An FPGA Based Hardware Accelerator for Remote Surveillance Cameras

by

Alexander John Petre Kane

A thesis
submitted to Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Engineering
in Electronic and Computer Systems Engineering.

Victoria University of Wellington
2013

Abstract

The Blackeye II camera, produced by Kinopta, is used for remote security, conservation and traffic flow surveillance. The camera uses an image sensor to acquire photographs which undergo image processing and JPEG encoding on a microprocessor. Although the microprocessor performs other tasks, it is the processing and encoding of images that limit the frame rate of the camera to 2 frames per second (fps). Clients have requested an increase to 12.5 fps while adding more image processing to each photograph. The current microprocessor-based system is unable to achieve this.

Custom digital logic systems perform well on processes that naturally form a pipeline, such as the Blackeye II image processing system. This project develops a digital logic system based on an FPGA to receive images from the image sensor, perform the required image processing operations, encode the images in JPEG format and send them on to the microprocessor. The objective is to implement a proof of concept device based upon the Blackeye II's existing hardware and an FPGA development board. It will implement the proposed pipeline including one example of an image processing operation.

A JPEG encoder is designed to process the 752×480 greyscale photographs from the image processor in real time. The JPEG encoder consists of four stages: discrete cosine transform (DCT), quantisation, zig-zag buffer and Huffman encoder. The DCT design is based upon the work of Woods et al. [1], which is improved on. An analysis of the relationship between precision and accuracy in the DCT and quantisation stages is used to minimise the system's resource requirements. The JPEG encoder is successfully tested in simulation.

Input and output stages are added to the design. The input stage receives data from the image sensor and removes breaks in the data stream. The output stage must concatenate the data from the JPEG encoder and transmit it to the microprocessor via the microprocessor's ISI (image sensor interface) peripheral. An image sharpening filter is developed and inserted into the pipeline between the input and JPEG encoder. Because remote surveillance cameras are battery powered, the minimisation of power consumption is a key concern. To minimise power consumption a mechanism is introduced to track those modules in the pipeline that are in use at any time. Any not in use are paused by gating the module's clock source.

Once the system is complete and tested in simulation it is loaded into hardware. The FPGA development board is attached to the image sensor board and microprocessor board of the Blackeye II camera by a purpose-built breakout board. Plugging the microprocessor board into a PC provides a live stream of images proving the successful operation of the FPGA system. The project objectives were exceeded by increasing the frame rate of the Blackeye II to 20 fps, which will not decrease with additional image processing operations.

The project was viewed as a success by Kinopta, who have committed to its further development.

Acknowledgments

I am indebted to my supervisor, Prof. Dale Carnegie, for his support in this endeavour. He shared his wealth of experience in research and writing in guiding me to the completion of this thesis.

Much thanks is owed to Don Peat of Kinopta, without whom this project would not have been possible. He and Rick Hudson welcomed me into the team and provided financial, technical and moral support.

Victoria University Scholarships Office provided financial support when the initial funding of the project fell through.

This thesis would make for a poorer read if not for the many people who gave me feedback on it along the way. I especially would like to thank my mother, Jane Petre, who knows more about grammar than anyone else, Ann Hibbard and Dayna Kivell for telling me when my explanations were not up to scratch, Tina Rønhovde Tiller for her meticulous proofing, and Rory Sarten and my father, John Kane, for undertaking the Herculean task of reading the thesis from cover to cover in one go.

Thanks also go to all my friends and family for their support throughout, and for putting up with me hiding away in my writing cave and not coming out to play.

Contents

Chapter 1

Introduction

Remote surveillance cameras provide a means to observe places that are not easily accessible. For example, to monitor access points, farms that are vulnerable to trespassers require remote cameras that don't depend on the electric grid or fixed communication systems. The applications for such devices extend well beyond rural security. The conservation of native fauna is a major issue in New Zealand. Conservation workers are too stretched to watch out for every animal in their care, but using remote cameras allows them to observe the animals' behaviour, find out where they've been, or if pest species are in the area.

Kinopta is a small New Zealand company whose flagship product is the Blackeye II camera. This camera is designed to operate for days at a time on a single battery charge. It takes photographs regularly for the entire period of operation. When the batteries are low, the camera is accessed and the batteries changed or recharged. When it is accessed by a PC, either via USB or wirelessly, it presents itself as a website for the user to access, browse and download the photos.

A major strength of the Blackeye II as a remote surveillance camera is its unobtrusive night-time photography. It takes high quality photos in dark conditions without the use of any flash perceptible to human or animal. Most locations where remote surveillance cameras are used are

devoid of artificial light, so many competing cameras struggle to perform at night. To increase battery life other camera systems stop taking and storing photographs if the scene remains unchanged, but these systems may miss events that happen quickly. For example, cameras are known to use passive infra-red sensors to trigger the camera. A kakapo leaving its nest will trigger the camera because it is warm, but when it returns its feathers will have cooled and it will not trigger the camera. The Blackeye II is in constant operation and will not miss events. However, this makes it even more important that it uses as little power as possible.

Kinopta treats the Blackeye II as a product platform; they customise the camera for different applications. Blackeye II cameras are currently used for the applications described above as well as traffic flow planning surveys. In each of these applications the Blackeye II was selected for its superior night time photography. The camera system is highly flexible and applications under development extend to medical instrumentation.

The two main components to the camera are the image sensor and microprocessor. The image sensor captures the digital images and sends them to the microprocessor. The microprocessor performs image processing operations on the images, encodes them as JPEG files and stores them on-board. The camera hosts a web server so that when it is networked with a computer the user can access it with a web browser. Using an embedded web server allows the camera to provide a custom interface for easy access to the images stored on-board without the need to install custom software on the PC.

Clients are demanding new and improved features on the cameras, but these are proving hard to fulfil with the current system. Faster frame rates are required for traffic monitoring where cars are not in shot for long. Conservation workers need to observe fast events, such as pest species preying on kiwi and other endangered NZ birds, so their cameras also require a faster frame rate. A frame rate of 12.5 fps (frames per second) has been identified as being sufficient for these applications. However, the current

system is only capable of taking photos at 2 fps. The microprocessor in the camera must perform operations on every frame as it arrives and it is unable to keep up with a faster frame rate.

In addition to a faster capture rate, clients demand improved image quality. The nature of these improvements depends on the application. Traffic flow planning operations require the identification of individual cars travelling through a city, so require that the license plates on cars are clearly legible. Monitoring of wildlife reserves requires that a wide field of view is recorded. It should be able to detect creatures moving in the shadows in the background. These improvements in image quality can be achieved with image processing techniques, but as more image processing operations are added to the system, the frame rate decreases even further.

The microprocessor could be upgraded to a more powerful one, but this increases the cost and power consumption of the device and still presents a trade-off between frame rate and image processing capability. An alternative is to add a digital hardware system to the camera to perform the image processing and encoding. Digital hardware systems can be designed in a pipeline to eliminate the trade-off between data rate and the number of operations being performed.

This project proposes the use of a field programmable gate-array (FPGA) device to implement a digital hardware image processing system. FPGAs are highly flexible digital devices that allow for the implementation of almost any kind of digital logic. The proposed system will sit between the image sensor and microprocessor in the data flow. It will take the images from the image sensor, perform the necessary image processing operations on them, then encode them as JPEG data before sending them to the microprocessor for storage. It was decided to continue using JPEG encoding rather than other types of image encoding due to its compression levels and widespread use.

1.1 Project Goal

As mentioned, there is demand for greater features and faster performance of the Blackeye II camera. Its microprocessor is unable to provide these. It can achieve a frame rate of only 2 fps with the current image processing pipeline. The company, Kinopta, would like the Blackeye II camera to be able to capture photos at a rate of 12.5 fps with the addition of more image processing operations. The camera runs off battery for long periods, so the power consumption of the improvements must be kept low.

This project will develop an FPGA-based system to complement the microprocessor in the Blackeye II camera. The aim is to develop a system that will receive data from the image sensor, perform image processing on the data, encode it as a JPEG file and send it to the microprocessor. A proof-of-concept device will be constructed to show the system in operation. It is not the goal of this project to take the system through to a production-ready prototype. Instead the design will consist of an architecture for an image processing pipeline, an example image processing operation to insert in that pipeline, a JPEG encoder and interfaces to the image sensor and microprocessor. Throughout the design process care will be taken to minimise the power consumption of the device.

1.2 Outline of Thesis

This thesis continues in Chapter 2 by covering relevant background knowledge. This includes FPGA use in industry, their advantages compared with other devices, and how FPGAs can be included into systems. An overview of JPEG encoding is provided along with detail of important parts. Chapter 3 sets out the design process of the JPEG encoder. Previous work in the field is used and improved. Particular care is taken to explore the trade-off between precision and accuracy in the design in order to minimise the design resources. Chapter 4 discusses the interfaces between

FPGA and image sensor, and FPGA and microprocessor, and the design of the parts of the system responsible. A power saving mechanism is introduced and implemented, as well as an image sharpening filter. Chapter 5 begins by discussing the final implementation of the system and its performance. It then evaluates the system with regard to the project objectives, and the JPEG encoder is compared with other implementations. Finally, a discussion titled *Future Work* provides a path forward from this project to a final product.

Chapter 2

Background

2.1 FPGAs

An FPGA is a reconfigurable digital logic device. FPGAs have a configurable interconnect network linking from 10^3 to 10^6 logic elements. A logic element (LE) usually consists of some type of configurable look-up-table (LUT), a flip-flop and a multiplexer. By configuring the LEs and the interconnects, an FPGA can behave as almost any digital logic system provided it has sufficient resources.

FPGAs may be used in the same roles as fully customised integrated chips such as ASICs (Application Specific Integrated Chip). ASICs, unlike FPGAs, are not reconfigurable, which does have advantages. There are extra resources devoted to reconfigurability and the layout of the active resources in a reconfigurable system is sub-optimal. This results in longer signal paths and extra logic gates to perform the same task. Therefore there are drawbacks to implementing the same design in an FPGA compared with an ASIC: clock paths aren't as efficient, and therefore the same clock rates aren't achievable; static and dynamic power efficiency is worse; and silicon wafer area efficiency is poorer, therefore FPGAs have a greater per unit cost.

The reduced per unit cost of ASICs means that for large volume pro-

duction they are preferred to FPGAs. However, the non-recurring engineering (NRE) cost of developing an ASIC is significantly higher. While the digital logic design is a part of the NRE of both FPGA and ASIC projects, the ASIC process requires the logic to then be organised into a transistor-based design and the appropriate wafer patterns produced. Not only do these additional stages add to the cost of each design iteration, they add significantly to the time between iterations. Reducing development iteration time and cost, allows for the development to proceed more smoothly; unexpected errors can be rapidly solved before moving on. Reconfigurability also means that systems may be upgraded after production. Due to these advantages FPGAs are used in projects where the volume of production isn't sufficient to warrant the use of ASICs. The volume at which this occurs is increasing over time because FPGAs are being more widely accepted in industry, they become more energy efficient and their per unit cost decreases [2].

FPGAs were developed by Xilinx in the 1980s and are now a major product market currently dominated by Xilinx and Altera, with other companies taking a niche role. As the feature set of FPGAs has increased, the development tools have matured and the per unit costs have reduced, FPGAs are becoming increasingly common in product design in every sector of electronics. In addition to LEs and the interconnect network, modern FPGAs also have specialised hardware blocks, which commonly include RAM, multipliers, Phase Locked Loops (PLLs), and highly configurable I/O blocks for each individual I/O pin [3].

2.1.1 FPGA Development Process

While there are a range of independent development tools for FPGAs, development commonly utilises the toolchain produced or supported by the manufacturer of the target FPGA. This project uses an Altera FPGA from the Cyclone IV range.

Altera provides the Quartus II development software for FPGAs in both free and paid versions. The free version was used here because none of the restrictions of the free version inhibited the project. Quartus II provides several environments to edit functional designs, including the use of hardware description languages (HDLs). This software provides all the toolchains required to turn a design into a *programming file*. It is then able to load the programming file onto an FPGA board connected to the computer. Quartus II provides a wide range of tools to customise and optimise the system beyond just the functional design. Alongside Quartus II, Altera recommends the use of its own customised version of Mentor Graphic's ModelSim software for simulation of HDL designs. Functional design of this project is written in a language called Very-High-Speed Integrated Circuits (VHSIC) HDL, shortened to VHDL. Quartus II is used to synthesise and ModelSim to simulate this design. The following discussion on the development process is specific to these design tools.

A simplified VHDL-based FPGA design flow is depicted in Figure 2.1 as suggested by Pedroni [4]. The process begins with a specification stage, often utilising block diagrams, state machine diagrams and other hardware design techniques. The design is then turned into VHDL code.

