs are Verilog or VHDL. As mentioned earlier, Handel-C is very much like software programming. Unlike VHDL, when designing, the user does not need to worry about low-level decisions. At the algorithmic level, which Handel-C is in, it is much faster and more convenient to describe the systems desired behaviour. For prototyping and for most situations, Handel-C is the preferred language. But for time critical applications, the user may decide to program using VHDL instead. It provides an EDA tool stream targeting FPGA technology and leverages existing knowledge of C programming and algorithms.

2.6.2 Altera's Quartus II design suite

Altera's Quartus II design software is used in conjunction with the Altera DE2-115 development board. It compiles the EDIF netlist from Handel-C into a configuration file for programming the FPGA. It maps the design to particular resources on the FPGA and checks that the design meets the timing constraints.

2.6.2.1 SRAM Objective File (.sof)

This is a binary file with the extension .sof, generated by the Quartus II Compiler's Assembler module. This file contains data for configuring all SRAM-based Altera devices supported by the Quartus II software, using the Programmer.

Figure 2.5 shows the steps involved in producing a .sof file from a Handel-C program. The resulting file can be loaded directly onto the DE2-115 board.

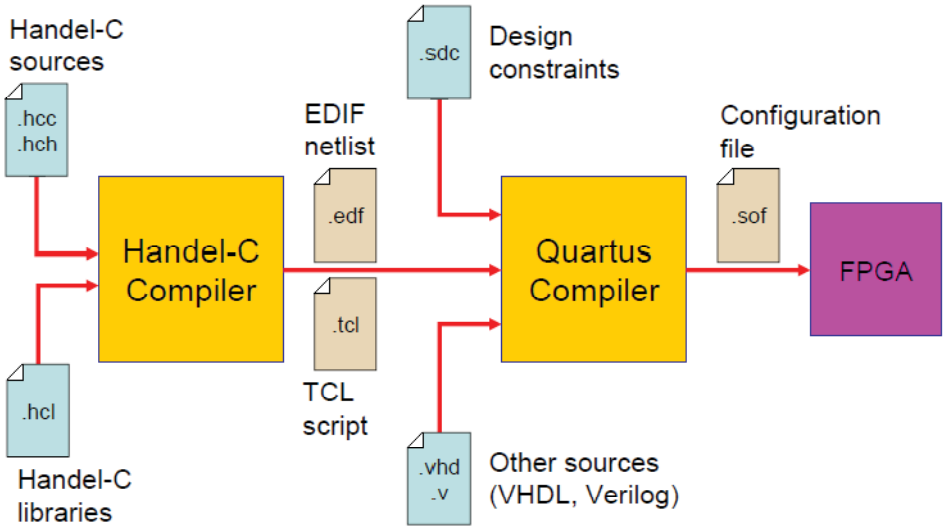


Figure 2.5: Steps of mapping Handel-C on FPGA [34].

2.6.2.2 Motivation of Using Quartus II design suite

Quartus II software delivers the highest productivity and performance for Altera FPGAs. It delivers superior synthesis and placement and routing, resulting in compilation time advantages. Compilation time reduction features include:

- Multiprocessor support
- Rapid Recompile
- Incremental compilation

2.7 Summary

This chapter has presented an overview of FPGAs. The design was targeted to a Cyclone IV FPGA on Altera DE2-115 development board. Two software suites were used during software development: Mentor Graphics DK design suite, and Altera's Quartus II design suite. Handel-C was adopted as the high-level language for implementing the JPEG compression system. Altera's Quartus II design software was used in conjunction with the Altera DE2-115 development board.

3

Image Compression

This chapter gives an introduction to image compression and outlines its advantages and disadvantages. It describes the basic architecture of JPEG compression, giving an outline of each step in the JPEG compression algorithm. Previous FPGA implementations on JPEG compression by others are presented in some detail.

3.1 Background

Image compression is used to reduce data by exploiting irrelevance and redundancy of the image data in order to be able to store or transmit data in an efficient form. Image compression can be lossless or lossy. In image compression, lossless compression is where an identical source image can be reconstructed from the compressed data of the original image. Lossy compression is a method where the decompressed image is not identical to original image but instead is reasonably close to it [35].

The principles of image compression algorithms are to:

- reduce the redundancy in the image data and,
- produce a reconstructed image from the original image while allowing some error that is not significant to intended application.

The goal is to achieve a more efficient representation of image data while preserving the essential information contained in the image.

3.2 Advantages and Disadvantages of Image Compression

Advantages:

1. It reduces the data storage requirements.
2. The rate of input-output operations in a computing device can be greatly increased due to the shorter representation of data.
3. Image compression reduces the cost of backup and recovery of data in computer systems by storing the backup of large files in compressed form.

Disadvantages:

1. Image compression generally reduces the reliability of the records because the reduction in redundancy leads to increased sensitivity to errors.

2. Transmission of a compressed image through a noisy communication channel is risky because the burst errors introduced by the noisy channel can destroy the transmitted image.

3. Disruption of image properties of a compressed image will result in the compressed image being different from the original data.

4. In many hardware and system implementations, the extra complexity added by image compression can increase the system's cost and reduce the system's efficiency.

There are several standards for image compression and decompression (CODEC). However this research is concerned only on the implementation of JPEG compression for grey-scale images using a FPGA.

3.3 JPEG Compression

The Joint Photographic Experts Group proposed the JPEG compression standard [2] in 1987 and, since then, this is the most used lossy compression for still photographic images. The baseline JPEG compression algorithm is the most basic form of sequential DCT based compression. This technique can produce very good compression ratios, at the expense of some sacrifice in image quality. By using transform coding, quantisation, and entropy coding at 8-bit pixel resolution, a high-level compression can be achieved.

There are losses of information in the baseline JPEG compression. Losses are due to the quantisation operation. These losses can be controlled to have an almost imperceptible influence to the human visual system [36]. One of the advantages of JPEG is the use of many parameters, allowing the user to adjust the amount of data

lost and thus also the compression ratio over a very wide range. The four basic steps commonly used in JPEG compression are shown in Figure 3.1.

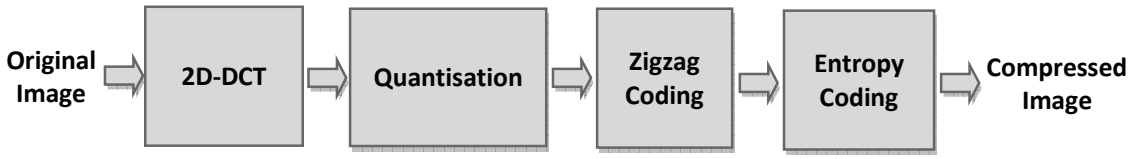


Figure 3.1: Basic architecture of JPEG compression

Each pixel in an original image is assumed to represent a value between 0 and 255. The unsigned pixel data of the original image are divided into 8×8 blocks and these blocks are processed sequentially from left to right and top to bottom.

The 2D discrete cosine transform (DCT) is applied to each block to create a 8×8block of frequency components. The DCT transforms a picture from the spatial domain into the frequency domain. The upper-left corner in each 8×8 block of DCT coefficients is the DC coefficient and the other 63 values are AC coefficients. They represent the average pixel value and successive higher-frequency changes within the block. When a block is processed by the DCT, the high-frequency coefficients appear at the lower-right corner of the block while low-frequency coefficients appear at the upper-left corner. It has been shown that the DCT is close to optimal at reducing the correlation between coefficients, and therefore concentrating the energy into a few significant coefficients.

Then each of the 64 frequency components in a block are divided by a separate quantisation coefficient and then rounded to an integer. The quantiser step size is determined by the acceptable visual quality of image. After quantisation, the 64 quantised coefficients are converted into a one-dimensional sequence by the zigzag operation. The quantised coefficients are arranged in increasing frequency order. This is because the energy is usually concentrated into the low frequency coefficients enabling the insignificant high frequency coefficients to be efficiently compressed by run length encoding or truncation.

Finally, the coefficients are encoded by Huffman coding. Its idea is to use fewer bits to represent a symbol which appears more frequently and more bits to represent a symbol which appears less often. Differential coding is applied to the DC component prior to entropy coding, where the AC components are directly entropy coded [14]. An end-of-block (EOB) mark is inserted at the end of each block. In this way, each block of 8×8 pixel values is turned into a smaller block of codewords and the effect of compression is thus achieved.

3.4 Compression Ratio and PSNR

Benchmarks in image compression are the compression ratio and peak signal to noise ratio (PSNR). The compression ratio is used to measure the ability of data compression by comparing the size of the compressed image to the size of the original image. A greater compression ratio means better compression.

PSNR is one of the parameters that can be used to quantify image quality. PSNR is often used to benchmark the level of similarity between the reconstructed image and the original image. A larger PSNR corresponds to better image quality.

$$PSNR = 10log_{10}\left(\frac{MAX_i^2}{MSE}\right), \tag{3.1}$$

where MAX_i is the maximum possible pixel value of the image and MSE is the Mean Square Error of the compressed image.

3.5 Related Work

The research reported in this thesis has been profoundly influenced by the work of numerous researchers. Many of these researchers give the results of the implementation of various sections of the JPEG compression algorithm, but present relatively few design details [6, 7, 10, 12, 13].

Haralick [37] showed that 2D-DCT computation can be implemented as a sequence of two 1D-DCTs which is commonly referred to as the separability property. Due to the

wide spectrum of applications in which DCT is used, several researchers have worked on this topic resulting in a vast amount of literature [14-17, 38-40].

Agostini et al. [16] have implemented a fast pipelined 2D-DCT for JPEG image compression. This implementation has a latency of 48 clock cycles because they divided the algorithm into six blocks to share hardware. The structure of the DCT is less regular than the FFT, making an elegant pipeline less practical.

Kovac and Ranganathan [13] described a fully pipelined single chip VLSI architecture for JPEG compression. The architecture exploits the principles of pipelining and parallelism to the maximum extent in order to obtain high speed and throughput. However they have only given details of their DCT and Huffman coding implementations. The DCT circuit has a latency of 59 clock cycles and 5 multipliers at the expense of each coefficient having a different scale factor.

Sun and Lee has proposed a JPEG chip for image compression and decompression in 2002 [12]. Their system was partitioned and fit into two FLEX 10K FPGAs, an EPF10K100 and an EPF10K70. Placement, routing, and programming of the FPGAs were done by ALTERA Maxplus II. The main limitation with this design is that the maximum working frequency is 27MHz and they have achieved this at the complexity of 411, 745 transistors and 23.264K-bit of memory. Their chip also has a power consumption of 1W.

Agostini et al. [10] presented a soft IP design of a high performance FPGA based JPEG compressor. This approach incurs in a minimum latency of 238 clock cycles. The multiplications were converted to shift-add operations, reducing the use of hardware resources and increasing the compressor performance. This has been mapped to Altera FLEX 10KE FPGAs, and it maintains a processing rate of 39.8MHz.

The fact that there are not any research work exist in the literature that describes the complete architecture for implementing the JPEG compression on FPGAs was the initial motivation for this research. From the information available it was clear that we

can achieve better speeds by carefully designing a pipelined parallel architecture. Such architecture is advantageous in that higher clock speeds can be easily obtained by decreasing the granularity of processing in each stage.

This thesis presents a fully pipelined JPEG compression architecture for FPGA while minimising the latency at each stage of the JPEG standard and minimising logic resources. The proposed architecture for DCT is adopted from [11] and it is a simpler and elegant pipelined design which is based on a first order factorisation by Woods et al. [40]. This work has also investigated the data-width required at each stage of the DCT process. The zigzag and the initial Huffman coding architectures for this research were adapted from [11] and then modified to use the synchronous memory block on the FPGA. The quantisation architecture described in thesis was implemented using a signed by unsigned non-restoring divider which was proposed in [41].

4

Implementation of Block Processing and Discrete Cosine Transform

This chapter describes the transformation from streamed input data to the 8×8 blocks required by the DCT. It also presents implementation of low complexity pipelined DCT architecture for JPEG compression and compares this with alternative approaches. It also determines the optimal fixed point word length for the DCT. Finally it discusses the results obtained through the proposed DCT architecture and fixed point implementation.

4.1 Block Processing

In many image processing applications, computations are defined on very long streams of input data. Certain image processing operations involve processing an image in blocks, rather than processing the entire image at once. With block processing the image is divided into rectangular blocks, and the operation is performed on each individual block to determine the values of the pixels in the corresponding block of the output image.

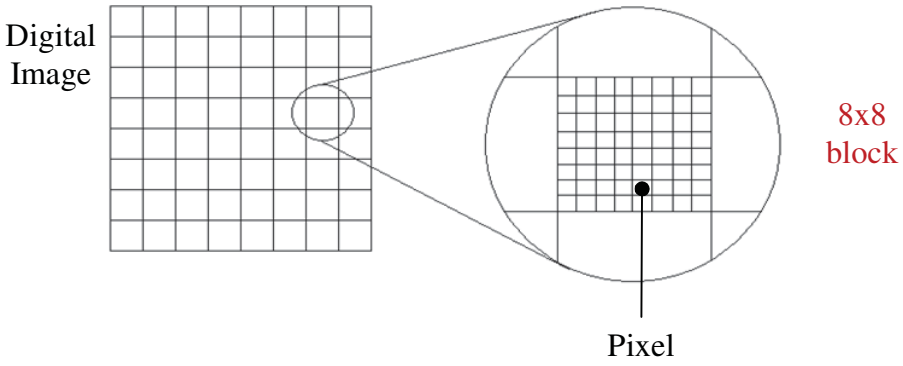


Figure 4.1: Digital image blocks

4.2 Transformation of Streamed Data to Blocks

JPEG uses block processing to maintain locality of the data to give good compression (nearby pixel values are highly correlated). As the data are streamed whole rows at a time, it is necessary to buffer 8 rows of the image before an 8×8 block of data is available for processing.

To achieve this on a FPGA it is necessary to buffer 16 rows of image. Once the first 8 rows have been written to the buffer, a signal indicates that one row of data blocks are now available for block processing. The data in these 8 rows are read and processed in block order while data continues to stream into the buffer for the next 8 rows. The two sets of 8 rows effectively form a ping-pong buffer with data being written to one and read from the other. On a FPGA this buffer can be implemented using a dual-port

RAM block. One port is used to write the values being streaming in, while the second port is used to read the values in block order. A counter has been used to get the row address; pixel address and the block address in-order to maintain the correct block. Rows are assumed to be a power of 2 long. When writing to the block processing, two counters are used. A 4 bit counter counts the rows, and another counter counts pixels. The two counters are concatenated to give the memory address for writing. At the end of every 8^{th} row, a pulse is generated which triggers reading out the data in block order. For readout, the three least significant bits of the row counter are inserted into the pixel counter, as shown in the Figure 4.2.

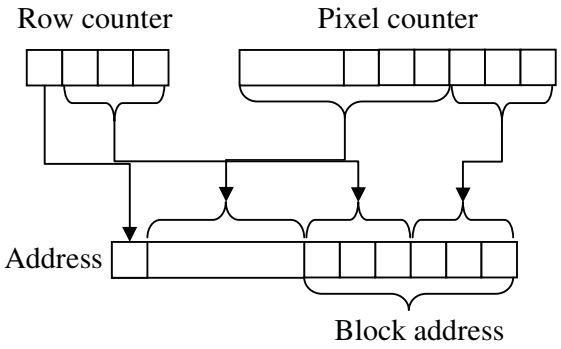


Figure 4.2: Addressing done on block processing.

A synchronisation pulse is generated for the first of each 8 pixels to control the timing of the DCT. Figure 4.3 shows the block processing for a 16×16 image. Red arrows indicate how the pixels are written in to the 1^{st} buffer and green arrows indicate how the pixels are read out in block order.

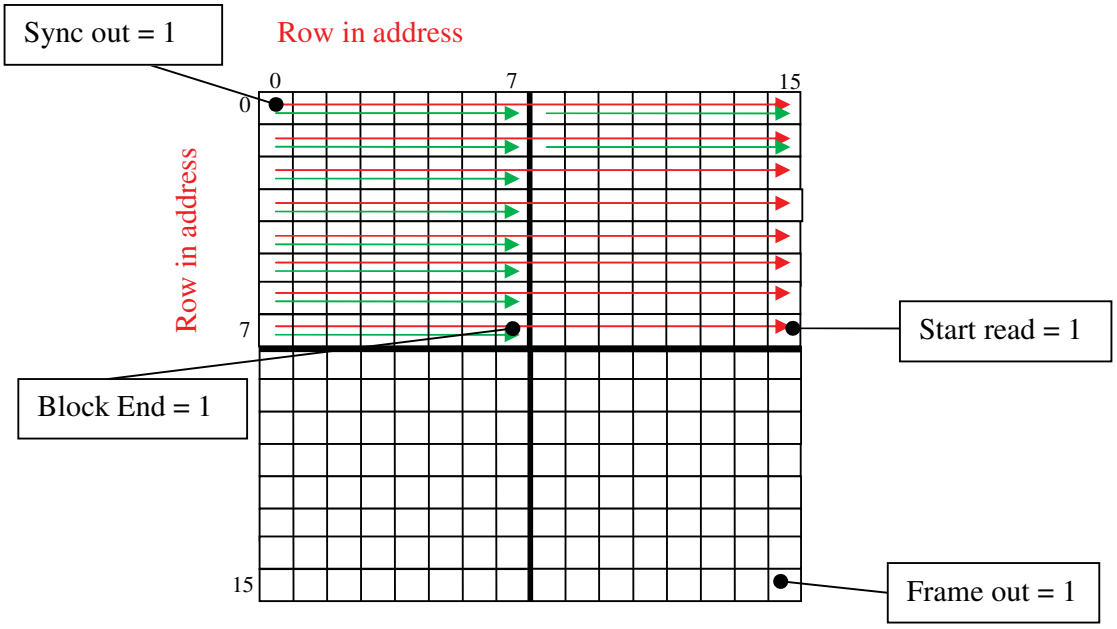


Figure 4.3: Block diagram of block processing for 16x16 image.