VHDL specifies the functional operation of the system, and is primarily based on the concept of signals. Signals in VHDL are strongly typed buses within the system of specified width. While strong typing is employed, the language does allow for significant flexibility; signals can undergo type conversion, so that a signal of type *standard logic vector* (an ordered 1D array of bits) can become the input to a signal of type *unsigned integer*. Signals may undergo a wide range of concurrent and sequential operations from a set of standard libraries (with the option to use user-defined libraries as well). An unsigned integer, for example, has arithmetic operations, such as addition and multiplication, available from a standard library. The code is organised by blocks called *entities*, which may consist of logical operations and other sub-entities known as *components*. An

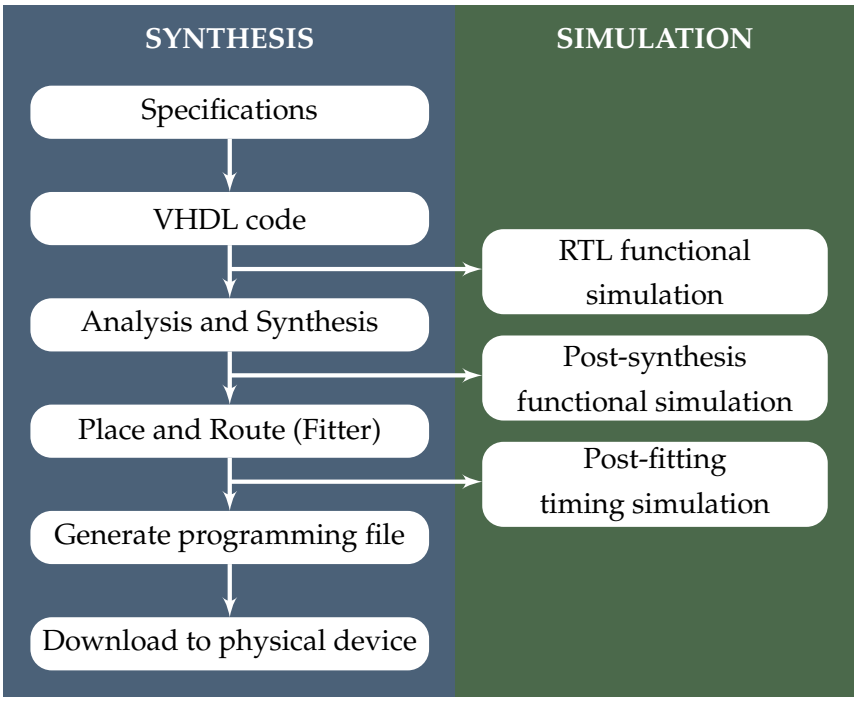


Figure 2.1: FPGA design flow.

entity can be thought of as corresponding to a block in a hardware block diagram and is referred to in this document by the generic term *module*. A module is defined by its I/O interface and its constituent logic.

The next stage of FPGA development is the analysis and synthesis of the VHDL code. This is performed by the development environment, in this case Quartus II, and the development environment will offer different ways to optimise or otherwise influence the analysis and synthesis process. This process converts the HDL description of a system into a netlist of hardware blocks and interconnecting buses. The hardware blocks of these netlists do not represent FPGA hardware components, but are abstract elements that are readily translatable into FPGA hardware components. Note that some signals and operations that are part of the VHDL code will not be explicit in the netlist, but will instead be combined with other signals and operations or removed entirely as part of the optimisa-

tion process. Where signals do explicitly correspond to VHDL signals, the VHDL labels are retained.

Following this stage, is the place and route process, also known as the *fitter*. This, again performed by development software, converts the functional netlist into a description of the utilisation of specific hardware components of the target device. This process is also used to ensure timing limitations of the system. Due to the complexity of FPGA hardware, signals that pass through several hardware resources can experience significant propagation delays. By specifying the timing requirements of a system, the fitter will attempt to ensure these are fulfilled.

In order to use the design on an FPGA, its configuration must somehow be sent to the FPGA. FPGAs use different communication standards to receive configuration data, which is determined by the manufacturer. The programming file specifies how the FPGA should configure its interconnect network, LUTs and other hardware components. Once generated, FPGA programming software is used to transmit the program file to the FPGA via some on-board communication system.

Alongside the synthesis is the simulation process. If a VHDL system passes rigorous simulation tests it is almost guaranteed to pass physical testing [4]. As indicated by Figure 2.1, there are three levels of simulation. All reference to simulation from here on refers to functional simulation. Timing simulation involves calculating propagation delays and displaying their effects. However, timing simulations were not necessary in this project because at no point was the logic sufficiently complex to cause propagation delays comparable with the clock speeds.

A VHDL simulation requires a special type of VHDL module called a *testbench*. A testbench instantiates the module to be tested and specifies the generation of the module's input signals. Signals in a testbench may be read from or written to files. Simulations can be set up to launch from Quartus II, which will run ModelSim and provide it with an automatically generated script to run. While testbenches may output infor-

mation to a command line, a simulation is primarily viewed via a timing diagram interface. In functional simulations the timing diagram assumes zero propagation delays so concurrent operations will occur concurrently, and sequential operations will occur on clock edges. The user may view any signal of the testbench, the main module or any of its constituent modules (so long as that signal still exists post-synthesis). Many signals may be selected simultaneously, and the waveforms of these signals are generated for a user specified simulation period. The timing diagram allows the developer to view the entire operation of a system in great detail in order to track down system bugs.

2.1.2 CPU-FPGA coupling

Advanced embedded systems are traditionally software driven; they are controlled by a CPU. While FPGAs provide improvements in many processing applications, many tasks are much better suited to software. As FPGAs have become increasingly common in these systems the various configurations by which they are coupled to the system's CPU have been categorised [5, 6]:

1. The most closely coupled case is that of a CPU with reconfigurable hardware functions integrated into it. The purpose of such a close coupling is to allow the reconfigurable hardware to add to the core architecture of the CPU. Custom instructions can be implemented to accelerate particular algorithms, passing data via shared registers. Closely linked to this is the increasingly common phenomenon of soft CPU cores, that is, CPUs that are implemented on an FPGA. These CPUs are becoming increasingly mature, particularly those provided by the major FPGA manufacturers such as Altera's NIOS II and Xilinx's MicroBlaze. These allow for a highly customisable CPU system with optional instructions easily added or removed.

2. The next most closely coupled case is a coprocessor system. The reconfigurable hardware has more independence than the functional unit described in 1, while still being highly integrated into the CPU. The coprocessor may receive a trigger from the CPU and operate independently for several clock cycles until a result is achieved. A clear advantage to this system is the core CPU and reconfigurable hardware may operate in parallel, which provides performance gains at the cost of added care required in software design. The coprocessor has access to the CPU's memory cache, as opposed to direct access to CPU registers as in 1.

3. An attached processing unit is one stage further from the CPU. It is further removed in terms of memory access too: it does have access to the CPU's memory, which it shares control of via a direct memory access (DMA) system. Communication between the CPU and the attached processing unit is therefore slower than the above cases, although direct memory access does allow for such a processor to undertake large tasks on data independently from the CPU and return the result in place for the CPU to easily access it.

4. A standalone system is one where the CPU and reconfigurable hardware are not integrated. The reconfigurable hardware will operate largely independently of the CPU and only infrequently communicate with it via system I/Os or a network link. This is the case where an FPGA and CPU are both part of a single device but do not share memory.

In general, reconfigurable hardware more closely integrated into the CPU has greater and faster access to it, but is more dependent upon it. Furthermore, the more highly integrated system will generally have fewer reconfigurable hardware resources than independent systems. Garcia et al. [7] propose a similar categorisation but with three categories: a functional unit (as in point 1 above); a coprocessor, which has access to the

CPU's memory and possibly its cache (as in points 2 and 3 above); and a heterogeneous multiprocessor, which may have a shared memory system or may communicate via a network or communication architecture (as in 3 and 4 above). The latter categorisation more closely reflects the necessary physical integration levels between reconfigurable hardware and a CPU: amalgamated into the CPU architecture, as a peripheral of the CPU, or on an independent chip. A general term for the reconfigurable hardware in these systems is a hardware accelerator, though this term also applies to other hardware systems.

2.1.3 FPGA-based Hardware Accelerators

FPGAs provide a hardware solution to many engineering problems. Software solutions running on a microcontroller or microprocessor can provide equivalent results. Each design task has its own problems that need to be solved, and an appropriate selection of technology is required to provide an appropriate solution. Table 2.1 provides a list of general advantages of software over hardware and vice versa.

Table 2.1: List of advantages of software and hardware in the general case.

Software	Hardware
Rapid updates	Highly optimised system
Lower development time	Faster system performance
Expertise widely available	More power economical
Lower development cost	

The choice between using software or hardware goes beyond that of resource cost versus degree of system optimisation. The degree to which systems can be improved through the use of hardware, or the reduction in development time for software, depends entirely upon the task being undertaken. While research is being done on the potential for systems

to be accelerated by FPGAs, the advantages of software have previously inhibited widespread uptake of FPGAs in industry [8].

CPUs excel at performing linear tasks where a lot of decision making occurs, and software systems are highly adaptable. Hardware systems are inherently less flexible and therefore are largely confined to tasks that are well defined. On the other hand hardware can perform operations on an almost limitless level of parallelism, whereas CPUs only perform one operation at a time, so are not suited to tasks that are well pipelined.

Digital system design faces continuous performance pressures. These pressures may be relieved by the appropriate use of hardware. This relief is needed, but at the same time so is the flexibility and rapid development cycles of software. To this end, there is an increasing use of digital hardware solutions coupled to software systems. Hardware accelerators are specialised hardware systems that are used to augment a software system, providing the benefits of a hardware solution to those parts of the system that stand to gain the most from it.

Specifically, the gains may be either in data throughput or in power reduction, or both. As mentioned at the start of Section 2.1, FPGAs do not provide the same gains in power reduction, nor are they capable of running at the same clock speeds as ASICs, but despite this they are capable of outperforming a microprocessor for some tasks. However FPGAs have made great progress in these areas in recent years [9].

The performance of hardware accelerators is measured by two metrics: run time and power consumption. Run time specifies how long it takes to perform a given operation, usually its main function, which may be compared with the equivalent run time in software. The power metric is defined as the average power consumption during the run time. The product of the power consumption and the run time is the operation energy. Again a direct comparison with software may be made. The overall power savings are more complex because the static energy (that is the energy consumed when the operation is not being performed) of the ac-

celerator must be taken into account as well as the proportion of time the accelerator is operating. Furthermore, no net power savings of a system will be achieved unless the energy consumption of the software system is decreased. Simply put, either the processor is clocked slower, enters a low power mode for longer periods of time, or is replaced by a lower power processor as a result of the addition of the hardware accelerator. Alternatively, rather than reducing power, the addition of the accelerator may be incorporated in order to increase the system throughput, in which case the total power consumption of the system may remain static, or in fact increase.

Traditionally FPGAs have been coupled to CPUs to provide what is known as *glue logic* [5], that is, an interface between one digital system, such as a CPU, and another digital system. The role of hardware accelerators has taken them into new fields where they have proved successful. These fields include networking, encryption, software-defined radio, medical imaging, scientific data acquisition and analysis, spacecraft, robotics, automotive and image and video [7].

2.1.4 FPGA Design for Embedded Systems

Portable embedded systems are becoming increasingly ubiquitous. The demands on embedded systems increase in the form of greater performance and flexibility, lower power consumption, and cheaper costs [10]. The incorporation of an FPGA may reduce the requirements of the microprocessor, so that a lower power microprocessor solution might be used. While the addition of an FPGA will make a significant contribution to the static power consumption of a device, they enable vast reductions in the dynamic power consumption of algorithms. An algorithm may take many more clock cycles to perform on a CPU than in an FPGA, if it is sufficiently pipelinable.

To show the effects of parallelism, an example is provided based on

the pseudo-code example in Figure 2.2. This simple algorithm takes input array, E, performs operations OP_1 and OP_2 on the data then outputs the result to array E″.

for $i = 0 \rightarrow 9$ **do**
    $E'[i] = OP_1(E[i])$
end for
$E''[0] = OP_2(E'[0], 0)$
for $i = 1 \rightarrow 9$ **do**
    $E''[i] = OP_2(E'[i], E'[i-1])$
end for

Figure 2.2: Example algorithm

Assuming both operations can be computed in a single step then a clock by clock comparison of how this algorithm would progress on a CPU and an FPGA is given in Table 2.2. Parallelism means that while the FPGA may take as many clock cycles as the CPU for each data element, the FPGA will output one result each clock cycle, whereas the CPU will output one result every four clock cycles.

Table 2.2: Comparison of the performance of an algorithm as implemented on CPU vs. FPGA.

clk	CPU	FPGA			
0	fetch(E_0)	fetch(E_0)			
1	$E'_0 = OP_1(E_0)$	fetch(E_1)	$OP_1(E_0)$		
2	$E''_0 = OP_2(E'_0, 0)$	fetch(E_2)	$OP_1(E_1)$	$OP_2(E'_0, 0)$	
3	store(E''_0)	fetch(E_3)	$OP_1(E_2)$	$OP_2(E'_1, E'_0)$	store(E''_0)
4	fetch(E_1)	fetch(E_4)	$OP_1(E_3)$	$OP_2(E'_2, E'_1)$	store(E''_1)
4	$E'_1 = OP_1(E_1)$	fetch(E_5)	$OP_1(E_4)$	$OP_2(E'_3, E'_2)$	store(E''_2)
⋮	⋮	⋮	⋮	⋮	⋮

The mechanism by which an FPGA may reduce power consumption springs from parallelism. As shown in the example above, an algorithm implemented in a parallel manner may be performed in far fewer clock cycles per data element. Therefore, to achieve the same data rate as the CPU, the FPGA must be clocked at a slower rate. As dynamic power consumption is directly proportional to clock rate, the power used by the FPGA for the algorithm is likely to be less than that off the CPU despite its inherently poorer power efficiency. Conversely, if the FPGA were to be clocked at a faster rate the overall power consumption may stay the same or even rise, but the data rate will be increased significantly. The benefits of using hardware accelerators can be significant, with examples showing a 90% power reduction for the same data rate, or a 200-fold data rate increase for the same power [11].

2.1.5 FPGA Power Considerations

The power usage of an FPGA differs across three time periods: power-up, configuration and execution. Note that Flash-based FPGAs, as opposed to the predominant SRAM-based chips, will be slightly different. Power-up refers to the spike in current draw during the initial few microseconds of power being provided. This is dependent upon the chip itself rather than its configuration, and is largely attributed to on-chip SRAM. Configuration occurs shortly after power-up, and at any point during operation in which the system is reconfigured. The execution component accounts for the rest of the FPGAs operation, and consists of both dynamic and static power [12].

Static power consumption is constant for a given device. The static power consumption of FPGA devices is a major point of competition between vendors, so that devices with equivalent resources are decreasing in static power from one generation to the next, and vary considerably between vendors [2]. Within a family of devices the tendency is for static

power to be proportional to the resources on the device. Dynamic power is dependent upon the rate at which signals change value due to the nature of CMOS architectures. Significant dynamic power consumption occurs both in the LEs and the interconnect [12]. Different configurations of the same functional design has different power efficiencies (for example, minimising the number of gates that might change on every clock cycle reduces dynamic power consumption).

To attain the best possible results from a hardware accelerator, the power consumption of the FPGA must be carefully considered. FPGA development software have variants of their place and route algorithms that favour low power design over other design parameters. The system on the FPGA, or any constituent system thereof, may not be utilised 100% of the time, therefore clock-enable systems may significantly reduce the proportion of the time that large parts of the chip are clocked without any reduction in performance. Specialised clock trees in the recent FPGAs' interconnect have allowed hardware clock gates to be implemented which are a vast improvement over clock gating via LEs, and furthermore are easily implemented with the vendor-supplied development software [13]. Idling systems for microprocessors are less flexible by comparison; the CPU, being the point of processing, must be fully active whenever any processing is required.

A complementary approach to reducing the FPGAs power consumption is to minimise the resource usage of the design, which has two effects: it decreases the overall dynamic power usage by minimising the size of dynamic systems; and it enables the use of FPGAs with fewer resources, and therefore lower static power consumption.

Some vendors produce FPGAs for low-power systems. These low-power FPGAs generally have fewer resources available than their standard counterparts. This provides further incentive to reduce the power resource usage of systems designed for FPGA. The proof of concept system built for this project uses an Altera Cyclone IV FPGA, which is not

considered a low-power device. This FPGA was chosen for convenience in the development process. However, the design of system is kept as vendor neutral as possible so that it may be ported to a different FPGA range in the future once the final resource usage and power consumption has been considered.

2.1.6 FPGA Summary

FPGAs present a compelling alternative to ASICs in digital hardware design, particularly for low volume production. The advantage of an FPGA system are the comparatively low development costs and rapid design iterations.

While they compete against CPUs in some applications, in a hardware accelerator configuration, FPGAs are used to complement the main processor of the digital system. The coupling between the processor and configurable logic may be very close, where the hardware extends the capabilities of the processor, or may be distant, where an FPGA operates on data independently only communicating where necessary with the CPU.

The Blackeye II camera system currently does all of its image processing and encoding on a microprocessor. This project proposes the use of an FPGA in between the camera sensor and microprocessor to offload these processes from the microprocessor (see Chapter 4). By careful design the additional hardware may improve the overall performance of the system.

This project uses a Cyclone IV FPGA from Altera, one of the two main FPGA manufacturers. Development is done in the hardware description language VHDL using Altera's Quartus II development software to synthesise designs and Mentor Graphic's ModelSim to simulate them. During the design process, care was taken to reduce the system's resource usage and power consumption.

2.2 JPEG Algorithm

The Joint Photographic Experts Group (JPEG) was formed in 1986 to establish a standard for the encoding of greyscale and colour images [14]. The resulting document published in 1992, officially titled *Information Technology–Digital Compression And Coding of Continuous-Tone Still Images–Requirements And Guidelines*, is commonly referred to by the name of the committee: JPEG. The standard itself is not a single algorithm, but provides a set of rules for the compression of images; however, the appendices elaborate on the standard by providing examples, which are commonly used for JPEG implementation. The standard does not specify a file format, but provides a structure to add file information. EXIF and JFIF are the two common file formats for JPEG images, and both are read by most software [15].

JPEG has become the most used format in digital image storage [16]. In the case of digital photography JPEG has become the de facto standard. The advantage of JPEG use in photography is that, of the common image formats, it provides the greatest level of compression [17] by exploiting the following. First, the JPEG system builds upon previous work that showed that human perception focuses on low spatial frequencies [18]. Secondly, photographs exhibit guassian distribution of DC data and lapplacian distribution of AC data [19]. This means that less precision is required for different spatial frequencies, which the JPEG standard exploits. And thirdly, the difference between the DC representations of neighbouring parts of an image is more likely to be smaller than the absolute value of those DC representations. While more sophisticated human visual system models now exist based upon psychological and neurological research these advanced models would require more complex compression systems [20].

Due to its widespread use, and high levels of compression, JPEG is used by the Blackeye system.

Viewed algorithmically, JPEG compression is divided into 7 or 8 dis-

tinct steps [21]:

1. Convert from RGB to a luminescence-based colour system such as YCbCr. In greyscale images this step is not required.
2. The chrominance components of the image are reduced in spatial resolution, usually either 2:1 reduction in horizontal direction only, or 2:1 in both horizontal and vertical directions. In greyscale images this step is not required.
3. The pixels of each colour are grouped in blocks of 8×8 pixels for further compression. If the resolution isn't a multiple of 8×8 then extra pixels are inserted, copying the pixels of either the right or bottom edge. These blocks are called *minimum coding units* (MCUs).
4. A two dimensional discrete cosine transform (DCT) is performed on each MCU to produce an 8×8 frequency map.
5. The data is quantised. This step is lossy; data is irretrievably lost. Each DCT coefficient is divided by a specific quantisation value then rounded to the nearest integer. These quantisation values are specified in a *quantisation table*. The JPEG standard provides recommended quantisation tables.
6. A variant of Huffman encoding is performed on each data unit. This may be replaced with arithmetic encoding.
7. Headers and requisite tables are added to the resulting file. The file format has multiple standards, the two most common of which are the JFIF and EXIF formats.

2.2.1 JPEG Modes

The JPEG specification is very flexible, allowing the encoders to choose from a range of options regarding the encoding process. The main options (indicated by bold type) are described as follows.

There are four distinct modes of operation available to JPEG encoders: **baseline sequential**, **baseline progressive**, **lossless** and **hierarchical**. Lossless is self-explanatory. Hierarchical is used for storing multiple frames in a single file, where each frame's encoding depends on the previous frames (i.e. the kinds of processes used in video encoding). Sequential baseline mode, the most common [22], encodes the data entirely in independent 8×8 data blocks, whereas baseline progressive performs DCT and quantisation operations on 8×8 data blocks, but then performs entropy encoding upon groups of these data blocks.

Multiple **image components** are allowed per frame. These image components can be used to represent the colour space of the image; for example an RGB colour space would require three image components per frame (one for red, blue and green data samples). The JPEG specification does not provide for different colour spaces, relegating this task to the file format. However, it does provide example Huffman and quantisation tables for luminance and chrominance. The JFIF file format, for example, specifies the use of YCbCr colour space. The components need not be of the same sample rate, for example, chrominance pixel data may be binned while luminance pixel data stays at full resolution. This means that pixels in Bayer pattern form may be directly encoded. Only one image component is used for greyscale images. These image components may be interleaved (the first block of all image components is followed by the second block of all components) or not (all the blocks of the first image component are followed by all the blocks of the second component).

The specification allows for **8-bit** or **12-bit** sample resolutions. These resolutions apply to the pixel data components such as luminance and chromitacity.

The entropy coding stage (where the data compression occurs) may use one of two different techniques: **Huffman coding** or **arithmetic coding**.

While the JPEG standard allows for many different options, the application that this work is concerned with only requires the simplest JPEG

implementation, namely 8-bit greyscale baseline sequential. Other aspects of the JPEG specification are ignored in the rest of this document.

2.2.2 Discrete Cosine Transform

The discrete cosine transform (DCT), sometimes referred to as the forward DCT to distinguish it from the inverse, is a discrete transform of data points to a sum of cosine functions. The DCT was first proposed in 1974 as a derivative of the discrete Fourier transform, and its potential application in image compression was immediately noted [23]. The transform has four main variants, but the JPEG standard specifies a variant of DCT-II [14]. The general DCT can be applied on a one-dimensional data array of arbitrary length, although multidimensional DCTs are derived from performing DCTs along each dimension sequentially. In the case of JPEG, the algorithm is further simplified because the data is always an 8×8 block of data of integers in the domain [-128, 127]. Equation 2.1 shows the form used in the JPEG standard. The resulting 8×8 matrix is a representation of the original MCU in the frequency domain. Note that element $g_{0,0}$ is the DC element: it is $\frac{1}{8}$ of the mean of the original MCU's pixels. The remaining elements are AC in nature and increase in frequency with their indices.

$$g_{u,v} = \alpha(u)\alpha(v) \sum_{x=0}^{7} \sum_{y=0}^{7} f_{x,y} \cos\left[\frac{\pi}{8}\left(x + \frac{1}{2}\right)u\right] \cos\left[\frac{\pi}{8}\left(y + \frac{1}{2}\right)v\right] \quad (2.1)$$

$$\alpha(u) = \begin{cases} \frac{1}{\sqrt{8}} & \text{if } u = 0 \\ \frac{1}{2} & \text{if } u \neq 0 \end{cases}$$

2.2.3 Quantisation

The quantisation stage of the JPEG algorithm is the lossy stage. DCT coefficients (the elements of an MCU after the forward DCT transform) are

each divided by a constant, and the loss of information occurs due to maintaining integer level precision of the result.

The quantised matrix is the Hadamard product of the DCT matrix and some quantisation matrix. No quantisation matrix is specified by the JPEG standard, and therefore one must be included in the JPEG file for the decoder to use. It is usually represented as the array of elements by which the DCT matrix elements are divided, known as the quantisation table. While there is no standardised matrix construction system, Annex K of the JPEG Standard gives example chrominance and luminance quantisation tables as shown in Table 2.3. In 1991 the Independent JPEG Group (IJG) released one of the earliest open source JPEG software libraries, from which stems the quality system that is widely, though far from universally, used. IJG refers to the tables from Annex K as Q_{50}, indicating a *50% quality factor*.

Table 2.3: The sample luminance quantisation matrix provided in Annex K of the JPEG Standard.

$$Q_{50} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Other quantisation tables are derived from Q_{50} by Equation 2.2 [24].

$$Q_x = \begin{cases} \frac{100-x}{50} \cdot Q_{50} & x \in (50, 100) \\ \frac{50}{x} \cdot Q_{50} & x \in (0, 50) \end{cases} \tag{2.2}$$

As discussed at the start of Section 2.2, lower frequencies are of greater significance than higher frequencies. This is reflected in this quantisation

table, which exhibits lower coefficients in the top left (low frequency) than in the bottom right (high frequency). Higher coefficients lead to smaller data values post quantisation which in turn are able to be further compressed than large values [25].

2.2.4 Entropy Encoding

This is the stage of the algorithm in which compression occurs. The two systems that can be used are Huffman encoding or Arithmetic encoding. Arithmetic encoding usually provides greater compression at the cost of further data processing. Huffman encoding, the one used in this project, is explained in further detail here.

The order in which each element of an MCU is encoded is based upon a zigzag ordering, starting with the top left element and ending in the bottom right element. The diagram in Figure 2.3 depicts this.