4.3 Level Shift

Before computing the DCT of the 8x8 block, its pixel values are shifted from a positive range to one centered around zero. For an 8-bit image, each entry in the original block falls in the range of 0 to 255. Therefore each pixel value is level shifted by subtracting 128 to produce a data range that is centered on zero, so that the modified range is in -128 to 127. This step reduces the dynamic range requirements in the DCT processing stage that follows. This is good for DCT as any symmetry that is exposed will lead toward better entropy compression.

4.4 Discrete Cosine Transform

The DCT is the basis for the JPEG compression standard. This was first introduced in 1974 by Ahmed et al. [42]. The primary purpose of image transformation within an image coding context is to concentrate the energy into as few components as possible [11]. This enables efficient compression by allowing quantisation of many

insignificant elements to zero. The DCT algorithm is completely reversible making it useful for both lossless and lossy compression techniques.

The DCT is closely related to the Fourier transform. Whereas the Fourier transform has both sine and cosine components, the DCT is made purely from cosine terms. This is achieved by enforcing even symmetry and enables the transform to be calculated with real rather than complex numbers [11].

For JPEG, a 2D-DCT algorithm is used, and because the 2D-DCT is separable, it can be split into a series of 1D-DCTs on the rows, and then on the columns. With this property, there are numerous ways to efficiently implement a software or hardware based DCT module [43]. The DCT is applied independently to each 8×8 block of pixels. After each input block of pixels is transformed to frequency space using the DCT, the resulting block contains a single DC component which is the average pixel value of the entire block, and 63 AC components which represent the spatial frequencies that compose the input pixel block. The coefficients effectively scale a set of 2D basis functions made up of the cosine terms. Figure 4.4 represents the 2D-DCT basis functions where mid grey corresponds to 0, with positive coefficients being lighter and negative coefficients being darker.

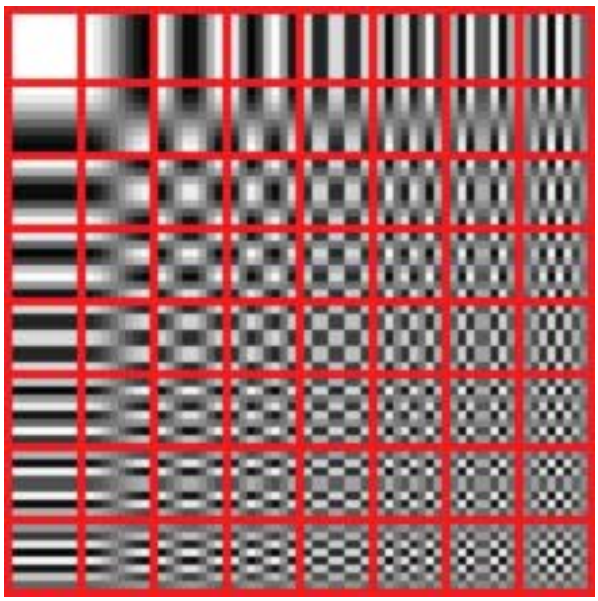


Figure 4.4: 2D-DCT basis functions

The advantage of the representation in the frequency domain is that, unlike in the spatial domain before the DCT, not every dimension has the same importance for the visual quality of the image. Removing the higher frequency components will reduce the level of detail but the overall structure remains almost the same, since it is dominated by the lower frequency components. This is essentially why DCT is used for JPEG compression. The DCT is effective in producing good quality images at low bit rates and is easy to implement with hardware based algorithms [44].

4.4.1 One Dimensional DCT (1D-DCT)

With the property of separability, a two dimensional DCT can be calculated by taking the 1-D transform of the rows followed by the 1-D transform of the columns. While algorithms for direct computation of the 2-D transform can be developed that require fewer arithmetic operations than the separable transform, the separable algorithm allows hardware to be reused and results in simpler implementation for streamed data [11].

The most common DCT definition of a 1-D sequence of length N is:

$$F[u] = \sum_{x=0}^{N-1} \sqrt{\frac{\alpha}{N}} f[x] \cos\left(\frac{\pi u \left(x + \frac{1}{2} \right)}{N} \right) \tag{4.1}$$

where $\alpha(u)$ is defined as: $\alpha = 1$ when u = 0, otherwise $\alpha = 2$

Similarly, inverse transform is defined as:

$$f[x] = \sum_{u=0}^{N-1} \sqrt{\frac{\alpha}{N}} F[u] \cos\left(\frac{\pi u \left(x + \frac{1}{2} \right)}{N} \right) \tag{4.2}$$

4.5 Different Approaches to Implement 1D-DCT

There are several fast parallel algorithms for implementing the DCT (see for example [17, 38, 39, 45]). Table 1 outlines most of the well-known fast algorithms for performing a 8 element 1D-DCT, listing the number of multiplications and additions/subtractions each requires.

Table 1. Fast algorithms for implementing 1D-DCT.

Algorithm	Number of Multipliers	Number of adders / subtractions
Chen [38]	16	26
Hou [45]	14	30
Cvetkovic [39]	12	29
Loeffler [17]	11	22

The most efficient algorithm is that of Loeffler et al. [17], which is shown in Figure 4.5.

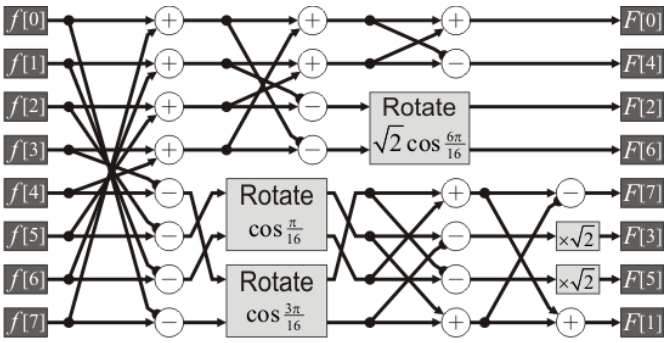


Figure 4.5: DCT architecture with 11 multiplications [11].

The scaled DCT was introduced in 1995 [13] and is shown in Figure 4.6. In this implementation the number of multiplications has reduced to 5 at the expense of each coefficient having a different scale factor. For JPEG compression, this scaling can be taken into account with the quantisation stage by scaling the quantisation step size by the corresponding amount. Agostini et al. [16] have implemented this on an FPGA for JPEG image compression. They divided the algorithm into six blocks to share hardware and therefore they had a latency of 48 clock cycles.

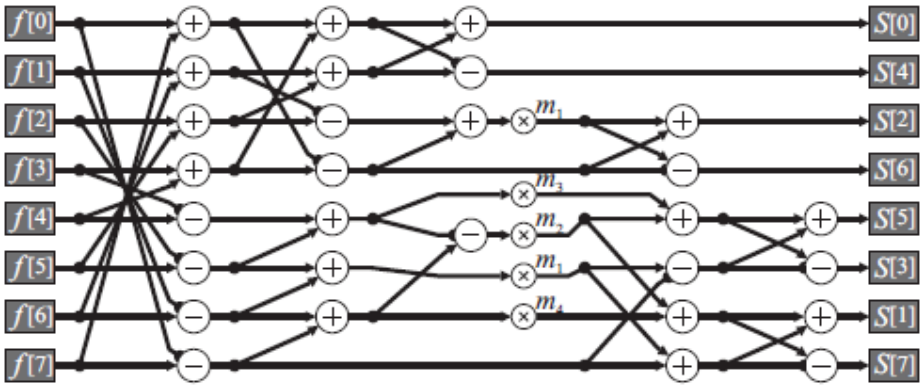


Figure 4.6: DCT architecture with 5 multiplications [11].

These designs are parallel in that they assume that the 8 samples are available simultaneously on the input. For stream processing, they can be adapted with one input pixel arriving each clock cycle and producing one output value per clock cycle.

A streamed pipelined architecture of the above design is shown in Figure 4.7. This design maintains a throughput of one pixel per clock cycle, and has a latency of only 9 clock cycles.

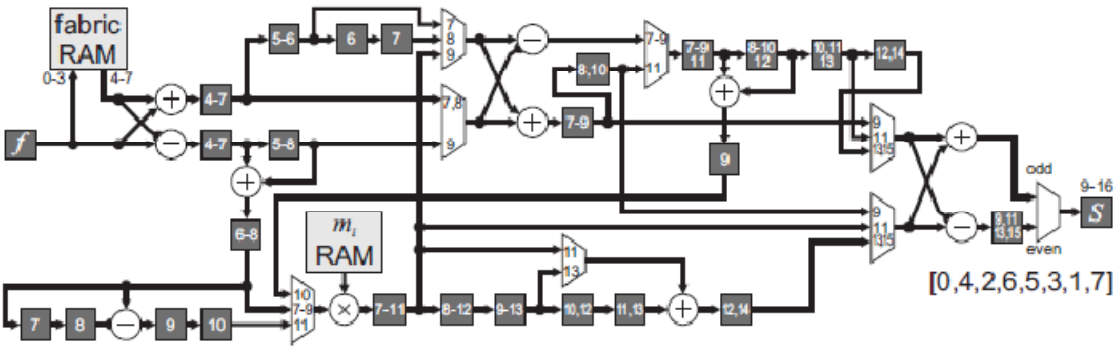


Figure 4.7: Pipelined implementation of figure 4.6 [11].

A major limitation with this method is the resulting architecture. It sacrifices regularity to achieve the low number of multipliers. That is generally not a good trade-off in FPGA design, where highly irregular architectures do not translate into efficient implementations, primarily because of increased routing cost. A balance between the number of multipliers and quality of architecture is essential for an efficient implementation.

4.6 Proposed DCT Architecture

Modern FPGAs have plentiful multipliers. This enables a simpler and elegant pipelined design to be implemented. It is based on a first order factorisation by Woods et al. [40]. The main difference with this design is that it uses one level of factorisation to reduce the number of multiplications for each coefficient from eight to four. The factorisation means that the even and odd samples are calculated separately.

$$\begin{bmatrix} F[0] \\ F[2] \\ F[4] \\ F[6] \end{bmatrix} = \begin{bmatrix} c_4 & c_4 & c_4 & c_4 \\ c_2 & c_6 & -c_6 & -c_2 \\ c_4 & -c_4 & -c_4 & c_4 \\ c_6 & -c_2 & c_2 & -c_6 \end{bmatrix} \begin{bmatrix} f[0]+f[7] \\ f[1]+f[6] \\ f[2]+f[5] \\ f[3]+f[4] \end{bmatrix} \quad (4.3),$$

$$\begin{bmatrix} F[1] \\ F[3] \\ F[5] \\ F[7] \end{bmatrix} = \begin{bmatrix} c_1 & c_3 & c_5 & c_7 \\ c_3 & -c_7 & -c_1 & -c_5 \\ c_5 & -c_1 & c_7 & c_3 \\ c_7 & -c_5 & c_3 & -c_1 \end{bmatrix} \begin{bmatrix} f[0]-f[7] \\ f[1]-f[6] \\ f[2]-f[5] \\ f[3]-f[4] \end{bmatrix} \quad (4.4)$$

$$\text{where } c_k = \frac{1}{2}\cos\frac{k\pi}{16}.$$

Equations (4.3) and (4.4) require 32 multiplications and 32 additions to perform each 1D transform. This corresponds to 512 multiplication and 512 additions for the full 8x8 block. This approach does not achieve the lowest number of multiplications but it achieves a more efficient solution. The implementation of this approach, which directly performs the matrix multiplication, is shown in Figure 4.8.

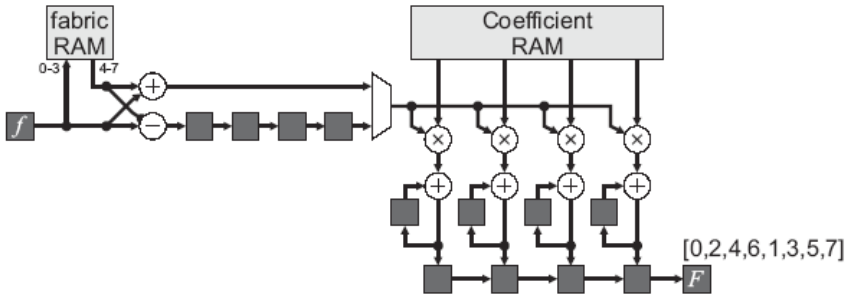


Figure 4.8: Block diagram of proposed DCT architecture [11].

The whole process is pipelined to operate on streamed input data at one pixel per clock cycle. A synchronisation pulse is provided with the first pixel in each row of eight. This controls the sequencing of operations of the DCT module. The memory at the start holds the first 4 samples, and returns them in reverse order to calculate the sum and differences in (4.3) and (4.4). Each multiply and accumulate unit is reset every four clock cycles, and calculates a separate output frequency. First, four clock cycles calculate the even frequencies using (4.3), and then while the multiplication and accumulating is happening with the first 4 elements, the differences for odd frequencies are stored in the input shift registers. Then, while the even coefficients are being streamed out using registers, the odd coefficients are calculated using (4.4). An output synchronisation pulse is provided with the first coefficient of each row to control the next stage in the pipeline. This converts the parallel implementation into pipelining since we need only one output per clock cycle.

Fixed point arithmetic is used to simplify the multiplier logic. Scaling the coefficients by a power of 2 makes all of the operations integer.

$$c_k' = \text{round}\left(c_k 2^B\right) \tag{4.5}.$$

The number of bits output from the multiply and accumulate is reduced by truncating unwanted bits. Initialising the accumulator with an appropriate value converts the truncation to rounding.

In this research, pipelining is used to begin a new DCT operation before the previous DCT operation is completed. The multiply and accumulate units are then utilised with every clock cycle. The outputs are not in natural order; for JPEG compression, this does not matter because the data can be reordered later during the zigzag operation.

4.7 2D-DCT Implementation

The 2-D DCT of a data matrix is defined as,

$$\boldsymbol{D = M.X.M^T} \tag{4.6},$$

where X is the data matrix, M is the matrix of DCT coefficients for implementing a 1D-DCT, and M^T is the transpose of M.

With separability, (4.6) can be transformed to (4.7)

$$\boldsymbol{D^T = M(MX)^T} \qquad (4.7).$$

This implies the hardware implementation shown in Figure 4.9. The 2-D DCT designed in this work was broken in to three main blocks: a first 1-D DCT on the rows, a transposition buffer and a second 1-D DCT on the columns. The difference between the two 1-D DCT architectures is the number of bits used in each stage of pipeline. The transposition buffer stores the results of first 1-D DCT, row-by-row and sends these results to the second 1-D DCT, ordered column-by-column.

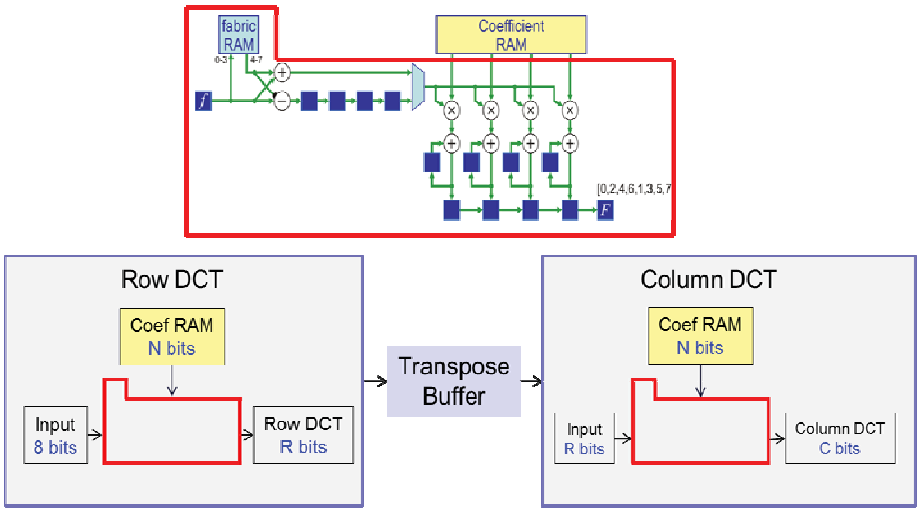


Figure 4.9: Fixed point DCT implementation.

The next section of this chapter will describe the process for determining the number of bits required to implement the DCT shown in Figure 4.9.

4.7.1 DCT Word Length Optimisation

Due to the complications and large number of logic gates needed to perform floating point operations in hardware, a fixed point representation was chosen. In FPGAs, it is well known that a fixed-point implementation uses significantly less logic than a

floating point implementation [46]. Therefore all variables were scaled by a fixed power of 2 and represented as signed or unsigned integers. When arithmetic operations were performed, operands were explicitly shifted to ensure alignment. By using fixed point numbers in Handel-C the bit length of each step can be defined to be different. This gives a great deal of flexibility in design and can save on the amount of hardware used for both computation and registered outputs.