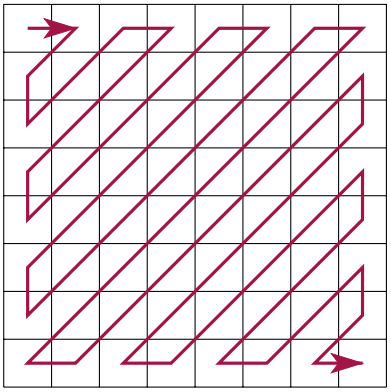


Figure 2.3: Zig-zag ordering starting in the top left (0,0) and ending in the bottom right (7,7)

As mentioned above, the lower right is more likely to have runs of zeros and this ordering therefore clumps them together. Runs of zeros are the most compressible form of data in this type of Huffman encoding.

The DC elements (i.e. the first element of each MCU) is considered differently from the other elements. The DC value is replaced with the

difference between the current and the previous DC values. In general, photographs exhibit the frequent tendency of neighbouring MCUs having similar average intensities. Therefore the difference between DC values will often be small. Small values result in greater compression. Additionally, different Huffman tables are used for DC elements than those for AC.

Each encoded element consists of a code followed by a value (some values are of length 0). The code is from a Huffman table and specifies the length of value and the size of the run of zeros preceding the value. Huffman values are a form of binary representation where the length of the representation is important. This conversion is shown in the Table 2.4.

Table 2.4: Huffman value encoding table. This table compares the quantised element with its corresponding Huffman value.

Value Length	Quantised Values		Huffman Values	
0	0	0		
1	-1	1	0	1
2	-3, -2	2, 3	00, 01	10, 11
3	-7, -6, -5, -4	4, 5, 6, 7	000, 001, 010, 011	100, 101, 110, 111
4	-15,...,-8	8,...,15	0000,...,0111	1000,...,1111
5	-31,...,-16	16,...,31	00000,...,01111	10000,...,11111
⋮	⋮	⋮	⋮	⋮
11	-2047,...	...,2047	00000000000,...	...,11111111111

DC elements are always at the start of an MCU, thus they do not have runs of zeros preceding them, so there is one Huffman code for each value length. As in the case of quantisation tables, these values are not specified by the JPEG specification, because each Huffman table will provide differing levels of compression for any given image. However, there are Huffman tables provided in Annex K of the specification that provide generally good levels of compression and are widely used. Table 2.5 shows the

Huffman table for DC elements (the equivalent AC table is in Appendix A).

Table 2.5: Huffman Table: luminance DC coefficient differences (JPEG Standard Annex K)

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

The table for AC elements is much longer as both value length and zero count are encoded. Quantised AC elements that are 0 do not have a corresponding Huffman output, but are only counted towards the current zero run. If 16 consecutive zeros occur there is a special code used and the zero run count restarts. If there are only zeros remaining in the current MCU, there is a special code (end of block) for terminating the MCU prematurely.

As an example to clarify this encoding: suppose the first MCU has DC element -16. This corresponds to a Huffman value of 01111 (Table 2.4). As it is value length 5 (the length in bits of the Huffman value) the Huffman table specifies the corresponding code word to be 110 (Table 2.5). Therefore, the entire Huffman code would be 110 01111.

The Huffman codes are concatenated into a single stream of data. The stream must be of an integer number of bytes in length, so if the number

of bits in the entire stream is not a multiple of 8 then the difference in bits is padded with 1's. Furthermore, as the byte 0xFF is used to indicate a marker in a JPEG file, any 0xFF that occurs in the data must have a 0x00 inserted after it.

2.2.5 File Format

The discussion in this section thus far has only considered the compression of images, not how the data is stored. JPEG image data is stored inside a file with extension .jpg or .jpeg, in a flexible format specified in the JPEG document. In addition to image data a variety of parameters are stored in the file specifying the specific implementation of JPEG being used, including quantisation and Huffman tables. There are two main JPEG file formats, EXIF and JFIF, both of which specify additional metadata that may be included [15].

The diagram in Figure 2.4 shows the form that a JFIF file may take, and in this case it is of the simplest form, as used in this work.

Note that each section preceding the data starts with a JPEG-specified marker (always 0xFFXX), followed by 2 bytes that specifies the length of that section; when decoding the file only the data is of unknown length. The end of the data is signified by the end of image marker. As noted in Section 2.2.4, bytes in the data which are 0xFF have a 0x00 byte stuffed after them to prevent confusion with markers. If none of the parameters vary, then the entire file, less the data, may be hardcoded into the system producing the file.

2.2.6 JPEG Summary

The JPEG specification [14] provides a standard to encode images. While it does have a lossless option, it is for the standard's lossy compression that it has been widely adopted for digital photographs. The Blackeye II uses a software library to encode images from its camera chip into JPEG-based

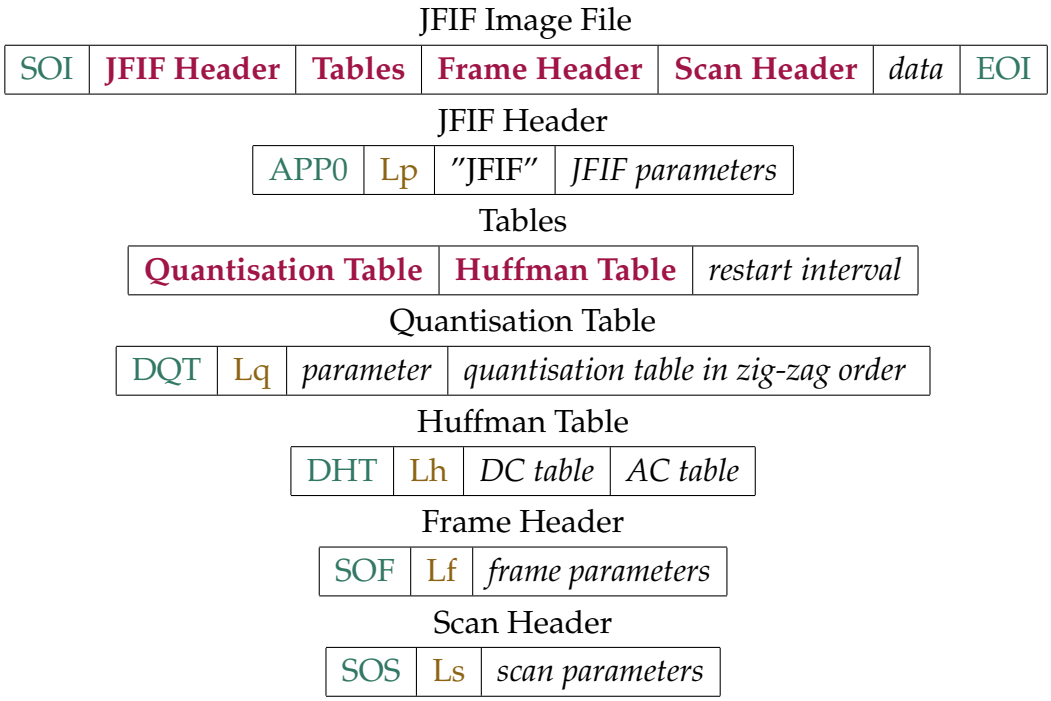


Figure 2.4: Example JPEG file containing in JFIF format. Standard JPEG markers XXX are 2 bytes. Length indicators Lx, also 2 bytes, indicate the length of each header or table in bytes.

files. This project implements a JPEG encoder on an FPGA to reduce the load on the microprocessor.

The unencoded data is in the form of a 752×480 pixel image. Each pixel is represented by an 8-bit greyscale value, which is piped into the JPEG encoder along an 8-bit bus. This stream arrives row by row, left to right, rows starting at the top and going down. The encoder performs a baseline sequential greyscale encoding of the data which is stored in a JFIF file.

The encoder performs a 2D-DCT operation on 8×8 blocks of pixels called MCUs. The DCT-MCU, also an 8×8 block of data, represents the original MCU in a discrete frequency domain. A quantisation operation reduces the precision of the DCT-MCU, and in the process makes the en-

coding process lossy. The quantised-MCU is reordered and compressed by a Huffman encoder. The Huffman encoded data forms the JPEG data, which is encased in a JFIF file that adds metadata, such as, specifications of which JPEG mode was used, along with Huffman and quantisation tables.

2.3 Image Processing on FPGAs

Image processing has been an ongoing field of computing since 1957 [26]. Image processing techniques have made significant advances since then as computer hardware and software techniques have improved, and as early as 1991 researchers began developing imaging systems using FPGAs [27]. Hardware excels in a real-time system because the sheer number of operations that must be performed within a time frame preclude traditional software solutions. As images consist of many data points, the fetch-decode-execute paradigm of CPUs results in excessive overhead compared with the parallelism of FPGAs [28]. This parallelism exists in images both spatially and temporally; different parts of frames can be processed simultaneously, and subsequent frames can commence while the current frame is still being processed [29]. Complex algorithms can be pipelined in an FPGA to respond to the requirements of real-time imaging systems, which include preprocessing, classification and encoding.

2.3.1 Image Processing Pipeline

The pipeline is made up of stages corresponding to operations. Some operations require longer pipelines than others. The length of the pipeline is the number of clocked registers the data must pass through from the start to the finish of the pipeline. This length is equal to the latency of the system; more specifically, the latency in clock cycles between data entering the pipeline and the corresponding data leaving the pipeline. A longer pipeline will result in a greater latency that may put pressure on any real-

time constraints the system has.

A longer pipeline will also consume more hardware resources. Even stages of the pipeline that involve no operation, but only buffer, need resources: either the data are stored in LEs, in the FPGA's on chip block RAM, or on off-chip RAM. Off-chip RAM is limited by the bandwidth of its interface. This makes off-chip RAM inappropriate for use by several parts of the system (different buffers or operations). The different buffers must write and read from the RAM simultaneously. This may be possible with the appropriate type of RAM and a well-designed memory access system but this adds significantly to the complexity of the design. Thus off-chip RAM is of most use in the storage of a large amount of data at a single section of the pipeline.

On-chip block RAM is better suited for use by many operation buffers. As the name suggests block RAM is made up of many blocks of RAM, each highly configurable and able to be accessed independently. This means that it is usable by several parts of the systems at once, each part having a block, or multiple blocks configured to operate as required. The limitation is that any single block RAM can only be used by one RAM interface, meaning that any partially used block is wasteful. LEs are highly configurable, but the extent of their configurability means that much of their function is wasted if used purely for buffering. Thus both LEs and block RAM make suitable buffers for a pipeline but both are limited resources on an FPGA. Therefore adding many large buffers to a design will rapidly consume the FPGAs resources.

2.3.2 Image Processing Operations

Image processing operations can be broadly divided into three categories [28]: point operations, window operations and global operations. These refer to the portion of the image that each operation affects. Point operations are a mapping of one input pixel to one output pixel, whereas

window operations take a sliding rectangle, or window, of pixels as input to produce an output pixel. Global operations are those that depend on data from the large parts, or the entirety of the frame. While all three of these types of operations may be performed on both CPUs and FPGAs, point and window operations are more suitable to FPGAs due to their short pipeline. Global operations with a much longer pipeline are more suited to software that is better suited to repetitive memory access systems.

Point operations are the simplest image operation, they perform some function on each pixel individually. A simple example of this would be thresholding: values below the threshold are output as white, values above the threshold are black. To pipeline image operations the image should be processed serially, that is, going along each pixel on a row before starting to the next row, or less commonly going down column by column. This is simply a case of reading image data as it arrives, or sequentially from a buffer. Adding a point operation will usually involve a buffer of one pixel followed by the operation logic, as shown in Figure 2.5.

Thus the output of the point operation is a serial stream of pixel data with a 1 clock cycle delay, or *latency* of 1 relative to the input.

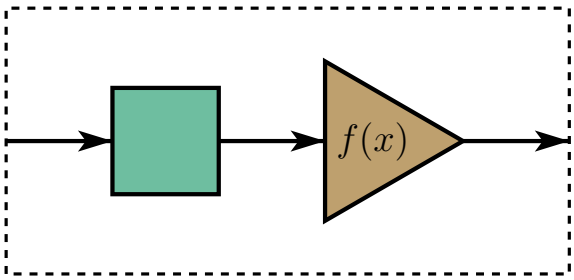


Figure 2.5: Point operation. Single pixel buffer followed by an operation results in a 1 clock cycle latency.

Window operators are more complex. Assuming that pixels are being received serially row by row, sufficient pixels must be buffered in order to obtain a window of the desired size. In reference to Figure 2.6, a window

of size $M \times N$ requires a buffer of $(N-1)+M$ pixels, as the window slides along and down the frame the buffer fills with the input pixels. The size of the buffer determines the latency as shown in Equation 2.3.

$$\text{latency} = L_{row}(N-1)/2 + (M+1)/2 \tag{2.3}$$

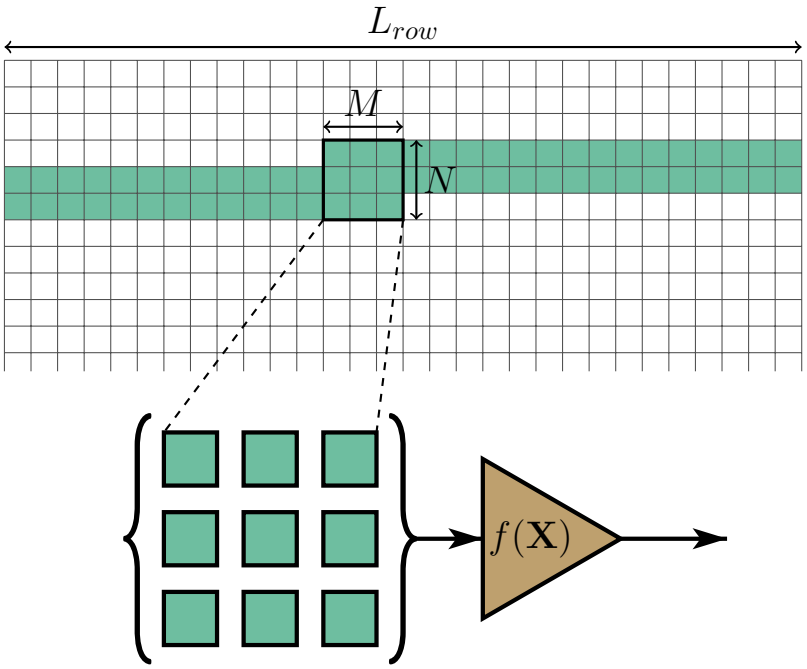


Figure 2.6: Window operation. In this example the window is 3x3, and the grid above shows the pixels of the frame that are buffered when the window reaches the point indicated.

Common window operators are linear functions; the output pixel is a weighted sum of the input pixels. An example of a linear window operation is a Gaussian blur [30].

2.3.3 Summary of Image Processing

Image processing was identified as an application of interest early in the history of FPGAs. Many image processing operations are appropriate for acceleration in hardware due to their pipelinability. Operations that do not require an entire image, but only a part, called a window, are most suitable for efficient hardware implementation.

The Blackeye II system currently performs many image processing operations on its microprocessor. This project provides an architecture by which image processing operations may be implemented in hardware. To provide an example of this a simple linear window operation, an image sharpening filter, is included in the FPGA system.

2.4 Pre-existing FPGA-based JPEG Encoders

Commercial hardware designs for implementation in FPGAs (or other hardware systems such as ASICs) are available as intellectual property (IP) blocks. There are several companies offering JPEG encoder IP blocks. Information was able to be obtained about ones from Entner Electronics, VISENGI, CAST, Sundance Microprocessor Technology and Barco Silex [31, 32, 33, 34, 35]. IP blocks are offered under many licensing schemes: some use a royalty system, others use a one-off payment. Most provide the IP block as an encrypted file for addition to an FPGA project so that the source code is unavailable. Entner Electronics will provide you with the VHDL code of their system for extra cost.

Kinopta are wanting to trial a hardware-based image processing and encoding system in their cameras without having to fully commit themselves upfront to, what is for them, a new technology. For a small company, such as Kinopta, who aren't already committed to an FPGA based system the costs of these IP blocks can be prohibitively expensive.

The power requirements of Kinopta's camera systems is a major con-

cern of theirs. By not having access to the source code of any system they utilise, they are severely limited in their capability to optimise the system to their needs.

There are some open source JPEG encoder designs, these are free and provide the source code. However, most were too limited in their capabilities for this application. Furthermore, the licensing of most of these designs was too restrictive for Kinopta.

Despite these problems, these commercial designs are important reference points by which to compare JPEG encoder designs. Most of the examples mentioned provide their FPGA resource usage and advanced features. In the final chapter of this thesis these will be used in the evaluation of the JPEG encoder design.

2.5 Summary

An FPGA is a flexible digital system that can be configured to perform almost any digital logic. They can be used to great effect for operations that are inherently parallel. Paired with a CPU an FPGA may take the role of hardware accelerator: it provides addition processing capability for the CPU. This project uses an FPGA to perform image processing and encoding for a CPU to increase the system's performance.

Many image processing operations are parallel; each part of the image is operated on separately. Therefore, FPGAs have been identified as being good devices to perform image processing on. A pipeline can be formed on the FPGA so that while the first part of the image is undergoing the second operation, the next part of the image is undergoing the first operation.

The JPEG specification provides a standard to encode images. It performs compression to reduce the size of the image data and is widely used for digital photographs. The encoding process operates on 8×8 pixels blocks called MCUs, which are first transformed into the frequency do-

main by a two dimensional discrete cosine transform (2D-DCT). The data is then reduced in precision and encoded by a Huffman encoder. The Blackeye II uses a software library to encode images from its camera chip into JPEG-based files. In this project an FPGA-based JPEG encoder is developed.

Chapter 3

JPEG Encoder Implementation

Since the JPEG standard was first announced many encoder and decoders have been developed in hardware. In recent years interest in JPEG encoders in FPGAs has increased for the reasons described in Section 2.1. JPEG encoders are partitioned into four stages to modularise the design process: 2D-DCT, Quantisation, Zigzag Buffer and Entropy Encoder [36, 37]. This chapter describes the design process of the JPEG encoder, and is partitioned into one section for each module. The subsequent chapter describes the design of the system in which the encoder is incorporated, adds further context to the encoder's design, and hence introduces further control mechanisms.

The JPEG algorithm, as described in Section 2.2, divides the frame up into 8×8 blocks of pixels called MCUs. As each MCU is encoded sequentially the JPEG algorithm can form a pipeline: the first MCU begins encoding, and while it is still being processed the second MCU commences encoding. At this point it is assumed there is a constant stream of data. This stream is 8-bit parallel data, each representing a greyscale pixel value in the range [0,255], traversing an image row by row, from left to right and top to bottom. Before the data enters the encoder it must be converted from 8-bit unsigned data to 8-bit signed data appropriate for the DCT, within the range of [-128,127]. As per Figure 3.1, this conversion is com-

pleted in hardware by a single NOT gate. The output of this subtraction operation flows directly into the input of the DCT module.

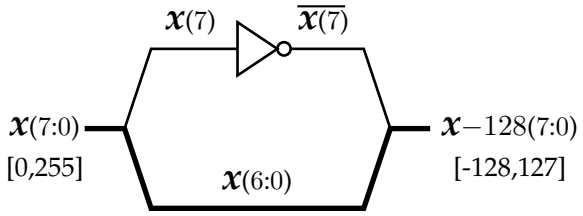


Figure 3.1: Subtracting 128 from an unsigned 8-bit integer and converting it to an 8-bit two's compliment signed integer.

3.1 Discrete Cosine Transform

As described in Section 2.2.2, the JPEG variant of the DCT takes an input of an 8×8 block of pixels, $\mathbf{F}$, and outputs a frequency-based representation of the block as an 8×8 matrix of values, $\mathbf{G}$, as in Equation 3.1. A 2D-DCT is a process intensive operation; assuming the cosine terms are held in a look-up-table, each of the 64 elements requires the summation of 64 terms, each the product of 4 terms.

$$\mathbf{G}_{u,v} = \sum_{x=0}^{7}\sum_{y=0}^{7} \alpha(u)\alpha(v)\mathbf{F}_{x,y} \cos\left[\frac{\pi}{8}\left(x+\frac{1}{2}\right)u\right] \cos\left[\frac{\pi}{8}\left(y+\frac{1}{2}\right)v\right] \quad (3.1)$$

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{8}} & \text{if } u = 0 \\ \sqrt{\frac{2}{8}} & \text{if } u \neq 0 \end{cases}$$

The 2D-DCT is, however, separable; two 1D-DCT operations may be performed on each dimension of $\mathbf{F}$ sequentially. The form of the one-dimensional Equation 3.2 is still the same as Equation 3.1. In practical terms, a transitional matrix is produced by performing a 1D-DCT on each

row of the input MCU, after which each column of the transitional matrix undergoes a 1D-DCT process, resulting in the 2D-DCT. By separating the two parts the hardware required to perform the transform can be simplified. Each 1D-DCT is simpler than the 2D-DCT, and by buffering the transitional matrix, the row-DCT, buffer and column-DCT form a pipeline ideal for a real-time hardware implementation.

$$\mathbf{g}_u = \sum_{x=0}^{7} \alpha(u)\mathbf{f}_x \cos\left[\frac{\pi}{8}\left(x + \frac{1}{2}\right)u\right] \tag{3.2}$$

3.1.1 1D-DCT Design

As the DCT algorithm is widely used throughout the field of image compression, there has been extensive study of its implementation since its conception. A very efficient algorithm for a JPEG-compatible DCT is that of Loeffler et al. [6, 38]. A scaled version of this algorithm is presented by Kovac and Ranganathan [39], which consists of a 1D-DCT with only 5 multiplications and 29 additions and subtractions as depicted in Figure 3.2 [6]. This already presents a huge advantage in reducing the computation over the regular form of the 1D-DCT, shown in Equation 3.2, that consists of 64 multiplications and 64 additions. As depicted in this figure, each element of the output vector, $\mathbf{g}$, is a linear combination of the elements of the input vector, $\mathbf{f}$. Another vector, $\mathbf{h}$, is introduced in this figure as an intermediate between $\mathbf{f}$ and $\mathbf{g}$. The elements of $\mathbf{f}$ are paired off, and the sums and differences of these 4 pairs constitute $\mathbf{h}$ as shown in Equation 3.3.

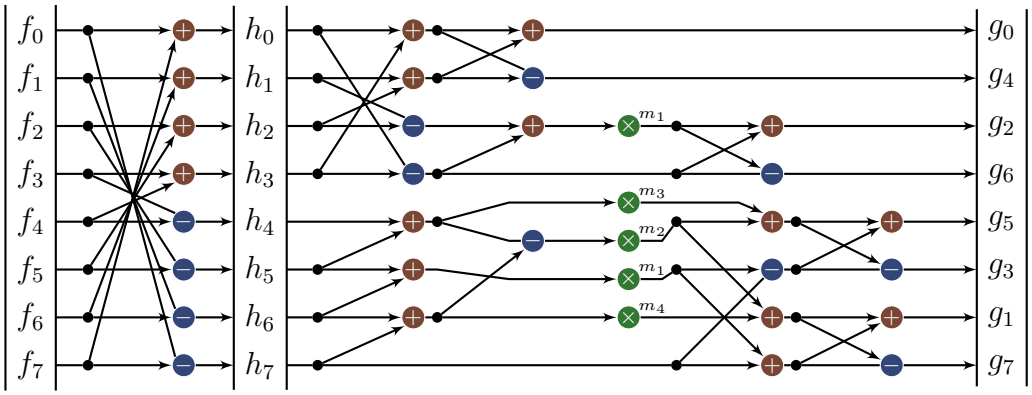


Figure 3.2: Scaled DCT. Constant multiplicands are $m_1 = \cos\left(\frac{2\pi}{8}\right)$, $m_2 = \cos\left(\frac{3\pi}{8}\right)$, $m_3 = \cos\left(\frac{\pi}{8}\right) - \cos\left(\frac{3\pi}{8}\right)$, $m_4 = \cos\left(\frac{\pi}{8}\right) + \cos\left(\frac{3\pi}{8}\right)$.

$$\mathbf{h} = \begin{pmatrix} f_0 + f_7 \\ f_1 + f_6 \\ f_2 + f_5 \\ f_3 + f_4 \\ f_0 - f_7 \\ f_1 - f_6 \\ f_2 - f_5 \\ f_3 - f_4 \end{pmatrix} \tag{3.3}$$

From this point $\mathbf{h}$ is treated as the input vector. The vector $\mathbf{h}$, like $\mathbf{f}$, combines linearly to produce each element of $\mathbf{g}$ (Equation 3.4). Note that to simplify the coefficient matrix, C, the elements of $\mathbf{g}$ are reordered as $\mathbf{g}'$ defined in Equation 3.5.

$$\mathbf{g}' = \mathbf{C} \cdot \mathbf{h} \tag{3.4}$$

$$\mathbf{g}' = \begin{pmatrix} g_0 & g_2 & g_4 & g_6 & g_1 & g_3 & g_5 & g_7 \end{pmatrix}^T \tag{3.5}$$

Following through Figure 3.2 (modified from Bailey [6]) the coefficient matrix is defined in Equation 3.6, divided into two blocks, $\mathbf{C}_{sum}$ and $\mathbf{C}_{dif}$,

corresponding to the coefficients of the pairs of sums and the pairs of differences in $\mathbf{h}$ respectively.