During the design stage, the data width of each term was chosen individually to minimise the logic usage while keeping the truncation error within a pixel value of 1 in the reconstructed image, and keeping sufficient bits to avoid overflow. It is necessary to determine how many bits are required to represent the DCT coefficients, and since a multiplication increases the bit-width of the numbers, the output of the multiply and accumulate is an obvious truncation point. Table 2 describes the bit allocation for each step in the DCT process.

Table 2. Propagation of bit width and binary point position.

	Sign	**Total bits**	**Binary places**
Input Pixels*	signed	8	0
DCT Coefficients	signed	*N*	*N*
Sum and Difference	signed	9	0
After multiplication	signed	*N*+9	*N*
After row DCT	signed	*N*+11	*N*
Want *R* bits after row DCT. Therefore drop (*N*+11-*R*) bits			
After truncation of row DCT	signed	*R*	*R*-11
Sum and Difference	signed	*R*+1	*R*-11
Multiplication	signed	*N*+*R*+1	*N*+*R*-11
After column DCT (2D-DCT)	signed	*N*+*R*+2	*N*+*R*-11
Want *C* bits after 2D-DCT. Therefore drop (*N*+*R*+2-*C*) bits			
After truncation of 2D-DCT	signed	*C*	*C*-13

*Unsigned pixels are offset by 128 before DCT

N = Number of bits to represent DCT coefficients

R = Number of bits want after row DCT (1D-DCT)

C = Number of bits want after column DCT (2D-DCT)

First, a large number of bits are assigned to *R* and *C* to ensure that these will not significantly limit the accuracy of the result (20 bits are sufficient). Then the number of bits required to represents DCT coefficients, *N*, is varied, and the 2D-DCT is performed. An inverse DCT is calculated on the result using double precision floating point with MATLAB's inbuilt IDCT function. The difference between the original image and the reconstruction is taken to determine the errors introduced through the reduced precision arithmetic.

The results are shown in Figure 4.10 for the Lena image and indicate that 9 bits are sufficient for representing the DCT coefficients. Having more than 9 bits will not give any significant improvement to the image.

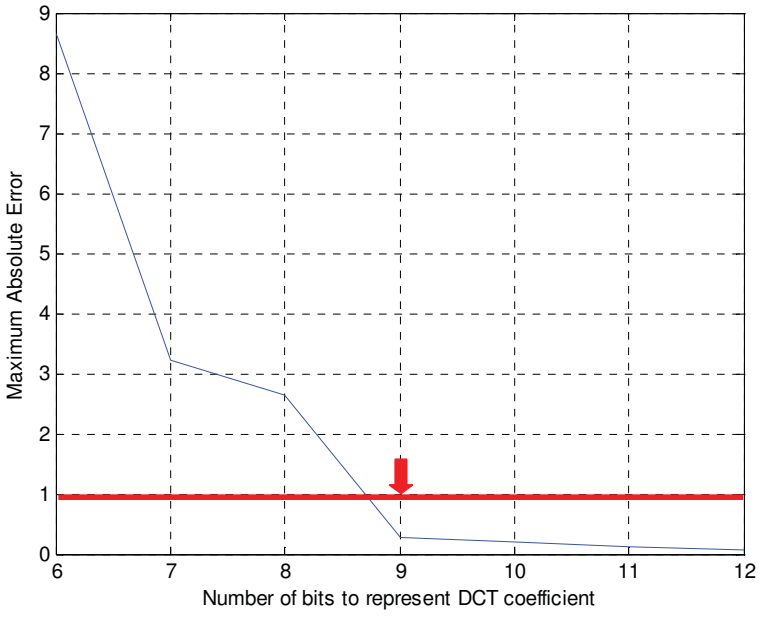


Figure 4.10: Effect of different DCT coefficient quantisations with R = 20, and C = 20. Arrow shows quantisation which gives less than 1 MAE.

A similar process was conducted to determine the optimum values for *R* and *C*. The results of these experiments are shown in Figure 4.11and Figure 4.12. From Figure 4.11, 12 bits are required to represent the output of the row DCT, and from Figure 4.12, 14 bits are required to represent the output of the 2-D DCT.

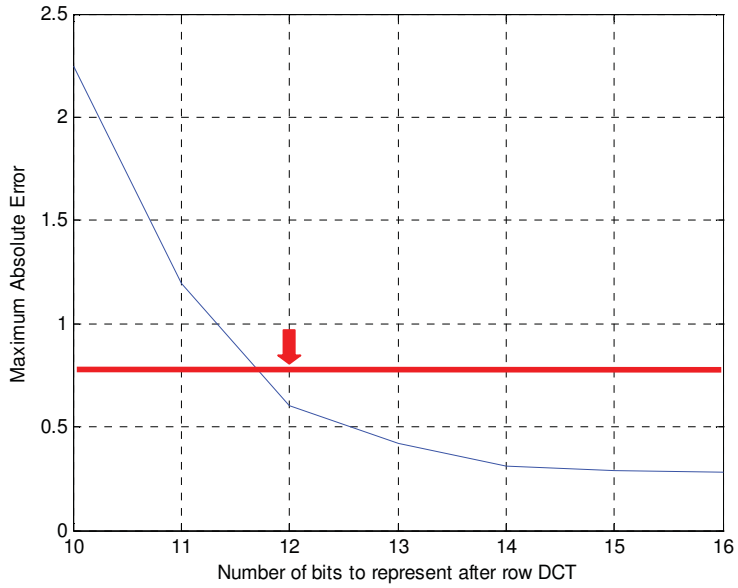


Figure 4.11: Effect of different row DCT quantisations with N = 9, and C = 20 Arrow shows quantisation which gives less than 1 MAE.

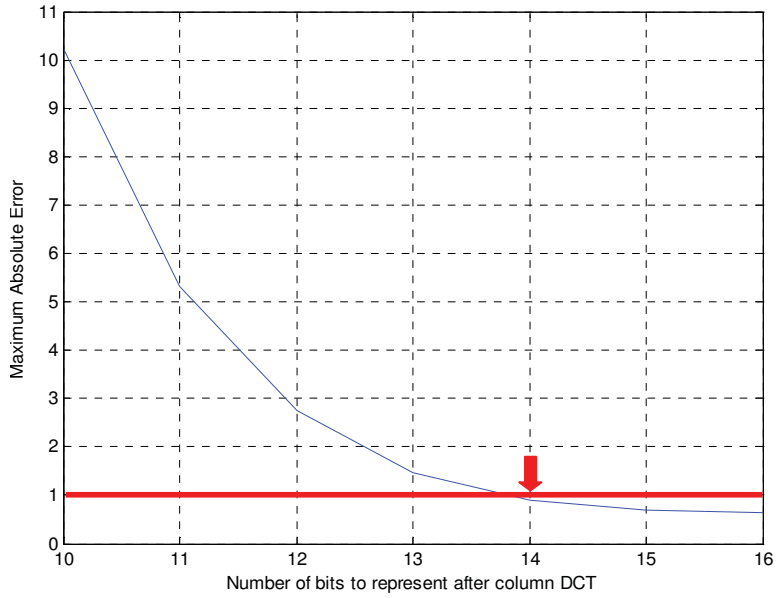


Figure 4.12: Effect of different column DCT quantisations with N = 9 and R = 12 Arrow shows quantisation which gives less than 1 MAE.

Therefore the following Barbara and Lena images are tested using the following bit-widths: N=9 R=12 C=14

The reconstructed Barbara image has RMSE of 0.2086 and maximum absolute error (MAE) of 0.8982 pixel value on the greyscale pixel range (0 to 255). Lena image has RMSE of 0.2095 and MAE of 0.8131 pixel value. These error values show that there

is no impact on visual or perceived quality of the reconstructed image and that the errors are within the desired one pixel value. These errors are not apparent to human eye due to the fact that the difference is too small to see.

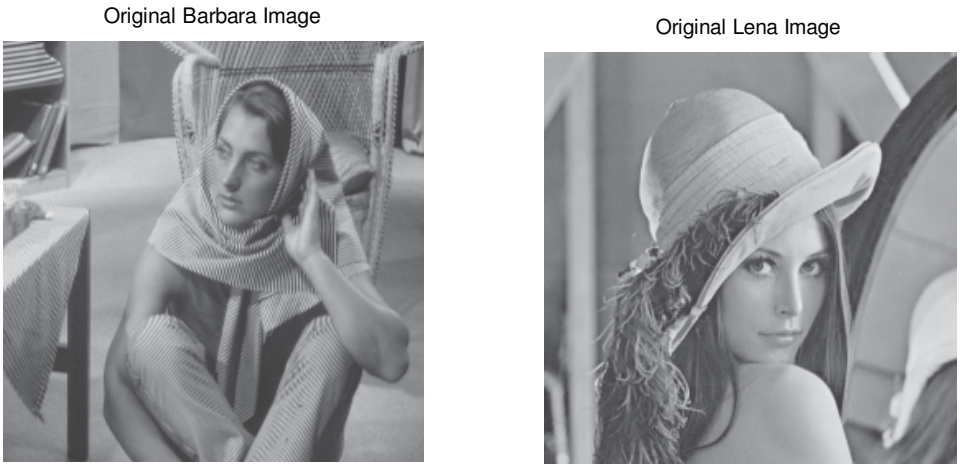


Figure 4.13: Original Barbara image (Left) and Original Lena image (Right)

4.7.2 Transpose Buffer

The transpose buffer is used to connect the two 1-D DCT architectures, where the results from the first 1-D DCT are stored row-by-row, and read column-by-column by the second 1D-DCT. To enable simultaneous read and write access, a dual-port RAM block on the FPGA is used. One port is used to write the results of the row transform, while the second port is used to read the values in column order. Although the memory can be reused, requiring only a 64×12-bit memory, the design is simplified by using a 128×12-bit memory. On the Cyclone IV, memory blocks are 9 kbits in size; therefore using 128×12-bit memory does not use more memory. This is divided into two blocks of 64 entries, with the blocks used in ping-pong mode. The row data is written into the first block during the first 64 cycles. Once the eight row DCTs computation is completed, the data can be streamed out of the transpose buffer in column order for the column DCTs. The address logic to write and read to the buffer consists of a counter with 7 bits. When there is an input pixel, the counter increments and when it reaches to 64 it sends a signal to start reading from the buffer in column order. For reading out, a 7-bit address counter is also used, with bits 0-2 and 3-5

swapped to swap the rows and columns. An output synchronisation pulse is provided with the first coefficient of each column to control the column DCT.

The data for the next 8×8 block can be loaded during the operation of the column DCT, creating a pipelined architecture. Thus, the output of row DCT computation is transposed for column DCT computation.

4.8 Summary

The proposed DCT architecture has a latency of 10 clock cycles for each row DCT and column DCT. The transpose buffer has a latency of 64 clock cycles. The output of the row DCT is rounded to 12 bits. The output of the column DCT process is rounded to 14 bits. Since the 2-D DCT samples will be reordered later with the zigzag reordering, the fact that the output stream is ordered by column can be corrected later.

5

Implementation of Zigzag Coding and Quantisation

This section gives the basic overview of zigzag coding and quantisation. It discusses the hardware implementation for each module; and presents the results after zigzag coding and quantisation obtained through the proposed approach.

5.1 Zigzag Coding

After 2D-DCT processing, zigzag coding is used to transform the 8×8 block into a sequential list of 64 values. The zigzag process organises the sequence to have the lower frequency components, which are less likely to be zero, in the first part of the data stream. This attempts to organise the data to have long runs of zeros, especially at the end, making run length coding very efficient [2]. Figure 5.1 and Figure 5.2 below show the standard order that coefficients come out after zigzag coding.

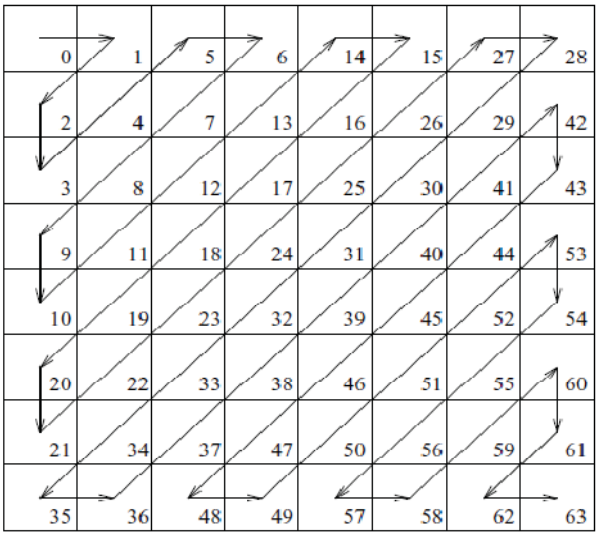


Figure 5.1: Sequence obtained by zigzag.

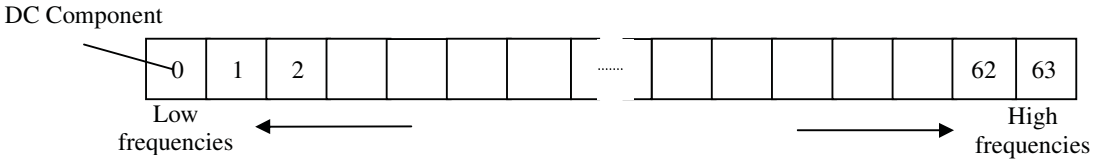


Figure 5.2: Zigzagged coefficients in 64 element vector.

5.2 Hardware Implementation of Zigzag Buffer

Zigzag reordering is achieved by writing the coefficients for a block into a buffer, and reading them out again in the required order. The architecture for the zigzag buffer (see Figure 5.3) is therefore very similar to the transpose buffer used in the 2-D DCT. A dual-port 128×14-bit memory block is split into two 64 entry buffers. On receipt of an input synchronisation pulse, 64 DCT coefficients are written to one port of the buffer. Once the 64 coefficients have been written, an output synchronisation pulse is generated which triggers a readout on the other port in zigzag order in the subsequent 64 clock cycles. At the same time, the next 8×8 block can be loaded into the second buffer. For each 8×8 block, the role of the buffers is swapped in ping-pong mode.

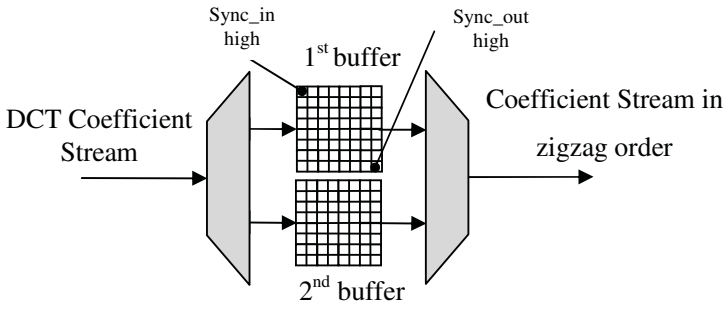


Figure 5.3: Zigzag buffer architecture.

Figure 5.4 shows how the output addressing is done in the zigzag module. The sequential addresses of a counter are converted to zigzag order by a lookup table (LUT). The LUT is implemented using a memory block on the FPGA. The LUT entries also take into account the non-sequential ordering of the data from the DCTs (even/odd grouping of the data from the DCT), and that the coefficients are transposed.

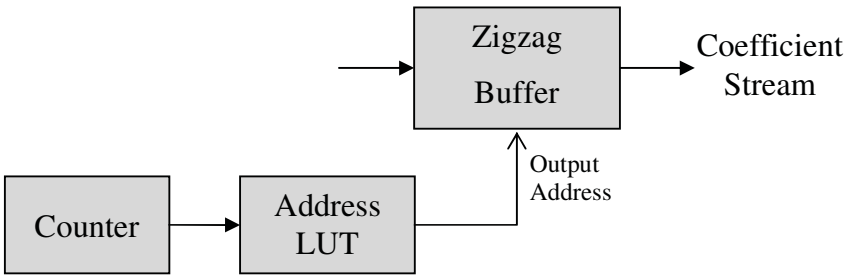


Figure 5.4: Addressing of zigzag reordering.

The Figure 5.5 shows the actual zigzag order that coefficients are read out after considering the transpose and even and odd grouping from DCT module. This is the information stored in the address LUT.

Rows

Columns	0	2	4	6	1	3	5	7
0	0	3	10	21	2	9	20	35
2	5	12	23	37	7	18	33	48
4	14	25	39	50	16	31	46	57
6	27	41	52	59	29	44	55	52
1	1	8	19	34	4	11	22	36
3	6	17	32	47	13	24	38	49
7	15	30	45	56	26	40	51	58
9	28	43	54	61	42	53	60	63

Figure 5.5: Actual zigzag order that coefficients read out.

5.3 Quantisation

Quantisation is an extremely important step in the JPEG compression algorithm, as it helps to reduce a considerable amount of data, thus reducing the entropy in the input data stream. The quantisation is an integer division of all the 2D-DCT coefficients by constants and rounding to the nearest integer value. For image compression, different transformed coefficients have different visual significance, so different quantisation step sizes are used for each coefficient. The JPEG standard suggests a table specifying the quantisation step sizes for each coefficient [2] which is shown in Table 3.

Table 3. Standard luminance quantisation table

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

The quantisation step is:

$$Cq_{ij} = \text{round}\left(\frac{C_{ij}}{Q_{ij}}\right) \qquad (5.1),$$

where Cq_{ij} is the output quantized coefficient; C_{ij} is the input 2-D DCT coefficient; and Q_{ij} is the Quantisation constant from Table 3.

Quantisation is the operation that introduces information losses in the JPEG compression process. This is intended to remove the components less important to the visual reproduction of the image. The aim of quantisation is to compress the image as much as possible without visible artefacts. Each step size ideally should be chosen as the perceptual threshold for the visual contribution of its corresponding cosine basis function. Larger quantisation step sizes will result in visual artefacts.

Typically, a single quality control parameter is used to control image quality and compression in an image codec. This quality factor, used to scale the values in the quantisation table, effectively adjusting the number of steps in the resulting quantised value [43]. Larger quantisation steps will lead to greater distortion and a smaller resulting data set.

According to [11] the quality factor, QF, which varies from 1 to 100 scales the standard quantisation table according to:

$$\dot{Q}_{ij} = \begin{cases} round\left(\dfrac{50Q_{ij}}{QF}\right), QF < 50 \\ round\left(Q_{ij}\left(2 - \dfrac{QF}{50}\right)\right), 50 \le QF \end{cases} \tag{5.2}$$

Figure 5.6 shows the maximum error for Barbara image for different quality factors. MATLAB was used to obtain these results. It clearly shows that as the quality factor increases, the error becomes really small.

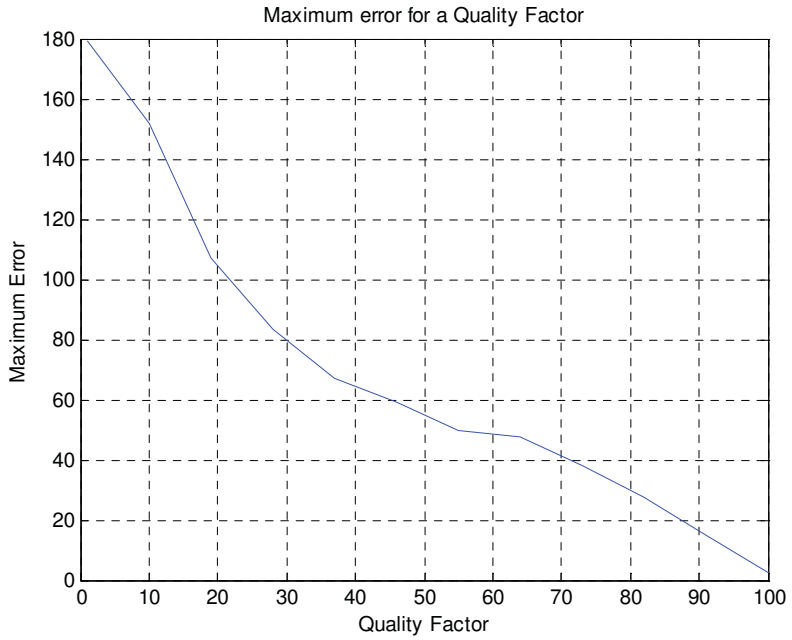


Figure 5.6: MAE for Barbara image for different quality factors.

5.4 Hardware Implementation of Quantisation

Quantisation is an integer division of each DCT coefficient by the corresponding constant, and rounding the result to the nearest integer. The division can be implemented very efficiently in hardware as a shift operation if the quantisation factors are restricted to powers of 2 [12]. However, this is overly restrictive.

There are several algorithms which perform division in digital designs, for example, standard long division, non-restoring division and SRT division. Each method has its own advantages and disadvantages, so it is necessary to select an appropriate divider

according to the application. For quantisation, the range of divisor is from 0 to 255, and non-restoring algorithms give a good compromise between cost and latency when the operator's size is not large, it is appropriate to use non-restoring algorithm to design the divider [47].

Therefore, the division was implemented using a signed by unsigned non-restoring divider. The algorithm used to implement the division is based on [41]:

$$R_i = 2R_{i-1} + \begin{cases} D & \text{if } R_{i-1} < 0 \\ \overline{D}+1 & \text{if } R_{i-1} \geq 0 \end{cases} \tag{5.3}$$

$$q_i = \begin{cases} -1 & \text{if } R_{i-1} < 0 \\ 1 & \text{if } R_{i-1} \geq 0 \end{cases} \tag{5.4}$$

where R_i is the partial remainder, D is the divisor, $\overline{D}$ is its one's complement, and q_i is the i^{th} bit of the quotient. The assumption is that $|R| \leq D$, Otherwise overflow will occur.

This algorithm will initially use the dividend, shifted by the required number of bits as the initial remainder, shift it, and based on the sign, will add or subtract the divisor. This process is repeated to produce a quotient and remainder. The addition of the 1 as part of the 2's complement does not actually require additional logic because the $2R_{i-1}$ will leave the least significant bit as 0. The 1 can be inserted instead if $q_i = 1$ [41].

The advantage of using non-restoring division over the standard restoring division is that a test subtraction is not required. The sign bit determines whether an addition or subtraction is used. If there are not enough bits to represent the result then overflow occurs. This can easily be detected because the first subtraction should always change the sign of the partial remainder.

5.4.1 Summary

The zigzag buffer has a latency of 64 clock cycles. The zigzag ordered coefficients are then passed in to the quantisation module in order to achieve more compression.

The quantisation module is shown in Figure 5.7. It consists of RAM to store the quantisation table. Quantisation was implemented assuming the quality factor is 50 which uses the quantisation values in Table 3 without scaling.

The inputs to the quantisation module have 14 bits and the output has 8 bits. The latency of the quantisation module is one clock cycle. Note that the quantisation table is reordered from that shown in Table 3 to account for the zigzag ordering of the data.

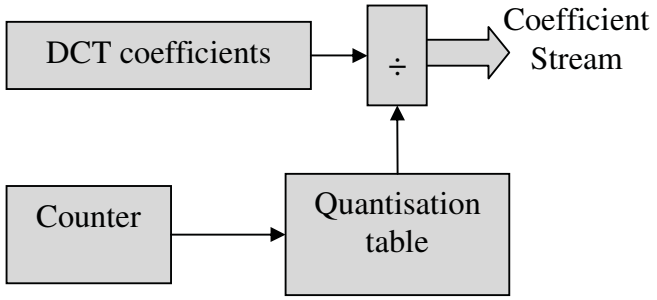


Figure 5.7: Quantisation Module.

6

Implementation of Entropy Coding

This chapter describes entropy coding as used by the JPEG compression algorithm. It gives the detailed implementation and discusses the proposed method of Huffman coding and results obtained through this approach.

6.1 Entropy Coding

Each 8×8 block has one DC coefficient, and 63 AC coefficients. Sequences of successive zero AC coefficients are run-length encoded to reduce the number of symbols which need to be output for each block.

The final stage of JPEG compression is entropy coding. This assigns a variable length code to each symbol in the output stream based on the frequency of occurrence [11]. The objective of entropy coding is to use fewer bits to represent a symbol which appears more frequently and more bits to represent a symbol which appears less frequently. With JPEG compression, the symbols actually encoded are the run-length of consecutive zero coefficients, and size of the quantised coefficient. The coefficient itself is simply written to the output bit stream (also using a variable number of bits).

One of the most common forms of entropy coding is Huffman coding. This uses the optimum integer number of bits for each symbol. Huffman coding requires a Huffman code table to be specified by the application. Since the DC and AC coefficients have quite different statistics, a separate Huffman table is used for the DC and AC symbols. Each block consists of one DC codeword, and one or more AC codewords. The same Huffman tables used to compress an image will be used to decompress it.

6.1.1 DC Differential Coding

Differential coding is used to reduce the entropy of the DC coefficient since adjacent blocks are likely to have similar average values. The DC value of the first block is passed directly to the Huffman coding module. For subsequent blocks, the value coded is the difference between the DC value of the current block and the DC value of the previous block.

$$DC_{code} = DC_i - DC_{i-1} \tag{6.1}$$

The symbol actually coded is the size of DC_{code} according to the range in Table 4 [2]. This is then followed by the actual coefficient using the required number of bits.

6.1.2 Run length Coding

The run length encoder receives the sequences from the zigzag module and looks for the runs of zeros in the AC coefficients. It counts the zero coefficients between each non-zero coefficient. The size of a non-zero coefficient is the number of bits required to represent the value. The combined run length and coefficient size jointly make up an AC symbol, which is Huffman encoded, and followed by the non-zero coefficient [12].

Table 4. The relationship between size and amplitude.

Size	Amplitude
0	0
1	-1, 1
2	-3, -2, 2, 3
3	-7 ~ -4, 4 ~ 7
4	-15 ~ -8, 8 ~ 15
5	-31 ~ -16, 16 ~ 31
6	-63 ~ -32, 32 ~ 63
7	-127 ~ -64, 64 ~ 127
8	-255 ~ -128, 128 ~ 255
9	-511 ~ -256, 256 ~ 511
10	-1023 ~ -512, 512 ~ 1023
11	-2047 ~ -1024, 1024 ~ 2047

6.1.3 Huffman Coding

This is the last step in the encoding process. It packs the data by assigning unique variable length codewords for each symbol that can be recovered without loss during decompression. The run length encoder and the Huffman encoder modules are packaged together to generate one block of codeword for each non-zero coefficient.

A DC codeword consists of two parts, the size and the amplitude of the coefficient. Table 5 is the Huffman table for the DC coefficient size.

Table 5. Huffman table for luminance DC coefficient differences.

Size	Bits	Code
0	2	00
1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

Appendix A-1 gives Huffman tables for the AC coefficients which have been developed from the average statistics of a large set of 8-bit images [2].

Two special symbols are introduced. One is EOB which represents that the remainder of the block is zeros. This code can appear at any time in the sequence to indicate that the remaining data values are all zeros. The second is ZRL which is used when a run-length greater than 16 is encountered. Since a maximum run of 15 zeros is allowed, the ZRL code represents a block of 16 zeros without a corresponding non-zero coefficient.

6.2 Hardware Design for Huffman Coding

6.2.1 Huffman Coding Architecture

The architecture of the entropy encoder is shown in Figure 6.1. The DC coefficient of each block is passed to the Differential Coding module to subtract the DC coefficient of the previous block. Then the difference is passed to the size detector. For the AC coefficients, the zero-run counter counts the number of successive zeros in the streamed output from the zigzag process. If the input coefficient is nonzero, the coefficient is sent out to the size detector and the zero-run counter is reset to zero.

The size detector determines the number of bits required to represent the coefficient. Firstly, 1 is subtracted from negative coefficients. Secondly the most significant bits, which are identical, are eliminated. The remaining bits form the coefficient value, and the number of bits is the size according to Table 4. Lastly the bit counting is performed efficiently in a single clock cycle using a multiplexer based successive approximation counter. For the 8 bit input coefficient, the counter works as follows:

- the 5 most significant bits MSBs of the remainder are checked. If all 5 bits are same, the 4 least significant bits (LSBs) are selected and a 0 is output, otherwise the 4 MSBs are selected and a 1 is output.
- the process is repeated, checking the 3 MSBs of the result, and selecting either the 2 LSBs or 2 MSBs.
- Finally the process is repeated checking the 2 remaining bits.
- The three bits output give the size of the coefficient.

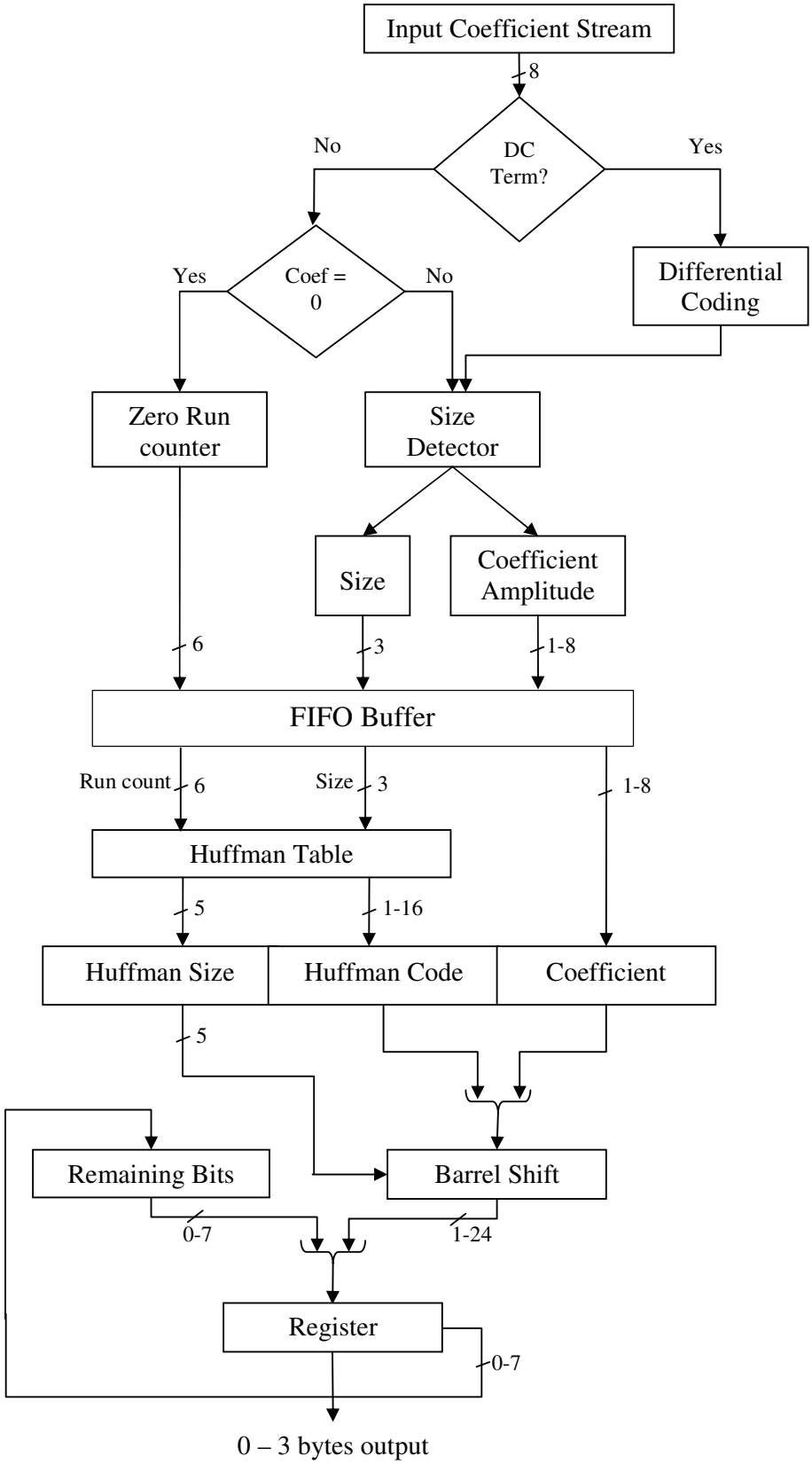


Figure 6.1: Huffman coding architecture.

These results (run length, size and coefficient amplitude) are then stored in FIFO buffer which has a depth of 4. This will then be used to determine the Huffman code. The FIFO buffer allows the ZRL and EOB codes to be encoded without having to pause the incoming coefficient stream.

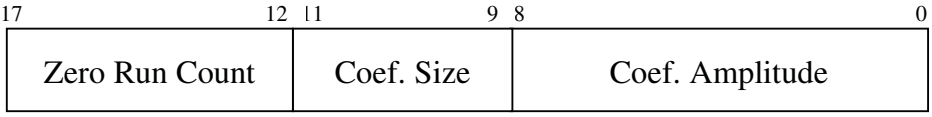


Figure 6.2: FIFO buffer bit allocation.

A lookup table based approach has been used to assign the Huffman code for each coefficient. The size and run length are looked up in the Huffman table shown in Appendix A-1 to determine the corresponding Huffman code (and length of the Huffman codeword, which is used for packing the bits into the output). The Huffman code is concatenated with the coefficient amplitude to form the codeword for each coefficient. Obviously, codewords obtained are variable in length. However, the width of the data bus is fixed. A barrel shift is thus used to align the variable-length codewords with the remaining bits. The lengths are added to the number of remaining bits with the most significant bits used to determine the number of completed bytes.

The Huffman encoder outputs up to 3 bytes for each symbol. These are saved into another FIFO buffer with depth of 256 to enable them to be streamed out 8 bits at a time. Any bytes containing all 1s (used in JPEG to indicate an escape code) are followed with a byte containing all 0s so that the image will decode correctly. The FIFO manages the conversion from a fixed rate stream to a variable rate stream. At the end of each image the remaining bits are padded with ones. The Huffman encoder input has 8 bits and has an output of 8 bits.

6.3 Latency

Entropy coding module has a minimum latency of 5 clock cycles. Obviously it is longer when there are sequences of zero coefficients, or when the resulting codewords are more than 8 bits in length. However, as a result of the compression, the number of output bytes is fewer than the number at the input. Therefore, the 5 clock cycles latency will be used in the final calculation. The 5 clock cycles are used to perform the following steps:

1. Do the run length encoding, size detection and store the results to FIFO buffer.
2. Extract each coefficient from FIFO.
3. Do the actual Huffman coding.
4. Pack each code into bytes.
5. Packing bytes into output FIFO.

7

Implementation of JPEG Headers

This chapter gives the details of JPEG headers and discusses the hardware implementation of writing the headers to the output.