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{sum} & 0_{4,4} \\ 0_{4,4} & \mathbf{C}_{dif} \end{bmatrix}$$

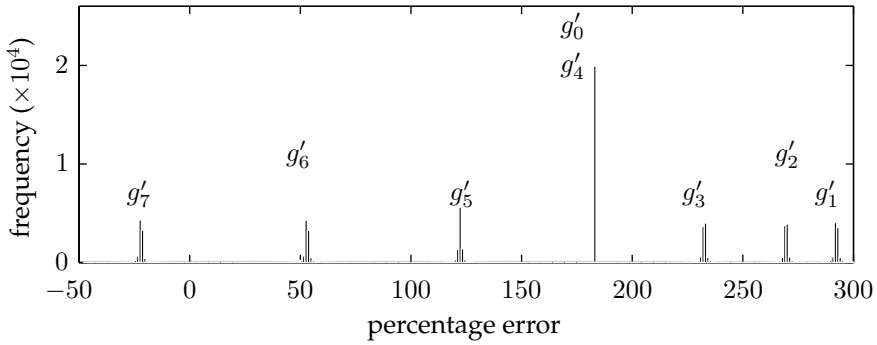
$$\mathbf{C}_{sum} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1-m_1 & -m_1 & m_1 & 1+m_1 \\ 1 & -1 & -1 & 1 \\ -1+m_1 & m_1 & -m_1 & 1-m_1 \end{bmatrix} \tag{3.6}$$

$$\mathbf{C}_{dif} = \begin{bmatrix} m_2 & m_1+m_2 & m_1-m_2+m_4 & 1-m_2+m_4 \\ -m_2-m_3 & -m_1-m_2-m_3 & -m_1+m_2 & 1+m_2 \\ m_2+m_3 & -m_1+m_2+m_3 & -m_1-m_2 & 1-m_2 \\ -m_2 & m_1-m_2 & m_1+m_2-m_4 & 1+m_2-m_4 \end{bmatrix}$$

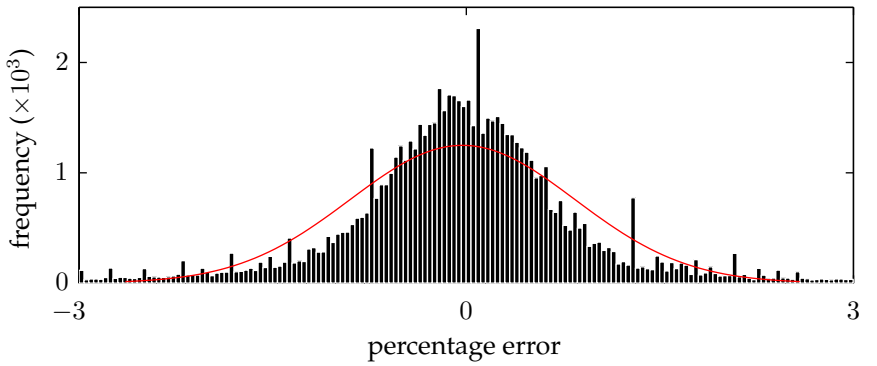
The coefficients of $\mathbf{C}$ are evaluated approximately in Equation 3.7.

$$\mathbf{C} \approx \begin{bmatrix} 1.00 & 1.00 & 1.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ -1.71 & -0.71 & 0.71 & 1.71 & 0.00 & 0.00 & 0.00 & 0.00 \\ 1.00 & -1.00 & -1.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ -0.29 & 0.71 & -0.71 & 0.29 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.38 & 1.09 & 1.63 & 1.92 \\ 0.00 & 0.00 & 0.00 & 0.00 & -0.92 & -1.63 & -0.32 & 1.38 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.92 & 0.22 & -1.09 & 0.62 \\ 0.00 & 0.00 & 0.00 & 0.00 & -0.38 & 0.32 & -0.22 & 0.08 \end{bmatrix} \tag{3.7}$$

The accuracy of this approximation was tested by computing the output of Equation 3.4 in MATLAB. A sample size of 10^4 sets of 8 pixels were generated from a uniform distribution of pixel values in the range [0, 255]. The output was compared with MATLAB's built-in DCT algorithm as a percentage error as in Equation 3.8. A histogram of the percentage error elements as in Figure 3.3a was expected to show a normal distribution. It in fact reveals seven peaks, which on further investigation correspond to the indices of the output vector as indicated by the labels in Figure 3.3a.



(a) Initial DCT Algorithm



(b) Corrected DCT Algorithm

Figure 3.3: Distribution of Errors of DCT Algorithm

The mean percentage error of each index is shown in Table 3.1. As output values are a linear combination of the input values the errors may be corrected by modifying the coefficient matrix. Each row of the coefficient matrix, $\mathbf{C}$, is multiplied by the reciprocal of the percentage error of the respective index as in Equation 3.9, the modified coefficient matrix labelled as $\mathbf{C}_s$.

$$\text{percentage error} = 100\% \cdot \frac{\mathbf{C} \cdot \mathbf{f} - fun_{dct}(\mathbf{f})}{fun_{dct}(\mathbf{f})} \tag{3.8}$$

Table 3.1: Mean percentage error of each index of $\mathbf{g}'$ in Equation 3.4.

Index	Error
g_0	+182.8%
g_1	+292.3%
g_2	+269.6%
g_3	+232.7%
g_4	+182.8%
g_5	+122.4%
g_6	+53.2%
g_7	-21.8%

$$\mathbf{C}_s \approx \begin{bmatrix} 0.35 & \cdots & 0.35 \\ 0.27 & \cdots & 0.27 \\ 0.35 & \cdots & 0.35 \\ 0.65 & \cdots & 0.65 \\ 0.25 & \cdots & 0.25 \\ 0.30 & \cdots & 0.30 \\ 0.45 & \cdots & 0.45 \\ 1.28 & \cdots & 1.28 \end{bmatrix} \circ \mathbf{C} \tag{3.9}$$

An approximation of the new coefficient values are shown in Equation 3.10.

$$\mathbf{C}_s \approx \begin{bmatrix} 0.35 & 0.35 & 0.35 & 0.35 & 0 & 0 & 0 & 0 \\ -0.46 & -0.19 & 0.19 & 0.46 & 0 & 0 & 0 & 0 \\ 0.35 & -0.35 & -0.35 & 0.35 & 0 & 0 & 0 & 0 \\ -0.19 & 0.46 & -0.46 & 0.19 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.10 & 0.28 & 0.41 & 0.49 \\ 0 & 0 & 0 & 0 & -0.28 & -0.49 & -0.10 & 0.41 \\ 0 & 0 & 0 & 0 & 0.42 & 0.10 & -0.49 & 0.28 \\ 0 & 0 & 0 & 0 & -0.49 & 0.41 & -0.28 & 0.10 \end{bmatrix} \tag{3.10}$$

Again a set of 10^4 input pixel vectors was generated, and the percentage error data were collected, but now using $\mathbf{C}_s$ as the coefficient matrix. Figure 3.3b shows a histogram of the data that is clearly normal. The new percentage error has a mean of -0.3% and a standard deviation of 1.1%.

3.1.2 1D-DCT Implementation

Woods et al. [1] present a hardware implementation of the system described above, specifically for FPGAs. This implementation uses fewer resources to perform a 1D-DCT than other implementations of the same algorithm. Others, such as Agostini et al. [36] and Kusuma and Widodo [37], have based their designs on the above algorithm, but these are more complex. This is due to the focus on minimising the required multipliers of the system, as dedicated multipliers on FPGAs have previously been scarce, and using LEs for multiplication is resource intensive. However, modern FPGAs have dozens, if not hundreds of dedicated multipliers, usually 18×18 wide, hence the use of 4 such multipliers in the 1D-DCT is of little consequence. Therefore the Woods et al. system [1] was used for this design. Bailey [6] presents a detailed interpretation of this design. Other than the changes to the coefficients described above, the system described here departs from this design only in the reordering of buffers purely for the convenience of the implementation in VHDL. This design is depicted in Figure 3.4 and is described in detail below.

The functional blocks of of the 1D-DCT as labelled in Figure 3.4:

I Stack Buffer. This operates on a cycle of 8 clocks: for the first four clocks it stores data from its input, and on the subsequent four clocks it outputs the data in reverse order.

II Multiplexer. This multiplexer outputs one of its inputs for four clock cycles, then outputs the other input for the next four clock cycles.

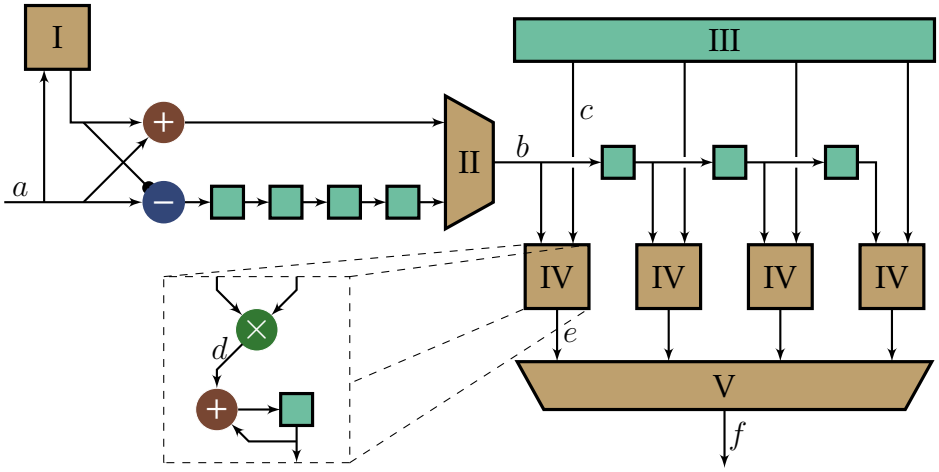


Figure 3.4: Block Diagram of 1D-DCT.

III Coefficient ROM. This ROM contains 32 elements of width w_{dc} (the exact value of which is discussed later) corresponding to the elements of the $\mathbf{C}_s$ matrix. As 8 elements correspond to each MAC unit, and always repeat in the same order, the ROM is simplified to contain 8 elements of width $4w_{dc}$, and the output bus is split in four. See Table 3.3 for the ordering of the coefficients in the ROM.

IV Multiply and Accumulate Units (MACs). These operate on a cycle of four clocks. The accumulator buffer is initially reset to 0, the two inputs are multiplied together and on each clock cycle the product is added to the accumulator. After four clock cycles one of the elements of the output DCT is obtained after which the MAC unit is reset in preparation for the next element.

V Multiplexer. On any given clock cycle one of the MAC units' outputs will correspond to one of the elements of the output DCT vector. This multiplexer outputs each of these in turn.

The signal paths of the 1D-DCT as labelled in Figure 3.4:

a The pixel values arrive in row order. They have been shifted to the range [-128,127]. Each set of 8 consecutive pixel values corresponds to the $\mathbf{f}$ vector of Equation 3.2, in ascending order. *Bit width:* 8.

b The first multiplexer selects first the top row for four clock cycles. During these cycles b equals $\mathbf{f}_0 + \mathbf{f}_7$, $\mathbf{f}_1 + \mathbf{f}_6$, $\mathbf{f}_2 + \mathbf{f}_5$ and finally $\mathbf{f}_3 + \mathbf{f}_4$. Meanwhile the equivalent outputs of the subtracter have been buffered, and are then selected by the multiplexer, in order: $\mathbf{f}_0 - \mathbf{f}_7$, $\mathbf{f}_1 - \mathbf{f}_6$, $\mathbf{f}_2 - \mathbf{f}_5$ and finally $\mathbf{f}_3 - \mathbf{f}_4$. Note that this means that b corresponds to the $\mathbf{h}$ vector of Equation 3.3, cycling through the elements in ascending order. As the range of possible values has now increased by an order of one, due to the addition and subtraction, the bus width has also increased by one. As each element is used in four of the output elements they are buffered into each of the MAC units. *Bit width:* 9.

c These are the elements of the coefficient matrix $\mathbf{C}_s$. Rather than using floating point numbers, it is more resource efficient to use integer values in multiplication and addition. However rounding means that the coefficients have a range of [-1,1]. By bit shifting these values sufficiently, precision is retained. *Bit width:* w_{dc}.

d The product of the elements $\mathbf{h}$ by their corresponding coefficients of $\mathbf{C}_s$. *Bit width:* $w_{dc} + 9$.

e The accumulation of the subsequent products of each MAC unit. After the fourth sum these are equivalent to one of the outputs of the 1D-DCT. *Bit width:* $w_{dc} + 11$.

f The output of the 1D-DCT. These are the elements of $\mathbf{g}'$ of Equation 3.5, in ascending order. These values have been bit shifted down to compensate for the bit shifting of the coefficient matrix. *Bit width:* 11 [37].