7.1 JPEG File Format

A JPEG file consists of more than just the encoded coefficients. A series of headers [2] is used to :

- identify the fact that the file is JPEG encoded,
- provide the size of the coded image,
- provide additional metadata for the image,
- specify the Quantisation and Huffman tables.

The headers are prescribed by the JPEG File Interchange Format (JFIF) specifications [48]. JFIF is a minimal file format which enables JPEG bit streams to be exchanged between a wide variety of platforms and applications.

Figure 7.1 shows the minimum required set of JPEG headers which are required to decode a baseline JPEG compressed image. Compressed image data consists of only one image. An image contains only one frame. A frame contains one or more scans and a scan contains the complete encoding of one or more image components.

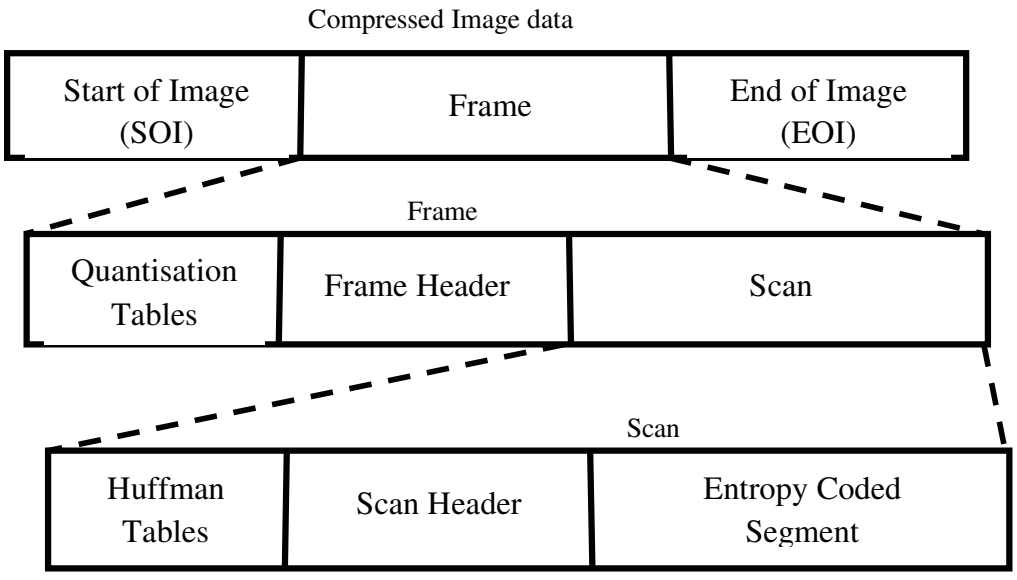


Figure 7.1: High level file structure.

Markers are used to define the header segments. Some markers are followed by particular sequences of parameters, as in the case of table specifications, frame header,

or scan header [2]. A marker will always begin with the first byte as 0xFF, and the second byte defines which type of marker it is. Table 6 lists the key markers. When a marker is associated with a particular sequence of parameters, the marker and its parameters comprise a "marker segment".

Table 6. JPEG file markers.

Marker	Symbol	Description
0xFFD8	SOI	Start of image
0xFFE0	APP0	Application specific; used for JFIF metadata
0xFFDB	DQT	Define quantisation table
0xFFC0	SOF	Start of frame
0xFFC4	DHT	Define Huffman table
0xFFDA	SOS	Start of scan
0xFFD9	EOI	End of image

The JPEG File Interchange Format is entirely compatible with the standard JPEG interchange format; the only additional requirement is the mandatory presence of the APP0 maker after the SOI marker [48].

Details of each marker are given in Appendix A-2.

7.2 Hardware Implementation of JPEG Headers

In this research, a single quality factor was used, so the quantisation tables and Huffman tables were fixed. This enabled the pre-initialised headers to be stored in a memory block. The output byte stream was started by streaming the JPEG headers while the pipeline was being primed at the start of the image.

At the end of the image, the remaining data bits (making up on incomplete byte) were flushed by padding with 1s, and an end of image marker code was output. Insertion of headers takes 328 clock cycles.

8

Simulation Results and Discussion

This chapter presents the overall system architecture and gives the details of the obtained results from the Handel-C simulation. It describes the testing and validation methods for the implementation. It also compares the obtained results with MATLAB compression results and discusses the outcome.

8.1 System Architecture

Figure 8.1 shows the overall system architecture. This shows how the individual modules in the previous chapters connect together. The system was designed to compress grey-scale images, and was implemented using Handel-C.

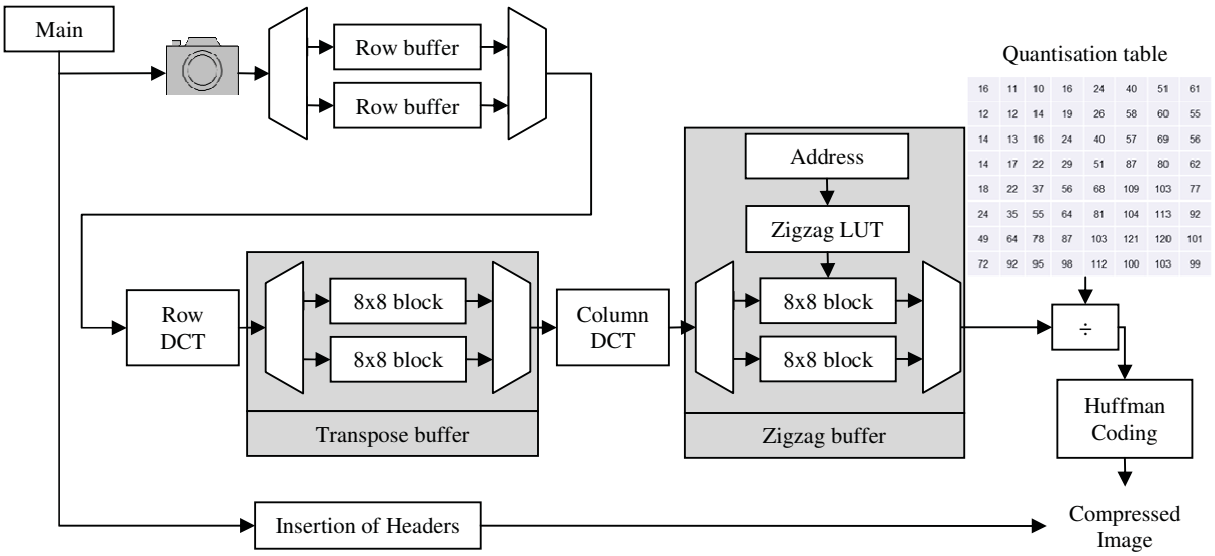


Figure 8.1: Overall system architecture for JPEG compression.

8.2 System Validation and Error Evaluation

To test the system an incremental design testing approach was used. Each JPEG module was firstly tested on its own and then checked for the functionality to validate results in Handel-C simulations. The results were validated using MATLAB and IrfanView.

When programming hardware, the compile times are significantly longer than with software. Debugging is also more difficult in hardware because many things happen at once in parallel, and the timing of the operations is important. For these reasons, the algorithms were first implemented in MATLAB using fixed point arithmetic. This gave a dataset which could test-bench the Handel-C implementation. Any differences in the results indicate an error in the implementation.

Then MATLAB JPEG compression was used as a reference for software compression. This uses the following MATLAB command and it does not consider the architectural restrictions, specifications, and rounding that used for hardware implementation.

```
I=imread('Lena256.bmp');
imwrite(I,'Lena_compress_50.jpg','Quality',50);
```

Therefore it generates a good reference to compare the similarity between the software compression and hardware compression. Then the compressed image was decoded using IrfanView. IrfanView was used to check that the file compressed by Handel-C could be successfully decoded.

PSNR and MSE were calculated for each compressed image in order to compare the error between original image and compressed image. Also the MSE between software compression and hardware compression was calculated to check that the two methods were giving similar results.

8.3 Simulation Results and Discussion

The latency of each step in the process is summarised in Table 7.

Table 7. Latency of each stage

	Latency (cycles)
Header	328
Block Processing	8 rows
Row DCT	10
Transpose Buffer	64
Column DCT	10
Quantisation	2
Zigzag coding	64
Entropy Coding	5
JPEG Compressor	154

As can be seen, the JPEG header takes 328 clocks cycles to stream out. This gives enough time to process the JPEG compression for the first 8×8 block of the image (which takes 8 rows + 154 clock cycles) as long as the image is wider than 22 pixels.

Functional testing and timing analysis were carried out for each JPEG module on its own and then checked in an integrated test for the functionality to validate results in Handel-C simulations. The system has been tested using three 16×16 test patterns (each row was expanded by 6 clock cycles of blanking to satisfy the minimum of 22 clock cycles per row) and the results validated using MATLAB. IrfanView and MATLAB were also used to check that the resulting compressed file could be successfully decoded. The system was then modified to test 256×256 images. Standard test images were used as benchmarks to test functionality of the design and the results validated using MATLAB.

Figure 8.2: Lena reconstructed image (PSNR 36.67 dB)

Figure 8.3: Peppers reconstructed image (PSNR 36.08 dB)

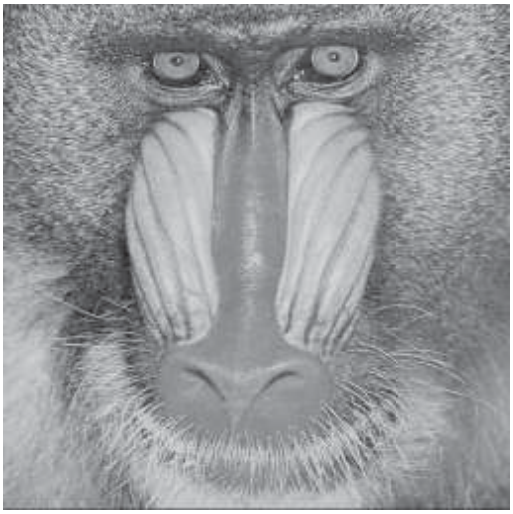


Figure 8.4: Baboon reconstructed image (PSNR 30.74 dB)

Figure 8.5: House reconstructed image (PSNR 38.77 dB)

Figure 8.2 – Figure 8.5 are 256×256 grey scale images. The images are first compressed using quality factor 50 in Handel-C simulation and then decompressed using MATLAB. These are compared with saving the same images from MATLAB with the same quality factor. In Figure 8.2, the compression ratio was 9.23:1 compared with MATLAB's 9.20:1. The small difference results from slight differences in the calculation of the DCT using fixed point arithmetic compared to double precision floating point in MATLAB. While the differences calculated in section 4.7.1 are small (much less than one pixel value), they can move a pixel from one side of a quantisation boundary to another, resulting in significantly larger differences in the compressed images.

The mean square error (MSE) between the original and compressed images from Handel-C and MATLAB are 14.00 and 13.84 respectively (PSNR 36.67 dB and 36.72 dB). These are sufficiently close to be satisfied that the hardware compression is not significantly different from the software compression in terms of quality. Indeed, the MSE between the two compressed images was only 0.8065.

For the image in Figure 8.3, the compression ratios are 9.05:1 for Handel-C and 9.02:1 for MATLAB. The MSE of the reconstructed image is 16.03 from Handel-C and 15.91 from MATLAB (PSNR of 36.08 dB and 36.11 dB respectively). The MSE

between the compressed images was 0.8335. These results are similar to the previous example. Table 8 summarise the results obtained from MATLAB compression and Handel-C compression for other test images.

Table 8. Comparison of software and hardware JPEG compression

Image (256x256)	Compression Ratio		MSE		PSNR (dB)	
	MATLAB	Handel-C	MATLAB	Handel-C	MATLAB	Handel-C
Lena	9.20:1	9.23:1	14.00	13.84	36.67	36.72
Peppers	9.02:1	9.05:1	15.91	16.03	36.11	36.08
House	7.94:1	7.94:1	8.48	8.63	38.85	38.77
Baboon	5.67:1	5.67:1	54.34	54.89	30.78	30.74
Cameraman	7.72:1	7.96:1	24.94	24.69	34.16	34.21
Barbara	7.93:1	7.90:1	25.33	25.46	34.09	34.07

From the Handel-C simulation and MATLAB results it can be seen that the software and hardware based algorithms did have small differences in the compressed images as a result of simplifying the arithmetic in hardware. However, these differences were small, with no discernible difference in image quality between hardware and software compressed images.

9

Final Implementation

This chapter gives the detailed implementation on the FPGA. The steps involved in mapping of the Handel-C into DE2-115 board are described. This chapter presents the initial performance results and discusses the issues of initial implementation. It also optimises the latency and resources used and presents the results for the final implementation. Then it compares the results with an existing implementation from the literature.

Handel-C was adopted as the high-level language for implementing the JPEG compression system. The architecture and behaviour of each module were represented using Handel-C. After the function of the design had been tested successfully within the Handel-C functional simulator, the design was then targeted to a Cyclone IV FPGA on an Altera DE2-115 development board.

9.1 Overview of mapping Handel-C on DE2-115

In order to map the Handel-C program to FPGA, first the source is compiled to an EDIF netlist. Then the EDIF file is compiled using Quartus II to give a configuration file for the Altera Cyclone IV 4CE115 FPGA device. The timing analysis that forms part of this compilation step checks that the design is capable of running at 50MHz. After modifying the design to meet the timing constraints the configuration file was loaded on to the DE2-115 board.

9.2 Results Validation

A module was written to simulate image capture which streams the image from memory, inserting appropriate blanking and providing synchronisation pulses. A 16×16 image was saved into an internal memory of the FPGA for testing purposes. Then the image was passed on to the JPEG compressor and the output from the compression is stored into a FIFO buffer for transfer to the PC by RS-232. On the PC end, a terminal program saves the transferred image to a file. Then IrfanView and MATLAB were used to decode the resulting compressed file and check its validity.

9.3 Initial Results

The utilisation of the FPGA used on DE2-115 development board for each module is shown in Table 9.

Table 9. Resource utilisation of 4CE115 FPGA device for initial implementation.

	Total combinational functions (114480 total)	Dedicated logic registers (114480 total)	Total Logic cells(114480 total)	Memory bits (3981312 total)	M9Ks	Latency (clock cycles)
Image Capture	79	63	102	2048	1	
Block Processing	235	68	248	2048	16	8 rows
Row DCT	1578	201	1619	18	1	10
Transpose buffer	30	14	30	1536	1	64
Column DCT	1978	252	2034	26	1	10
Zigzag coding	191	12	178	1792	1	64
Quantisation	344	28	288	512	1	1
Huffman Coding	6777	6870	10209	0	0	5
Headers	52	56	78	2624	1	
FIFO output	43	18	60	8192	1	
JPEG compressor (including block proc to Headers)	**11185 9.8%**	**7501 6.6%**	**14684 12.8%**	**8556**	**225.1%**	**154**

The initial system used 12.8% of total logic cells, 5.1% of the memory blocks, no multipliers and 154 clock cycles plus 8 image rows of latency. The system was not capable of running at a speed of 50MHz. The maximum clock speed was only 23.27MHz. Then the system was optimised in order to minimise the logic resources, minimise the latency, and to maximise the speed. The next section gives details on optimisation methods for each module.

9.4 Optimisation and Discussion

9.4.1 Image Capture

In this implementation to enable proper testing of the algorithm, the image capture module only simulates image capture by streaming an image from memory. It uses 1 memory block (M9K) to hold the "captured" image into internal memory. In practice, the data input would be streamed directly from the camera.

Similarly, the output FIFO was used purely to hold the compressed image for RS 232 transmission to the PC for checking. Therefore image capture module and FIFO output are not included in the total resource requirements for the compressor.

9.4.2 Block Processing

The latency for initial block processing was 8 rows. However, it is not necessary to wait until all 8 rows are available before beginning the DCT. The first block only requires 8 pixels on the 8th row. Since the DCT will start with the first row of the block, it is safe to start the DCT when the first pixel in the 8th row arrives. This is shown in Figure 9.1. This improved the latency by 1 row to 7 rows of pixels.

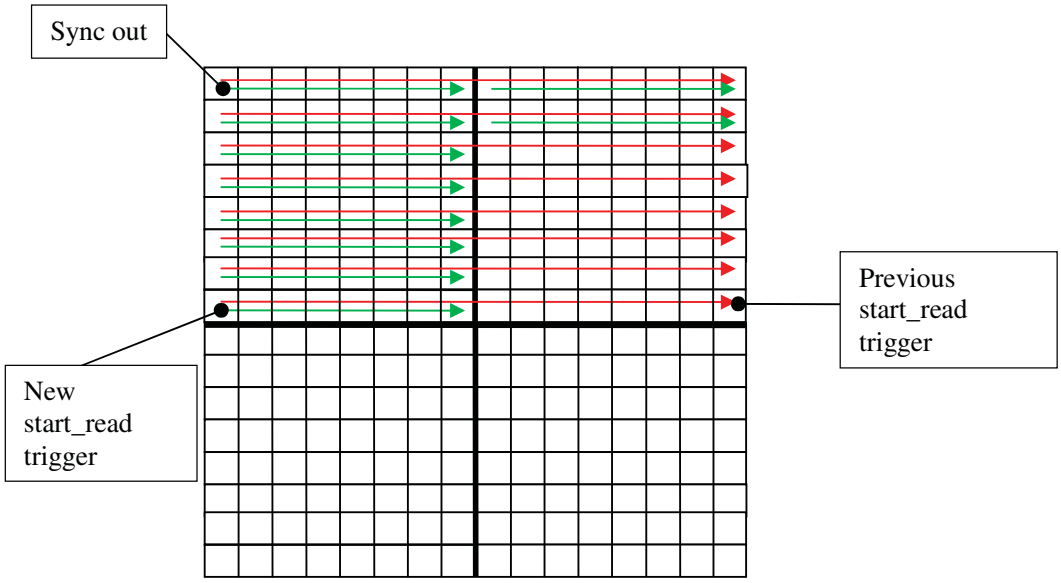


Figure 9.1: New timing diagram for block processing.