This design uses minimal resources, and furthermore is fully pipelined. As soon as one set of 8 values has been input the subsequent set of 8 values begins immediately. The system introduces a latency of 11 clock cycles; latency measures the number of clock cycles between an element arriving at the input of a system and the corresponding output being produced. This system is controlled by a simple cyclic state machine consisting of 8 states. The state determines the select line of each multiplexer, the reset lines for the MAC units (one is reset on every clock cycle), the address of the ROM and the state of the stack buffer.

In order to achieve multiplication by floating point numbers in the discrete system the coefficients are bit shifted to the left and the output of the DCT is bit shifted back to the right. Thus only integer arithmetic is required saving significantly on resources. The wider the coefficient values, the greater the accuracy of the 1D-DCT. However, as the output is rounded there are diminishing returns as the width of the bus is increased. To select an appropriate level of precision an emulation of the hardware system is constructed in MATLAB. This emulation simply applies Equation 3.4 with the appropriate level of precision during operation. The resulting integer vector is transformed back into pixel data via MATLAB's inverse DCT. Input data is selected via a uniform distribution of pixel values, and each input is tested with a range of coefficient precisions. The results of the inverse DCT are left as floating point numbers, and the absolute difference between them and the input pixel values are collected. A total of 10^5 random samples are generated, and the cumulative distribution plot of the results is plotted for each level of precision in Figure 3.5.

This plot is continuous, whereas the reality of the system is that it is discrete. That means that differences between 0 and 0.5 are be rounded to 0, between 0.5 and 1.5 to 1 and so on. Consequently errors below 0.5 have no effect on the final image. As the JPEG algorithm is lossy there are differences between the original and encoded image, therefore an error of 1 in a pixel value range of 256 is not very significant. Nevertheless it is

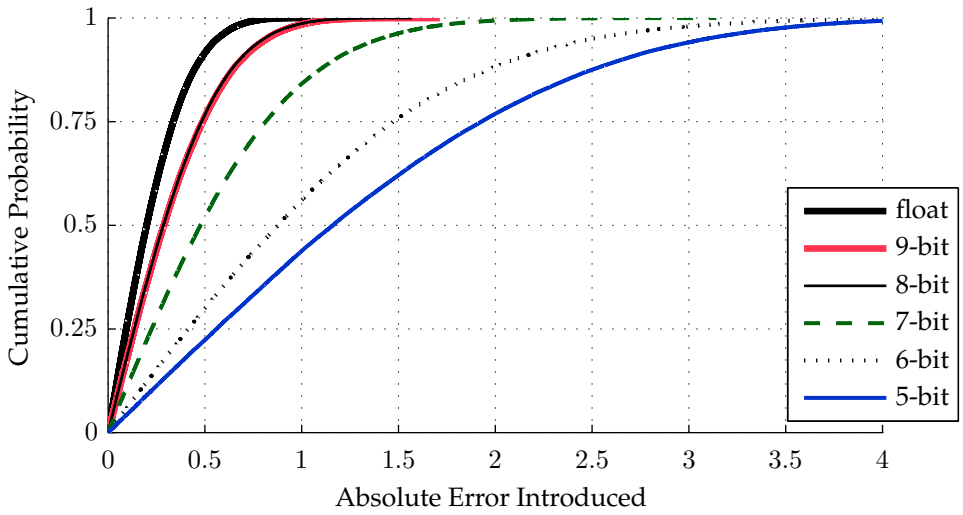


Figure 3.5: Cumulative Distribution Plot of Error of the 1D-DCT algorithm for different coefficient precisions. A floating point algorithm is included for comparison showing the error that occurs due to subsequent rounding.

desirable for this system to introduce errors less than 50% of the time. Table 3.2 shows the probability of no error being introduced for given values of w_{dc}.

Table 3.2: Probability of algorithm being accurate up to 0.5.

w_{dc} (bits)	5	6	7	8	9
probability	0.23	0.30	0.53	0.77	0.76

While no further advantage is gained by increasing the bit width of the coefficients beyond 8, 7-bit is sufficient to achieve accuracy 50% of the time. This figure will be re-evaluated with the application of the second 1D-DCT.

3.1.3 DCT Transpose Buffer

In order to achieve a two dimensional DCT both dimensions must be processed by a 1D-DCT. Because the data from the row-DCT arrives in row order, at least 7 rows of the frame must be buffered before a complete column is available for the column-DCT. In fact a 16-row buffer is used to ensure an uninterrupted flow of data into the second, column-DCT. The transpose buffer, as depicted in Figure 3.6 consists of two parts each of 8 rows of data.

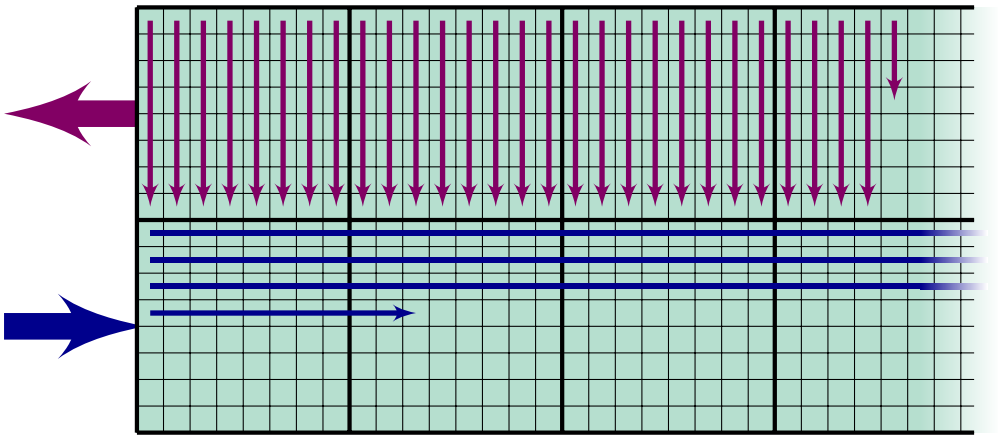


Figure 3.6: The effective operation of the transpose buffer between the two 1D-DCT modules.

This is called the *transpose* buffer as it transposes the MCU; writing in row order and reading in column order. While the data from the row-DCT is input into one half of the buffer row by row, the column-DCT is retrieving data from the other half of the buffer column by column. The two halves of the buffer operate in ping-pong mode; one is written to while the other is read, swapping roles simultaneously once the end of the half-buffer is reached.

Such a buffer will be 16 rows by 752 columns by 11-bits per datum (≈16 kB). It is appropriate to use the Block RAM of the FPGA for such a large buffer. The Block RAM cannot be rearranged into the dimensions

specified, but Quartus II provides an interface to the RAM of variable data width. A module of Block RAM of dimensions 16384 by 11-bits is used as the buffer of Figure 3.6. The read and write addresses of the block RAM match, which is not how the transpose buffer behaves. A system to select the appropriate address is devised.

The address width of the buffer is $w_{tbwa} = 14$, the MSB (most significant bit) of which is used to distinguish the two halves of the buffer. The write address, $tbwa$, increments by 1 on every clock cycle until the address reaches $tbwa = 8 \times 752 - 1 = 01\,0111\,0111\,1111$, after which point the data begins writing to the second block, that is, the address becomes $tbwa = 8 \times 752 - 1 = 10\,0000\,0000\,0000$. The read address is significantly more complicated as it must be incremented column by column; the address is increased by 752 to traverse down a column, and $(7 \times 752 - 1)$ subtracted from the address to go to the start of the next column.

$$tbra[12:0] = (tbwa[12:0] \bmod 8) * 752 + (tbwa[12:0]/8) \tag{3.11}$$

Two processes to calculate an appropriate read address, $tbra$, are proposed. The first is a calculation based upon the write address where the MSB of the address is inverted to change blocks and Equation 3.11 calculates the rest of the address. While division by 8 is a *cheap* operation in hardware, as it is a bit shift, the multiplication is a more expensive operation.

A second method to find the address is proposed to target this issue. A 3-bit counter marks the current position within a column. This column counter is incremented every clock cycle, while simultaneously the address is updated. When the column counter is in the range [0,6] the address is increased by 752, and when the column counter equals 7 the address is reduced by 5263. Both of these systems are used successfully, however analysing the FPGA resource usage of each alternative showed that the former in fact uses less, the latter still incurring the use of a multiplier when compiled.

3.1.4 2D-DCT Considerations

The column-DCT is almost identical to the row-DCT module. The primary difference is that the widths of all the signal paths are greater, as the incoming data is 11-bit, rather than 8-bit. This does not apply to the coefficients as they do not lie on the data pipeline. However, the coefficients are again tested for precision against accuracy, just as in Section 3.1.2.

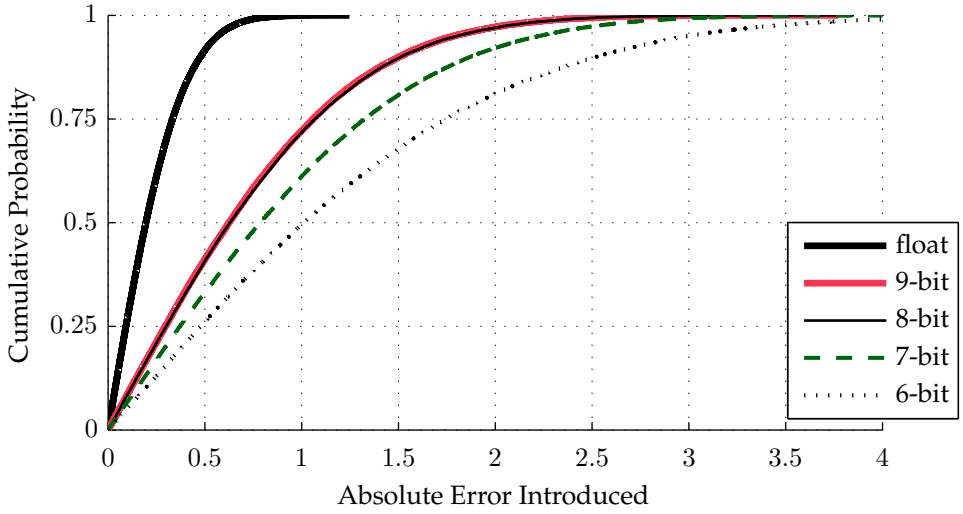
Once more 10^4 samples are generated, but in this case each sample consists of an 8×8 matrix of pixel values. A transitional matrix is formed by performing the row-DCT upon the rows of a sample. The row-DCT uses 7-bit coefficients as per Section 3.1.2. The columns of the transitional matrix undergo a column-DCT to achieve the 2D-DCT output. MATLAB's inverse 2D-DCT is applied to this, and the result is compared with the original sample. A cumulative distribution plot of the errors is produced for Figure 3.7a for several different coefficient precisions in the column-DCT. In no case are more than 50% of the results within 0.5 of expected value.

To solve this, the precision of the coefficients of the row-DCT is increased to 8. Accordingly the results of the experiment improve as demonstrated in Figure 3.7b, which shows that using 8-bit coefficients in both the row- and column-DCTs achieves the desired accuracy.

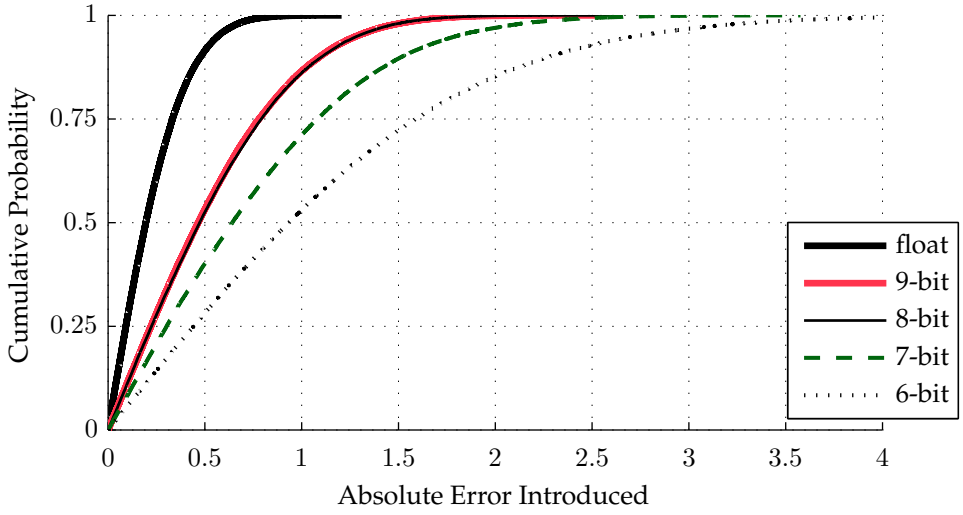
The coefficient ROM must therefore be populated with the coefficients of $\mathbf{C}_s$ of appropriate precision. The coefficients are ordered so that the corresponding coefficient is available to match the arrival of each element of $\mathbf{h}$ at each MAC unit. This ordering, along with the values to 8-bit precision are presented in Table 3.3.