From Table 9 it can be noted that block processing module uses 16 memory blocks (M9Ks). The memory in the table is based on a row length of 16 pixels. Since a separate memory is used for each row, the same number of memory blocks would handle row lengths up to 1024 pixels.

9.4.3 Row DCT and Column DCT Modules

In the initial implementation Handel-C compiled the multipliers to LUT logic. Therefore the next step was to minimise the logic usage by using hardware multipliers for the row DCT and column DCT processes since there are plenty of multipliers in FPGA. Table 10 shows that after modifying row DCT and column DCT modules, it significantly dropped the total logic usage of these modules.

Table 10. Comparison between DCT modules.

	Total combinational functions (114480 total)	**Dedicated logic registers (114480 total)**	**Total Logic cells (114480 total)**	**Memory bits (3981312 total)**	**M9Ks**	**Multiplies (532 total)**	**Latency (clock cycles)**
Initial row DCT Module	1578	201	1619	18	1	0	10
Optimised row DCT Module	314	201	353	18	1	8	10
Initial column DCT Module	1978	252	2034	26	1	0	10
Optimised column DCT Module	376	252	445	26	1	8	10

Use of on-chip multipliers has been able to significantly drop the total logic cells from 1619 to 353 for row-DCT module and from 2034 to 445 for column-DCT module. This implementation uses 8 multipliers for each DCT module to multiply the incoming pixel with DCT coefficient. Each multiplier in the DCT uses 2 multiplier blocks on the FPGA. The difference in resources between the row and column DCT is because of the wider bit widths.

9.4.4 Transpose Buffer

The initial transpose buffer had a latency of 64 clock cycles since it waits until the row data is written into the first block during the first 64 cycles. Once the eight row DCTs have computed, the data were streamed out of the transpose buffer in column order for the column DCTs. But it does not have to wait for the full 64 clock cycles to start to read-out. As long as the data in the first column is available when it is needed, read-out can start earlier. Figure 9.2 shows the previous trigger to read-out the data and new trigger. The critical pixel is pixel 0 on the last row. As this is the last pixel in the first column, it is possible to begin read-out part way through the previous row.

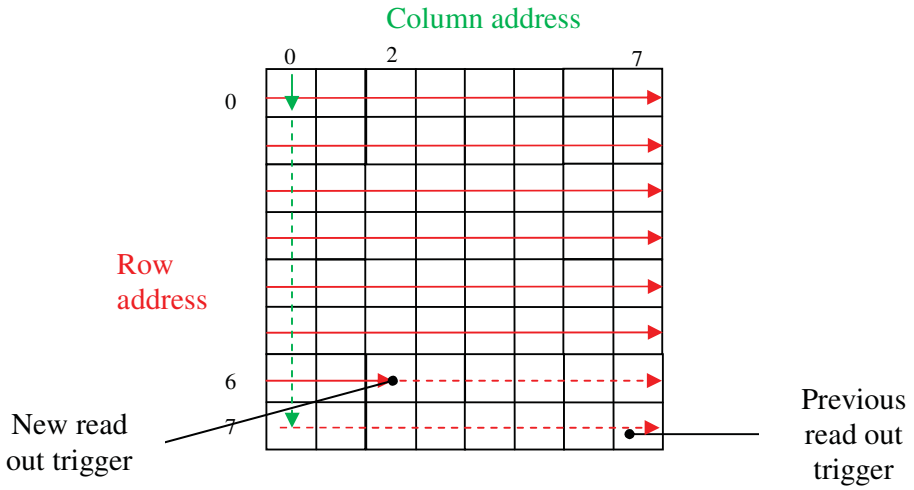


Figure 9.2: Timing diagram for transpose buffer

With this new implementation, read-out from the transpose buffer starts when the coefficient is written to (6, 2). This way it will save 14 clock cycles. Therefore the new transpose buffer has a latency of 50 clock cycles.

9.4.5 Zigzag Buffer

The initial zigzag buffer had a latency of 64 clock cycles. This behaved similarly to the transpose buffer. Considering the transpose, even and odd grouping from DCT module, Figure 9.3 shows the relative timing between writing each coefficient into the buffer and reading it out. The green numbers in the top left corners of Figure 9.3

indicate the order that coefficients are written to the zigzag buffer and the red numbers in the bottom right corners indicate the order that the coefficients are read out from the zigzag buffer. The middle black number represents the difference, if the write-in address is bigger than the read-out address. The largest difference between these represents the minimum latency. In initial implementation, it waited for 64 clock cycles before reading. With the new implementation we only need to wait for 34 clock cycles and start to read out from the next clock cycle. In other words, when it writes the 35th coefficient into the buffer, it will start to readout the 1st coefficient. In this way zigzag buffer will save 29 clock cycles. Therefore, the new implementation has a latency of 35 clock cycles.

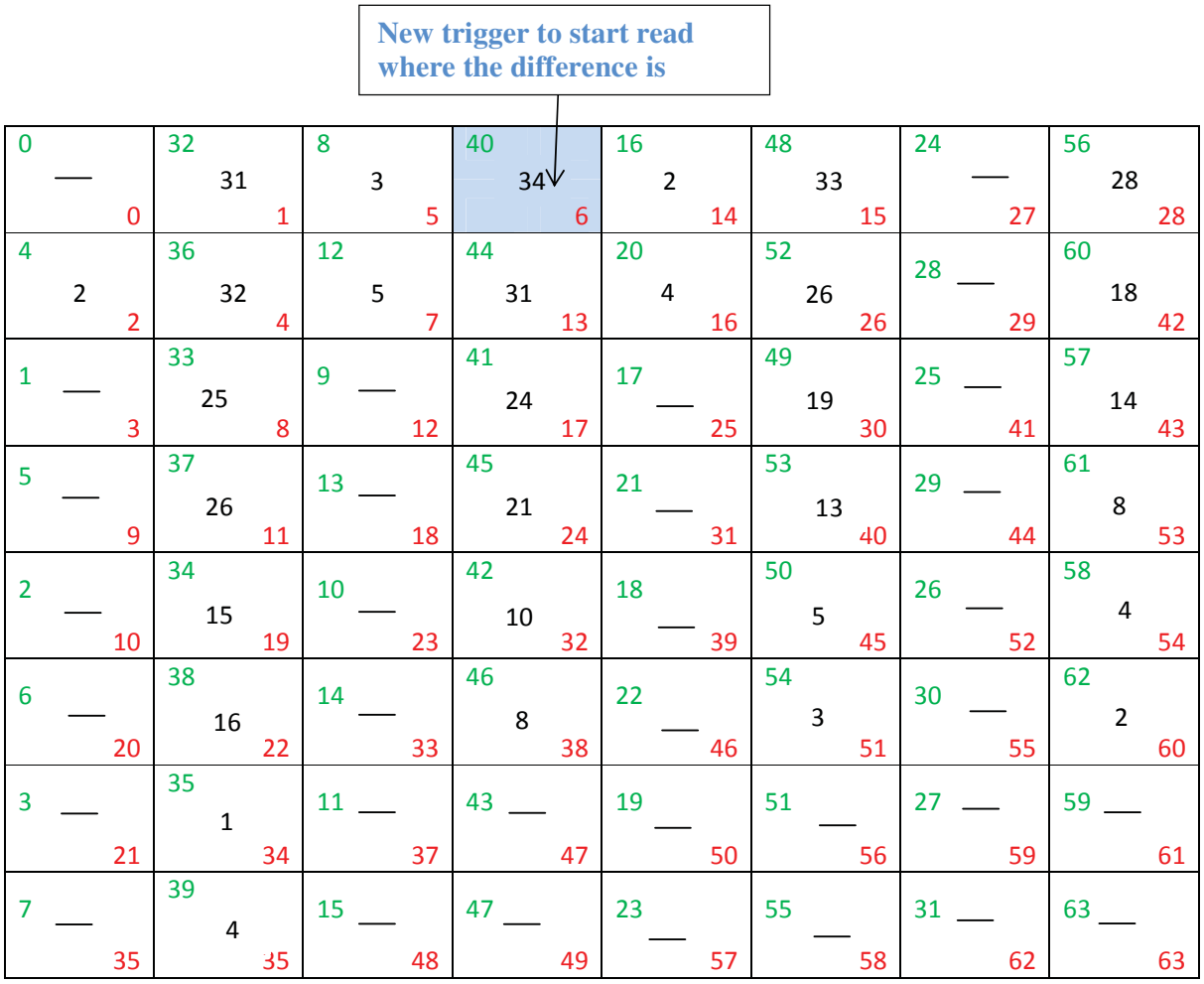


Figure 9.3 : Zigzag buffer addressing and timing. The green numbers represent the clock cycle (relative to the start of the block) that a particular coefficient is written to the buffer. The red number represents the clock cycle that a coefficient is read out.

9.4.6 Quantisation Module

The bottleneck in processing speed for the initial implementation was traced to the divider in the quantisation module. Since the memory is synchronous, the coefficients are only available at the middle of the clock cycle. Therefore the reading coefficients from memory will take half a clock cycle rather than the full clock cycle. This was sped up to the desired clock speed by pipelining the memory read and beginning it one clock cycle earlier, so that the divisor is available at the start of the clock cycle to do the division, and then pipelining the division over 4 clock cycles. This also increased the overall latency by 3 clock cycles.

Table 11. Comparison between quantisation modules.

	Total combinational functions (114480 total)	**Dedicated logic registers (114480 total)**	**Total Logic cells (114480 total)**	**Memory bits (3981312 total)**	**M9Ks**	**Latency (clock cycles)**
Initial Quantisation Module	344	28	288	512	1	1
Optimised Quantisation Module	238	82	274	512	1	4

After pipelining the quantisation operation over four clock cycles, it has increased the number of logic registers from 28 to 82. This is because pipelining required additional registers between the pipeline stages. However this step speeds up the compressor to run at 55.77MHz.

9.4.7 Huffman Coding

From Table 9 it can be noted that the Huffman coding module uses a relatively large number of logic cells compared to the other modules. This is because of the FIFO buffers used in this implementation. A short FIFO buffer is used to store the results from the run count and the size detector to allow up to 3 ZRL codewords to be saved. Another large FIFO saves up to 3 output bytes for each symbol to enable them to be streamed out 8 bits at a time. The initial implementation has used registers to create these two FIFO buffers and those use a large number of logic cells. The logic

resources are significantly reduced by implementing FIFOs using dual port memory. Table 12 shows that by using two separate dual port memories (2 M9Ks) for the FIFOs this has dropped the total logic cells from 10209 to 1317.

Table 12. Comparison between Huffman coding modules

	Total combinational functions (114480 total)	**Dedicated logic registers(114480 total)**	**Total Logic cells(114480 total)**	**Memory bits (3981312 total)**	**M9Ks**	**Latency (clock cycles)**
Initial Huffman Module	6777	6870	10209	0	0	5
Optimised Huffman Module	1310	136	1317	6728	2	5

9.4.8 Final results

Table 13 summarises the FPGA logic usage for the final implementation for each module.

Table 13. Resource utilisation for final implementation.

	Total combinational functions (114480 total)	**Dedicated logic registers(114480 total)**	**Total Logic cells(114480 total)**	**Memory bits (3981312 total)**	**M9Ks (432 total)**	**Multiplies (532 total)**	**Latency (clock cycles)**
Image Capture	79	63	102	2048	1	0	
Block Processing	235	68	248	2048	16	0	7 rows
Row DCT	314	201	353	18	1	8	10
Transpose buffer	36	14	27	1536	1	0	50
Column DCT	376	252	445	26	1	8	10
Zigzag coding	206	12	192	1792	1	0	35
Quantisation	238	82	274	512	1	0	4
Huffman Coding	1310	136	1317	6728	2	0	5
Headers	60	33	62	2624	1	0	
FIFO output	43	18	60	8192	1	0	
JPEG Compressor (including block proc to headers)	**2775 2.4%**	**798 0.7%**	**2918 2.5%**	**15284**	**24 5.5%**	**16 3%**	**114**

The final JPEG compressor uses 2.5% of logic cells, 5.5% of the memory blocks, 3% of multipliers and has a latency of 114 clock cycles, plus 7 image rows compared to the initial system's 12.8% of logic resources, 5.1% of the memory blocks, no multipliers and 154 clock cycles, plus 8 image rows of latency. The final implementation has also improved the maximum speed that it can run at to 55.77MHz where in the initial implementation the maximum clock speed was only 23.27MHz.

9.4.9 Alternative Approach

An alternative optimisation that can be considered is to apply the row-DCT directly on the data as it is streamed from the camera. The block processing can then stream out the data in column order eliminating the transpose buffer. Figure 9.4 shows the block diagrams of the current implementation and suggested implementation. This will reduce the latency by 50 clock cycles from the transpose buffer. Also we can reduce the latency for the row-DCT by 7 clock cycles because the read-out can start before the row DCT has finished. With this design we can start processing and reading out at the 3rd clock cycle. So this will drop the latency by 57 clock cycles and make the latency of the new system to 56 clock cycles (113 – 50 – 7) plus 7 rows. Even though it significantly drops the latency, this implementation requires 50% more memory bits because of the wider data width after the row DCT. The row pixels require only 8 bits per pixel whereas, after the row-DCT, 12 bits per pixel are required. Which design to use would depend on whether memory resources or latency is more critical.

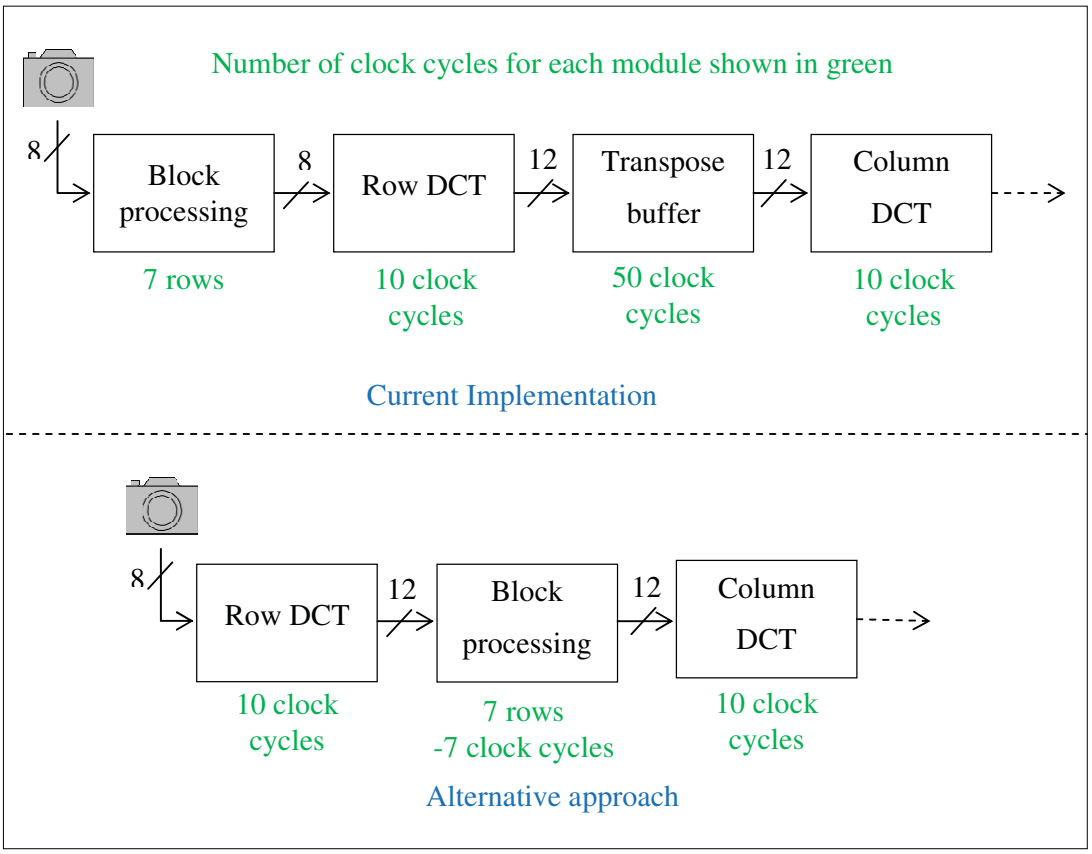


Figure 9.4: Comparison between the current approach and an alternative approach.

Table 14 shows the results from Agostini et al. [10]. This system was implemented on Altera FLEX10KE device.

Table 14. Synthesis results from Agostini et al. [10]

	Logic Cells	Memory (bits)	Frequency (MHz)	Latency (cycles)
2-D DCT	3,670	1,536	38.0	163
Quantization	296	768	49.0	4
Zigzag Buffer	86	1,280	133.3	66
Entropy Coder	580	3,852	45.7	5*
JPEG Compressor	**4,568**	**7,436**	**37.6**	**238***

Altera FLEX10KE device: EPF10K130EQC240-1 *minimal latency

In Agostini's implementation it can be seen that the block processing module and headers module were not implemented because they worked on images saved in

memory and therefore could read the pixels in block order. Considering these factors, my implementation (1D-DCT module to Huffman Coding module) uses a total of 2608 logic cells and 10612 memory bits. When we compare the results obtained from the final implementation with the results from Agostini et al. [10] (see Table 15), it clearly shows that the proposed implementation achieves the minimum latency of 114 clock cycles and with a minimum of 2608 logic cells.

Table 15. Comparison between two compressors

	Logic Cells	Memory (bits)	Frequency (MHz)	Latency (clock cycles)
Our JPEG Compressor	2608	10612	55.77	114
Agostini el al. JPEG Compressor	4568	7436	37.6	238

Even though this implementation uses more memory bits than Agostini's implementation, the number of logic cells required for this implementation is less than that required for Agostini's implementation. Our system has also increased the speed to 55.77MHz.

10

Overall Conclusions

This chapter presents the overall conclusion of this research and suggests possible future work.

10.1 Conclusion

This thesis presents the design of an FPGA implementation of a JPEG compressor for 8 bit grey scale images. The main aim of this thesis was to explore the implementation of JPEG compression onto an FPGA as the data are streaming from the camera while minimising the logic resource requirement of the FPGA and the latency at each stage of the JPEG compression. The JPEG compressor architecture for each module is presented. The JPEG compressor was designed in a fully pipelined fashion. Each JPEG module was firstly tested on its own and then checked for the functionality to validate results in Handel-C simulations. Standard test images were used as benchmarks to test functionality of the design and the results were validated using MATLAB. IrfanView was also used to check that the resulting compressed file could be successfully decoded.

From the simulation results it was confirmed that software and hardware based algorithms did have small differences in the compressed images as a result of simplifying the arithmetic in hardware. However, these differences were small, with no discernible difference in image quality between hardware and software compressed images. The mean square error between MATLAB compressed image and Handel-C compressed image was less than 0.9 pixel value.

Then the JPEG compression algorithm has been successfully implemented and tested on Altera DE2-115 development board. The initial implementation used 12.8% of logic cells, 5.1% of the memory blocks, no multipliers and 154 clock cycles plus 8 image rows of latency. It had a maximum clock speed of 23.27MHz. It was not capable of running at a speed of 50MHz. The system was then optimised in order to minimise the logic resources and to maximise the speed. The final JPEG compressor uses 2.5% of logic cells, 5.5% of the memory blocks, 3% of multipliers and had a latency of 114 clock cycles, plus 7 image rows. It has also improved the speed that it can run at to 55.77MHz.

The final implementation also showed that using FPGA multipliers for 1D-DCT and 2D-DCT, pipelining the quantisation module over four clock cycles, using memory for FIFO buffers in the Huffman coding module, and triggering the read-out at the correct place for block processing, the transpose buffer and zigzag buffer can significantly reduce the logic resources and latency. It has also showed that latency can be further improved to 56 clock cycles at the expense of more logic resources.

The results obtained from this implementation were very satisfactory. This research confirmed that JPEG compression can be implemented on streamed data from a camera at a cost of minimum logic cells and at a faster clock speed. Therefore in future this implementation can be useful for any embedded system image compression application.

10.2 Future Work

The research work presented in this thesis is a solid foundation for the JPEG compression on FPGAs for grey scale images and it confirmed that the hardware implementation for this is possible when data are streaming from the camera.

In future, the proposed architectures for each JPEG module can be used to extend the current design to colour images and other compression algorithms. Also there are few optimisations which can be performed to improve the use of logic cell.

Quality Factor: The quantisation tables can be scaled according to a quality factor selected by the user. This has a direct impact on compression ratios. This will allow a user to set the desired level of compression for a given application.

Colour images: The hardware described in this thesis is suitable for the luminance components. A similar circuit would also be required for the chrominance components. Since the chrominance components are down-sampled, one additional set of hardware could be used for both chrominance components. Alternatively, if the clock speed was set 50% higher than the pixel rate, then the luminance hardware

could be reused for all components. Colour images would also require converting the RGB to YUV, and an increase in the buffer size for the chrominance.

References

[1] V. A. M. Prakash and K. S. Gurumurthy, "A Novel VLSI Architecture for Digital Image Compression using Discrete Cosine Transform and Quantisation," *International Journal of Computer Science and Network Security,* vol. 10, pp. 175-182, September 2010.

[2] International telecommunication union, "Information technology – Digital compression and coding of continuous-tone still images: Requirements and guidelines," in *ISO/IEC 10918-1* vol. T.81, ed, 1993.

[3] C. Christopoulos, A. Skodras, and T. Ebrahimi, "The JPEG2000 still image coding system: an overview," *IEEE Transactions on Consumer Electronics, ,* vol. 46, pp. 1103-1127, 2000.

[4] Compuserve, "Graphics Interchange Format Specification," ed. Columbus,: OH: CompuServe, Inc.

[5] T. L. Thomas Boutell, "Portable Network Graphics (PNG) Specification and Extensions," vol. 1, The Internet Engineering Task Force (IETF), 1996.

[6] S. Gordoni, "Investigation of Hardware JPEG Encoder Implementation and Verification Methodologies," Department of Electrical and Computer Engineering, University of California Santa Barbara, 2006.

[7] S. A. K. Jilani and S. A. Sattar, "JPEG Image Compression using FPGA with Artificial Neural Networks," *International Journal of Engineering and Technology,* vol. 2, pp. 252-257, 2010.

[8] J. Ahmad, K. Raza, M. Ebrahim, and U. Talha, "FPGA based implementation of baseline JPEG decoder," *7th International Conference on Frontiers of Information Technology*, Abbottabad, Pakistan, 2009.

[9] C. Johnston, D. Bailey, and P. Lyons, "A Visual Environment for Real-Time Image Processing in Hardware (VERTIPH)," *EURASIP Journal on Embedded Systems,* vol. 2006, p. 072962, 2006.

[10] L. V. Agostini, I. S. Silva, and S. Bampi, "Multiplierless and fully pipelined JPEG compression soft IP targeting FPGAs," *Microprocess. Microsyst.,* vol. 31, pp. 487-497, 2007.

[11] D. G. Bailey, *Design for Embedded Image Processing on FPGAs*: John Wiley & Sons, 2011.

[12] S.-H. Sun and S.-J. Lee, "A JPEG Chip for Image Compression and Decompression," *The Journal of VLSI Signal Processing,* vol. 35, pp. 43-60, 2003.

[13] M. Kovac and N. Ranganathan, "JAGUAR: a fully pipelined VLSI architecture for JPEG image compression standard," *Proceedings of the IEEE,* vol. 83, pp. 247-258, 1995.

[14] R. Uma, "FPGA Implementation of 2-D DCT for JPEG Image Compression," *International Journal of Advanced Engineering Sciences and Technologies (IJAEST),* vol. 7, pp. 1-9, 2011.

[15] H. Anas, S. Belkouch, M. El Aakif, and N. Chabini, "FPGA implementation of a pipelined 2D-DCT and simplified Quantisation for real-time applications," in *International Conference on Multimedia Computing and Systems (ICMCS),* pp. 1-6, 2011.

[16] L. V. Agostini, I. S. Silva, and S. Bampi,, "Pipelined fast 2D DCT architecture for JPEG image compression," in *14th Symposium on Integrated Circuits and Systems Design*, pp. 226-231, 2001.

[17] C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical fast 1-D DCT algorithms with 11 multiplications," in *International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, pp. 988-991, 1989.

[18] Mentor Graphics, "Handel-C Language Reference Manual," version 5.3_2, Mentor Graphics Corporation, 2010.

[19] D.-U. Lee, "Hardware Compilation and Resource Sharing Optimisations," BEng Information Systems Engineering Final year project report, Imperial College, London, 2001.

[20] B. Bober. (2008, 30th March). *Altera and Xilinx Report: The Battle Continues* Available: http://seekingalpha.com/article/85478-altera-and-xilinx-report-the-battle-continues

[21] B. S. Darade, T. A. Parmar, and A. S. Chauhan, "Programming FPGA's Using Handel-C," presented at SHAASTRA, T.E. Electronics and Telecomm, 2003.

[22] J. Hawkins, "A Framework for Refining Functional Specification into Parallel Reconfigurable Hardware Implementations," PhD thesis, Department of Computer Science, The University of Reading, United Kingdom, 2005.

[23] R. Woods and J. McAllister, *FPGA-based implementation of signal processing systems*: John Wiley & Sons, 2008.

[24] M. A. Figueiredo and C. Gloster, "Implementation of a probabilistic neural network for multi-spectral image classification on an FPGA based custom computing machine," *Vth Brazilian Symposium on Neural Networks*, pp. 174-179, 1998.

[25] M. Naghmash, M. F. Ain, and C. Y. Hui, "FPGA Implementation of Software Defined Radio Model based 16QAM," *European Journal of Scientific Research,* vol. 35, pp. 301-310, 2009.

[26] F. C. J. Allaire, *FPGA implementation of an unmanned aerospace vehicle path planning genetic algorithm*. Royal Military College: Library and Archives, Canada, 2007.

[27] R. M. Jiang and D. Crookes, "FPGA implementation of 3D discrete wavelet transform for real-time medical imaging," *18th European Conference on Circuit Theory and Design (ECCTD)*, pp. 519-522, 2007.

[28] Z. Ge, Y. Jinghua, L. Qian, and Y. Chao, "A real-time speech recognition system based on the Implementation of FPGA," *Cross Strait Quad-Regional Radio Science and Wireless Technology Conference (CSQRWC)*, pp. 1375-1378, 2011.

[29] S. A. Kadir, A. Sasongko, and M. Zulkifli, "Simple power analysis attack against elliptic curve cryptography processor on FPGA implementation," *International Conference on Electrical Engineering and Informatics (ICEEI)*, pp. 1-4, 2011.

[30] H. M. Hussain, K. Benkrid, H. Seker, and A. T. Erdogan, "FPGA implementation of K-means algorithm for bioinformatics application: An accelerated approach to clustering Microarray data,"*NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 248-255, 2011.

[31] Y. Abhyankar, C. Sajish, P. Kulkarni, and C. R. Subrahmanya, "Design of a FPGA based data acquisition system for radio astronomy applications," *The 16th International Conference on Microelectronics (ICM)*, pp. 555-557, 2004

[32] ALTERA, "DE2-115 User Manual," ed: Terasic Technologies Inc, 2003.

[33] I. S. Uzun and A. Amira, "RAPID PROTOTYPING - Framework for FPGA-based discrete biorthogonal wavelet transforms implementation," *IEE Proceedings on Vision, Image and Signal Processing,* vol. 153, pp. 721-734, 2006.

[34] D. Bailey, "Implementing Image Processing Algorithms on FPGAs," in *Course Notes*, Massey University: IEEE NZ Central Section Workshop, 2011.

[35] W. Kou, *Digital Image Compression: Algorithms and Standards*: Springer, 1995.

[36] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, 2nd ed. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[37] R. M. Haralick, "A Storage Efficient Way to Implement the Discrete Cosine Transform," *IEEE Transactions on Computers,* vol. C-25, pp. 764-765, 1976.

[38] T. C. Chen, M. T. Sun, and A. M. Gottlieb, "VLSI implementation of a 16×16 DCT," *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 1973-1976, 1988.

[39] Z. Cvetkovic and M. V. Popovic, "New fast recursive algorithms for the computation of discrete cosine and sine transforms," *IEEE Transactions on Signal Processing,* vol. 40, pp. 2083-2086, 1992.

[40] R. Woods, D. Trainor, and J. P. Heron, "Applying an XC6200 to real-time image processing," *IEEE Design & Test of Computers,* vol. 15, pp. 30-38, 1998.

[41] D. G. Bailey, "Space Efficient Division on FPGAs," presented at the Electronics New Zealand Conference (EnzCon'06), Christchurch, NZ, pp. 206-211, 2006.

[42] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete Cosine Transform," *IEEE Transactions on Computers,* vol. C-23, pp. 90-93, 1974.

[43] I. E. G. Richardson, *Video Codec Design: Developing Image and Video Compression Systems*: Wiley, 2002.

[44] R. C. González and R. E. Woods, *Digital Image Processing*: Pearson/Prentice Hall, 2008.

[45] H. Hsieh, "A fast recursive algorithm for computing the discrete cosine transform," *IEEE Transactions on Acoustics, Speech and Signal Processing,* vol. 35, pp. 1455-1461, 1987.

[46] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "The Multiple Wordlength Paradigm," *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, pp. 51-60, 2001.

[47] J. P. Deschamps and G. Sutter, "Decimal division: Algorithms and FPGA implementations," *VI Southern Programmable Logic Conference (SPL)*, pp. 67-72, 2010.

[48] E. Hamilton, " JPEG File Interchange Format (Version 1.02)," 1992.

[49] K. Sakiyama, P. Schaumont, I. Verbauwhede, "Finding the best system design flow for a high-speed JPEG encoder," *Asia and South Pacific Design Automation Conference (ASP-DAC 2003)*, pp. 577-578, Kitak Yushu, Japan, January 2003.

Appendix A-1: AC Huffman Table

Tables below show the default coding for run-length and size for AC coefficients (from [2]).

Run/Size	Code length	Code Word
0/0 (EOB)	4	1010
0/1	2	00
0/2	2	01
0/3	3	100
0/4	4	1011
0/5	5	11010
0/6	7	1111000
0/7	8	11111000
0/8	10	1111110110
0/9	16	1111111110000010
0/A	16	1111111110000011
1/1	4	1100
1/2	5	11011
1/3	7	1111001
1/4	9	111110110
1/5	11	11111110110
1/6	16	1111111110000100
1/7	16	1111111110000101
1/8	16	1111111110000110
1/9	16	1111111110000111
1/A	16	1111111110001000
2/1	5	11100
2/2	8	11111001
2/3	10	1111110111
2/4	12	111111110100
2/5	16	1111111110001001
2/6	16	1111111110001010
2/7	16	1111111110001011
2/8	16	1111111110001100
2/9	16	1111111110001101
2/A	16	1111111110001110
3/1	6	111010
3/2	9	111110111
3/3	12	111111110101
3/4	16	1111111110001111
3/5	16	1111111110010000
3/6	16	1111111110010001

Run/Size	Code length	Code Word
3/7	16	1111111110010010
3/8	16	1111111110010011
3/9	16	1111111110010100
3/A	16	1111111110010101
4/1	6	111011
4/2	10	1111111000
4/3	16	1111111110010110
4/4	16	1111111110010111
4/5	16	1111111110011000
4/6	16	1111111110011001
4/7	16	1111111110011010
4/8	16	1111111110011011
4/9	16	1111111110011100
4/A	16	1111111110011101
5/1	7	1111010
5/2	11	11111110111
5/3	16	1111111110011110
5/4	16	1111111110011111
5/5	16	1111111110100000
5/6	16	1111111110100001
5/7	16	1111111110100010
5/8	16	1111111110100011
5/9	16	1111111110100100
5/A	16	1111111110100101
6/1	7	1111011
6/2	12	111111110110
6/3	16	1111111110100110
6/4	16	1111111110100111
6/5	16	1111111110101000
6/6	16	1111111110101001
6/7	16	1111111110101010
6/8	16	1111111110101011
6/9	16	1111111110101100
6/A	16	1111111110101101
7/1	8	11111010
7/2	12	111111110111
7/3	16	1111111110101110
7/4	16	1111111110101111
7/5	16	1111111110110000
7/6	16	1111111110110001
7/7	16	1111111110110010
7/8	16	1111111110110011
7/9	16	1111111110110100
7/A	16	1111111110110101

Run/Size	Code length	Code Word
8/1	9	111111000
8/2	15	111111111000000
8/3	16	1111111110110110
8/4	16	1111111110110111
8/5	16	1111111110111000
8/6	16	1111111110111001
8/7	16	1111111110111010
8/8	16	1111111110111011
8/9	16	1111111110111100
8/A	16	1111111110111101
9/1	9	111111001
9/2	16	1111111110111110
9/3	16	1111111110111111
9/4	16	1111111111000000
9/5	16	1111111111000001
9/6	16	1111111111000010
9/7	16	1111111111000011
9/8	16	1111111111000100
9/9	16	1111111111000101
9/A	16	1111111111000110
A/1	9	111111010
A/2	16	1111111111000111
A/3	16	1111111111001000
A/4	16	1111111111001001
A/5	16	1111111111001010
A/6	16	1111111111001011
A/7	16	1111111111001100
A/8	16	1111111111001101
A/9	16	1111111111001110
A/A	16	1111111111001111
B/1	10	1111111001
B/2	16	1111111111010000
B/3	16	1111111111010001
B/4	16	1111111111010010
B/5	16	1111111111010011
B/6	16	1111111111010100
B/7	16	1111111111010101
B/8	16	1111111111010110
B/9	16	1111111111010111
B/A	16	1111111111011000
C/1	10	1111111010
C/2	16	1111111111011001
C/3	16	1111111111011010
C/4	16	1111111111011011

Run/Size	Code length	Code Word
C/5	16	1111111111011100
C/6	16	1111111111011101
C/7	16	1111111111011110
C/8	16	1111111111011111
C/9	16	1111111111100000
C/A	16	1111111111100001
D/1	11	11111111000
D/2	16	1111111111100010
D/3	16	1111111111100011
D/4	16	1111111111100100
D/5	16	1111111111100101
D/6	16	1111111111100110
D/7	16	1111111111100111
D/8	16	1111111111101000
D/9	16	1111111111101001
D/A	16	1111111111101010
E/1	16	1111111111101011
E/2	16	1111111111101100
E/3	16	1111111111101101
E/4	16	1111111111101110
E/5	16	1111111111101111
E/6	16	1111111111110000
E/7	16	1111111111110001
E/8	16	1111111111110010
E/9	16	1111111111110011
E/A	16	1111111111110100
F/0 (ZRL)	11	11111111001
F/1	16	1111111111110101
F/2	16	1111111111110110
F/3	16	1111111111110111
F/4	16	1111111111111000
F/5	16	1111111111111001
F/6	16	1111111111111010
F/7	16	1111111111111011
F/8	16	1111111111111100
F/9	16	1111111111111101
F/A	16	1111111111111110

Appendix A-2: JPEG Header Specifications

Application Specific Header

The JFIF APP0 marker provides information which is missing from the JPEG stream such as, version number, X and Y density, pixel aspect ratio and thumbnail. Details of APPO marker as follows [2]:

APP0 Header (2 Bytes): Defines the application specific header

Length (2 Bytes): Total length of header

Application Identifier (5 Bytes): Identifies JFIF through ASCII hex codes, 0x4A46494600

Version Identifier (2 Bytes): Specifies major and minor version numbers for JFIF (V1.01)

Density Units (1 Byte): Specifies units used to give pixel density.

0: density given in pixel 1: density in dots per inch 2: density

X Density (2 Bytes): Horizontal pixel density

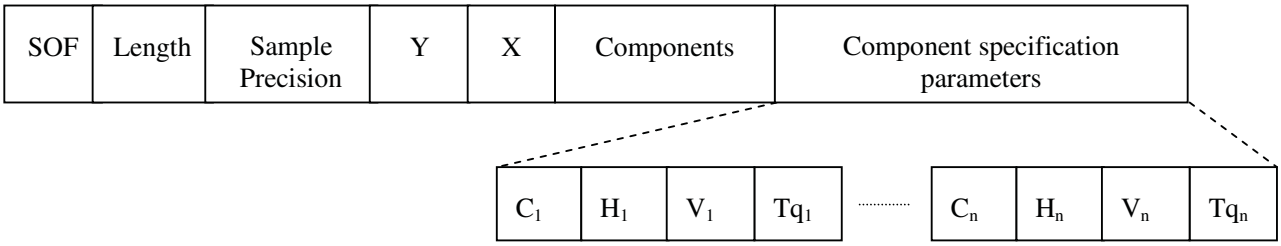
Y Density (2 Bytes): Vertical pixel density

X Thumbnail (4 bits): Specifies the horizontal pixel count of the thumbnail

Y Thumbnail (4 bits): Specifies the vertical pixel count of the thumbnail

Frame Header

Frame header specifies the source image characteristics, and the parameters that apply to all scans with the frame. SOF is the start of frame marker, which is unique for each type of JPEG implementation [2].



SOF (2 Bytes): Define by unique code 0xFFC0

Length (2 Bytes): Defines the length for the frame header

Sample Precision (1 Byte): Define the sample precision used. For baseline, this is 8 bits

Number of lines (Y) (2 Bytes): Specifies the maximum number of lines in the source image

Number of samples per line (X) (2 Bytes): Specifies the maximum number of samples per line in the source image

Components (1 Byte): Specifies number of component in a frame.

1 for gray scale images 3 for colour images

Component Identifier (1 Byte): Defines the identification number for the frame component that is going to be specified

Horizontal sampling (4 Bits): Specifies the relationship between the component horizontal dimension and maximum image dimension X

Vertical sampling (4 Bits): Specifies the relationship between the component vertical dimension and maximum image dimension Y

Quantisation table selector (1 Byte): Defines which Quantisation table will be used for the component

Quantisation Table Header

The Quantisation table header segment is used to define the Quantisation tables for luminance components and chrominance components. For this research it only uses Quantisation table for luminance components [2].

DQT	Length	Element Precision	Table Identifier	Quantisation Element

DQT (2 Bytes): Specifies the beginning of the Quantisation table-specification parameters

Length (2 Bytes): Specifies the length of all Quantisation table parameters

Element Precision (4 Bits): Specifies the precision of the Quantisation element

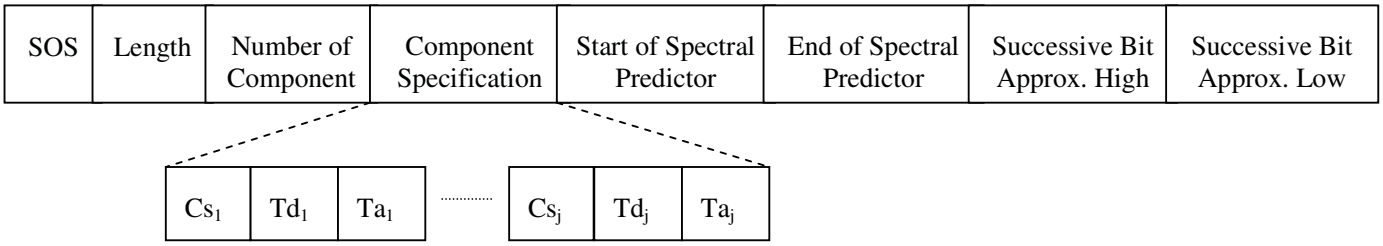
0 = 8 bit precision 1 = 16 bit precision

Table Identifier (4 Bits): Specifies one of four possible destinations at the decoder into which the Quantisation table shall be installed

Quantisation Element (1 or 2 Bytes): Specifies the correct element out of 64 elements. The Quantisation elements shall be specified in zigzag order

Scan Header

The scan header segment defines the parameters for the entropy coded data segment. These parameters specifies which components are contained in the scan, specifies the destinations from which entropy tables to be used with each component that are retrieved [2].



SOS (2 Bytes): Start of scan marker marks the beginning of the scan parameters.

Length (2 Bytes): Specifies the length of the scan header in bytes.

Number of Scan Components (1 Byte): Defines how many image components are contained within the current scan. This value is equal to the number of sets of scan specifications, which follow in this header segment.

Cs_j - Scan Component Selector (1 Byte): This is the header for a specific component, which will be followed by that component's specified tables. This will match one of the components identified in the frame header.

Td_j – DC Entropy Coding Table Selector (4 bits): This will specify one of four possible DC entropy coding table destinations from which the entropy table needed for decoding of the DC coefficients of component Cs_j is retrieved.

Ta_j – AC Entropy Coding Table Selector (4 bits): This will specify one of four possible AC entropy coding table destinations from which the entropy table needed for decoding of the AC coefficients of component Cs_j is retrieved.

Start of Spectral or Predictor Selection (1 Byte): This is used for other DCT based algorithms for selecting the first DCT coefficient in each zigzag order which shall be coded in the scan. For the Baseline algorithm, all components, 0-63, of the DCT are encoded, so this is set to 0.

End of Spectral Selection (1 Byte): Specifies the last DCT coefficient in each block in zigzag order which shall be coded in the scan. This parameter is set to 63 for baseline algorithm.

Successive Approximation Bit Position High (4 Bits): This specifies the point transform used in the preceding scan. This parameter is set to zero for the first scan of each band of coefficients.

Successive Approximation Bit Position Low (4 Bits): This specifies the point transform used before coding the band of coefficients specified by the spectral selection. This is set to zero for sequential DCT processes.

Entropy Coded Segment

An entropy coded data segment contains the output of an entropy-coding procedure. It consists of an integer number of bytes, whether the entropy-coding procedure used is Huffman or arithmetic [2].

DHT	Length	Table Class	Table Identifier	Number of Huffman Codes	Value

DHT (2 Bytes): Specifies the beginning of the Huffman table-specification parameters

Length (2 Bytes): Specifies the length of all Huffman table parameters

Table Class (4 Bits): Defines the type of the table.

0= DC table 1 =AC table

Table Identifier (4 Bits): Specifies one of four possible destinations at the decoder into which the Huffman table shall be installed

Number of Huffman Codes of length *i* (8 Bits): There are 16 length categories consisting Huffman codes. This defines the number of codes within category i.

Value (8 Bits): This is the value associate with each Huffman code within each length category.

Appendix A-3: Abbreviations:

AC – Alternative Current

ASIC – Application Specific Integrated circuit

CODEC – Coder - Decoder

DC – Direct Current

DCT – Discrete Cosine Transform

DHT – Define Huffman Table

DPCM – Differential Pulse Code Modulation

DVD – Digital Versatile Disk

DQT – Define Quantisation Table

EDIF – Electronic Design Interchange Format

EOB – End of Block

EOI – End of Image

FPGA – Field Programmable Gate Array

FIFO – First In - First out

GIF – Graphic Interchange Format

IDCT – Inverse Discrete Cosine Transform

I/O – Input Output

ITU – International Telecommunication Union

JFIF – JPEG File Interchange Format

JPEG – Joint Photographic Expert Group

LSB – Least Significant Bit

LUT – Look Up Table

MSB – Most Significant Bit

MSE – Mean Square Error

PLD – Programmable Logic Device

PNG – Portable Network Graphics

PROM – Programmable Read Only Memory

PSNR – Peak Signal to Noise Ratio

RAM – Random Access Memory

ROM – Read Only Memory

SOF – Start of Frame

SOI – Start of Image

SOS – Start of Scan

SRT Division –

VHDL – VHSIC hardware description language

VLSI – Very Large Scale Integration

ZRL – Zero Run Length

Appendix A-4: Publications and Presentations

PUBLICATIONS

- De Silva, A. M., Bailey, D. G., & Punchihewa, A., "Exploring the Implementation of JPEG Compression on FPGAs", 6th International Conference on Signal Processing and Communication Systems (ICSPCS), Gold Coast, Australia, 12-14 December, 2012

PRESENTATIONS

- "Challenges in Implementing JPEG compression on FPGAs", EICS Seminar, 11th October 2012, to the School of Engineering students and staff, at Massey University, Palmerston North

- "The Implementation of JPEG compression on FPGAs", IEEE NZ Central Section Post-Graduate Presentation Workshop, to the students in Lower North Island, 7th September 2012, at Massey University, Palmerston North

- "The Implementation of JPEG compression on FPGAs: work-in progress", EICS Seminar, 10th October 2011, to the School of Engineering students and staff, at Massey University, Palmerston North

- "Exploring the Implementation of JPEG compression on FPGAs", IEEE NZ Central Section Post-Graduate Presentation Workshop, to the students in Lower North Island, 1st September 2011, at Victoria University, Wellington

Exploring the Implementation of JPEG Compression on FPGA

Ann Malsha De Silva, Donald G. Bailey, Amal Punchihewa
Massey University, New Zealand

Image Compression—JPEG

In image processing, image compression can improve the performance of the system by reducing the cost and time in image storage and transmission without a significant reduction of the image quality . Image compression can be lossy or lossless. Lossy compression gives a greater reduction in data volume compared to lossless compression. JPEG is an lossy based international standard for still-image compression and it has been widely used since 1987.

FPGA

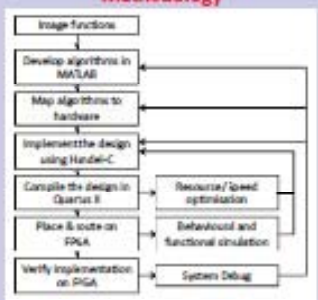
Field Programmable Gate Array (FPGA) s a reprogrammable hardware device. FPGAs are well suited for many embedded systems applications because they have several desirable attributes such as, small size, low power consumption, a large number of I/O ports, and a large number of computational logic blocks . Images have a high degree of spatial parallelism, thus image processing applications are ideally suited to implementation on FPGAs which contain large arrays of parallel logic and registers and can support pipelined algorithms.

Motivation

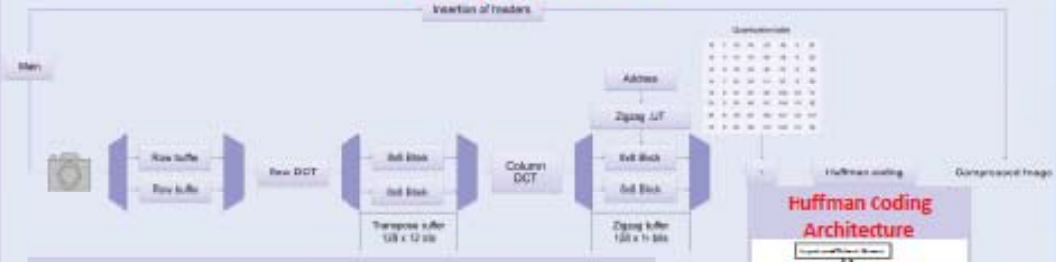
Processing time and power restrictions imposed on dedicated embedded systems make software compression unviable in many applications. Power efficiency and fast compression are often performance critical factors. For most digital image codecs, increasing the compression has been achieved at the cost of increasing the complexity of the techniques and implementations. These restrictions usually mandate a dedicated hardware implementation of a JPEG compressor, especially in applications such as digital cameras, DVD players, traffic controllers, secure ticketing, and many more.

This research focuses on the real-time implementation of JPEG compression on an FPGA device as the data are streamed from a camera while minimizing the logic resources of the FPGA, and the latency at each stage of the JPEG compression.

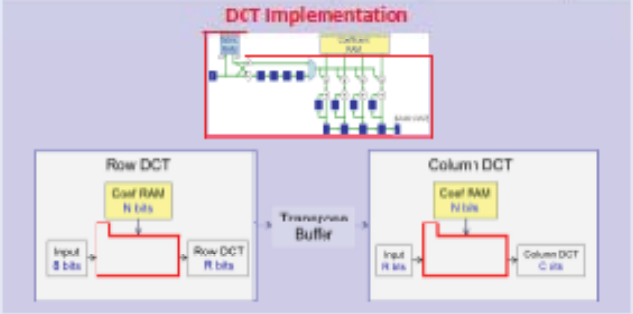
Methodology



Overall System Architecture



DCT Implementation



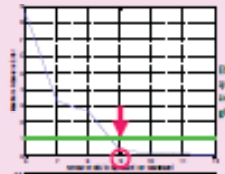
Huffman Coding Architecture

	Latency (cycles)
Header	328
Block Processing	8 rows
1-D DCT	16
Transpose Buffer	64
2-D DCT	10
Quantization	1
ZigZag Transform	64
Entropy Coding	5
JPEG Compressor	154

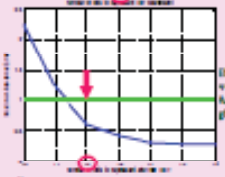
DCT Word Length Optimization

	Sign	Word bits	Binary places
Pixels	Signed*	8	0
DCT Coefficients	Signed	M	M
Sum and Difference	Signed	9	0
After multiplication	Signed	M+9	M
After row DCT	Signed	N+11	M
Want R bits after row DCT. Therefore drop (N+11-R) bits			
After truncation of row DCT	Signed	R	R-11
Sum and Difference	Signed	R+1	R-11
Multiplication	Signed	N+R+1	M+R-11
After column DCT (2D-DCT)	Signed	N+R+9	M+R-11
Want C bits after 2DDCT. Therefore drop (N+R+9-C) bits			
After truncation of 2D-DCT	Signed	C	C-13

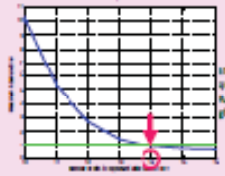
*Unsigned pixel are offset by 128 before DCT
N= Number of bits to represent DCT coefficients
R= Number of bits required after row DCT (1-D DCT)
C= Number of bits required after column DCT (2D-DCT)



Effect of different DCT coefficient quantization with R = 20, and C = 20. Arrow shows quantization which gives less than 1 MAE.



Effect of different row DCT quantization with N= 9, and C = 20. Arrow shows quantization which gives less than 1 MAE.



Effect of different column DCT quantization with N= 9, and R = 12. Arrow shows quantization which gives less than 1 MAE.

Results and Evaluation

Lena Reconstructed Image	Handel-C	MATLAB
MSE	14.08	13.84
PSNR (dB)	36.67	36.72
Compression Ratio	9.23:1	9.20:1

MSE between the two compressed images was only 1.8885

Peppers Reconstructed Image	Handel-C	MATLAB
MSE	16.01	15.91
PSNR (dB)	36.08	36.11
Compression Ratio	9.05:1	9.02:1

MSE between the two compressed images was only 1.8065

House Reconstructed Image	Handel-C	MATLAB
MSE	8.63	8.40
PSNR (dB)	38.77	38.85
Compression Ratio	7.94:1	7.94:1

MSE between the two compressed images was only 1.5590

Conclusion

This poster presents the design of an FPGA implementation of a JPEG compressor for 8 bit grey scale images. The JPEG compressor architecture for each module is presented, that minimises the logic resource of the FPGA and latency at each stage of the JPEG compression. The JPEG compressor was designed in a fully pipelined fashion. Improvements were made by minimising the latency, and increasing the performance. The proposed JPEG compressor architecture has a latency of 154 clock cycles plus 8 image rows.

The software and hardware based algorithms did have small differences in the compressed images as a result of simplifying the arithmetic in hardware. However, these differences were small, with no discernible difference in image quality between hardware and software compressed images.