As the ROM has a single output these four values are concatenated into a single 32-bit value, also shown in Table 3.3, resulting in the ROM being 8 bits, or 32 bytes.

The subsequent modules of the JPEG algorithm must take into account the ordering of the output of the 2D-DCT. The transform is shown in Equation 3.12 illustrating the reordering of the elements. The 2D-DCT outputs



(a) With 7-bit coefficients in the row-DCT



(b) With 8-bit coefficients in the row-DCT

Figure 3.7: Cumulative Distribution Function plot of the absolute value of the error of the 2D-DCT algorithm for different coefficient precisions of the column-DCT. A floating point algorithm is included for comparison showing the error that occurs due to subsequent rounding.

Table 3.3: Values of the coefficient ROM. The four signed decimal values that occur each clock, and the corresponding binary concatenation. Due to the timing from the buffers in the system the k_0 corresponds to the 4^{th} and 0^{th} rows of $\mathbf{C}_s$, k_1 to the 5^{th} and 1^{st} rows, staggered by one clock cycle, and so on.

clk	k_0	k_1	k_2	k_3	k
0	25	118	-90	118	00011001011101101010011001110110
1	71	-71	90	-118	01000111101110010101101010001010
2	106	-125	106	49	01101010100000110110101000110001
3	125	-25	25	-125	01111101111001110001100110000011
4	90	106	-126	106	01011010011010101000001001101010
5	90	-118	71	-71	01011010100010100100011110111001
6	90	-49	90	25	01011010110011110101101000011001
7	90	49	-90	-49	01011010001100011010011011001111
Indices	$C_{*,4}$	$C_{*,5}$	$C_{*,6}$	$C_{*,7}$	
of C_s	$C_{*,0}$	$C_{*,1}$	$C_{*,2}$	$C_{*,3}$	

the data column by column, and in non-sequential ordering. The order of data piped out of the 2D-DCT is indicated in G′ of Equation 3.12 as read from left to right, top to bottom.

$$\mathrm{F}' = \begin{bmatrix} f_{0,0} & f_{0,1} & \cdots & f_{0,7} \\ f_{1,0} & f_{1,1} & \cdots & f_{1,7} \\ \vdots & \vdots & \ddots & \vdots \\ f_{7,0} & f_{7,1} & \cdots & f_{7,7} \end{bmatrix} \rightarrow \mathrm{G}' = \begin{bmatrix} g_{0,0} & g_{0,2} & g_{0,4} & g_{0,6} & g_{0,1} & g_{0,3} & g_{0,5} & g_{0,7} \\ g_{2,0} & g_{2,2} & g_{2,4} & g_{2,6} & g_{2,1} & g_{2,3} & g_{2,5} & g_{2,7} \\ g_{4,0} & g_{4,2} & g_{4,4} & g_{4,6} & g_{4,1} & g_{4,3} & g_{4,5} & g_{4,7} \\ g_{6,0} & g_{6,2} & g_{6,4} & g_{6,6} & g_{6,1} & g_{6,3} & g_{6,5} & g_{6,7} \\ g_{1,0} & g_{1,2} & g_{1,4} & g_{1,6} & g_{1,1} & g_{1,3} & g_{1,5} & g_{1,7} \\ g_{3,0} & g_{3,2} & g_{3,4} & g_{3,6} & g_{3,1} & g_{3,3} & g_{3,5} & g_{3,7} \\ g_{5,0} & g_{5,2} & g_{5,4} & g_{5,6} & g_{5,1} & g_{5,3} & g_{5,5} & g_{5,7} \\ g_{7,0} & g_{7,2} & g_{7,4} & g_{7,6} & g_{7,1} & g_{7,3} & g_{7,5} & g_{7,7} \end{bmatrix} \tag{3.12}$$

To illustrate that the scaled 2D-DCT is as effective in this application as a true DCT, a comparison is made in simulation. A complex greyscale image underwent the approximated DCT process, and is then inverted

using a true DCT inversion. The original and modified images are shown in Figures 3.8a and 3.8b respectively.

(a) Original **(b)** 2D-DCT

Figure 3.8: Comparison of original image with one which has undergone the DCT algorithm used by this system.

The resultant image is visually indistinguishable from the original. A qualitative analysis reveals that pixel values differ from the original by at most 3, and 73% of pixels differ by no more than 1. This demonstrates that while the DCT system presented here does not produce completely accurate results, they are sufficiently accurate to be undetectable by humans.

3.1.5 Implementation and Testing

To implement the 2D-DCT in an FPGA, it must be written in a form translatable to a configuration file, in this case using the language VHDL. Before the module is developed the testbench is written. The VHDL testbench extracts pixel data from a file, called the *Random Bitmap File* (RBF), instantiates a 2D-DCT module, and pipes the data into the module, one pixel per clock cycle. The testbench also extracts the output of the 2D-DCT and stores it in a second file, the *Output Test File* (OTF).

A function is developed for MATLAB built on the code used for the experiments earlier in this chapter. This function, *myDCT2*(), performs the 2D-DCT operation using Equation 3.4. By using the appropriate coefficients, rounding at each stage, bit shifting the output and reordering to match Equation 3.12 this function is able to produce an output identical to that expected from the hardware implementation. The test procedure begins by generating the RBF from 752 by 480 uniformly distributed pixel values in the range [0,255]. This file is run through the testbench and the resultant OTF collected. A MATLAB script takes the RBF and OTF files, performs myDCT2() on the data of the RBF, and compares the output with the OTF. If the 2D-DCT implemented in VHDL is correct then the script found the outputs identical.

The development process for the 2D-DCT in VHDL was incremental. Initially only a single 1D-DCT was developed, and alongside it a 1D version of the test process. By using the simulation software, ModelSIM, on the VHDL testbench, bugs in the system can be identified by comparing intermediate results with those expected. After several debugging cycles the 1D-DCT passed testing. The 2D-DCT module was then developed using two of the 1D-DCTs, specifying different path width parameters for each, and the transpose buffer. Again simulations were used to identify bugs in the system. The myDCT2() function was modified to provide intermediate states of the data, such as the inputs and outputs of the transpose buffer, and these were compared with those found in the timing diagrams of the simulation. The debugging process was repeated until the test concluded that the two results were identical.

The complete 2D-DCT introduces a latency of 6031 clock cycles, primarily due to the transpose buffer.

3.2 Quantisation

The design requirements of the project specify a 70% quality level JPEG compression as per the IJG specification in Section 2.2.3. This quality level is determined by the quantisation module. In this case the DCT-MCU is divided by the elements of the $\mathbf{Q}_{70}$ matrix in Equation 3.13.

$$\mathbf{Q}_{70} = \begin{bmatrix} 10 & 7 & 6 & 10 & 14 & 24 & 31 & 37 \\ 7 & 7 & 8 & 11 & 16 & 35 & 36 & 33 \\ 8 & 8 & 10 & 14 & 24 & 34 & 41 & 34 \\ 8 & 10 & 13 & 17 & 31 & 52 & 48 & 37 \\ 11 & 13 & 22 & 34 & 41 & 65 & 62 & 46 \\ 14 & 21 & 33 & 38 & 49 & 62 & 68 & 55 \\ 29 & 38 & 47 & 52 & 62 & 73 & 72 & 61 \\ 43 & 55 & 57 & 59 & 67 & 60 & 62 & 59 \end{bmatrix} \tag{3.13}$$

However, as discussed in Section 3.1.1, tighter control of resources may be achieved by real number multiplication using bit shifting. The elements of the $\mathbf{Q}_{70}$ table are inverted to find the matrix, $\mathbf{Q}_{70}^{-}$. The quantised-MCU, $\mathbf{J}$, is calculated as in Equation 3.14, by taking the Hadamard product of $\mathbf{Q}_{70}^{-}$ and the DCT-MCU, $\mathbf{G}$.

$$\mathbf{J} = \mathbf{Q}_{70}^{-} \circ \mathbf{G} \tag{3.14}$$

Similar to the multiplication in the DCT algorithm described in Section 3.1.2, the elements of $\mathbf{Q}_{70}^{-}$ are bit shifted up and rounded so that integer multiplication occurs, and then the result bit shifted back. The same problem arises: how many bits should the elements of $\mathbf{Q}_{70}^{-}$ be shifted by.

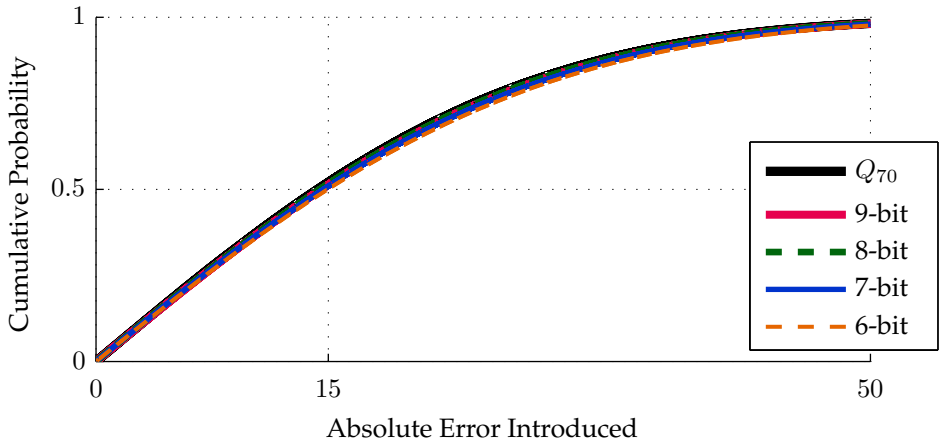
A similar experiment is performed as in Section 3.1.4. A MATLAB function is developed, *qt*(), which performs the operation of bit shifting $\mathbf{Q}_{70}^{-}$ by a value of w_{qt} to get a matrix, $\mathbf{Q}_{70}^{\prime-}$. The Hadamard product of $\mathbf{Q}_{70}^{\prime-}$ and DCT-MCU is bit shifted down by w_{qt} to give the quantised result, $\mathbf{J}$. An inverse matrix is constructed, $\mathbf{Q}_{70}^{\prime}$, by inverting the elements of $\mathbf{Q}_{70}^{\prime-}$,

bit shifting them up by w_{qt} and rounding. The matrix $\mathbf{Q}'_{70}$ is in fact the quantisation matrix that most closely matches the result of the operation of *qt*(), and is approximately equal to $\mathbf{Q}_{70}$.

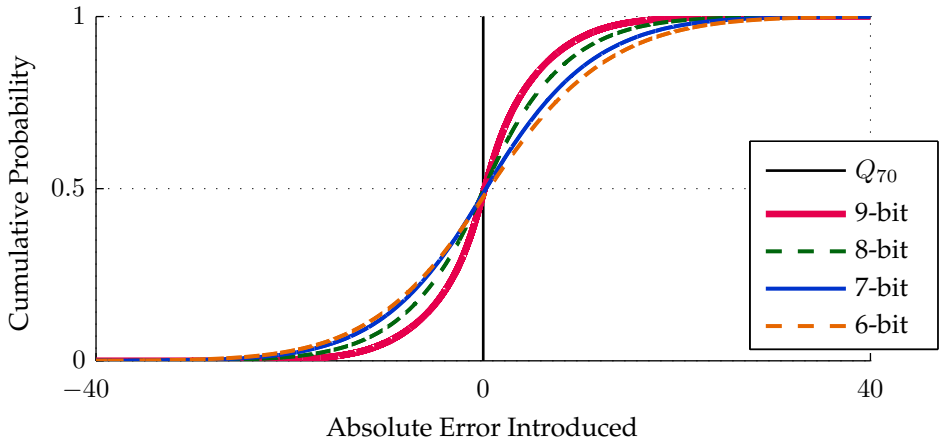
A single run of the experiment begins by generating a sample of an 8×8 matrix of uniformly distributed pixel values. This has a 2D-DCT performed on it to produce the DCT-MCU. The DCT-MCU is quantised using variations of *qt*() that differ by performing bit shifts of 6, 7, 8 and 9. Bit shifts of less than 6 result in some elements of $\mathbf{Q}'^{-}_{70}$ being 0, which leads to a value in the quantisation matrix of $\frac{1}{0}$. The resultant matrices are inverted: they are Hadamard multiplied by the appropriate $\mathbf{Q}'_{70}$ matrix, followed by the inverse 2D-DCT to give a resultant block of pixel values. The resultant block is compared with the original sample, and the absolute difference between the individual pixel values is recorded.

The experiment is performed on 10^4 samples, and the cumulative distribution function of the data is plotted in Figure 3.9a. For comparison, the effect of using a true 70% quality quantisation is included. There is in fact very little difference in the errors introduced by the different precisions of quantisation matrix.

To further explore this, the difference between the errors of each of the results and those from $\mathbf{Q}_{70}$ are collated. This 'difference' data is depicted as a CDF in Figure 3.9b. It shows that there is little deviation from $\mathbf{Q}_{70}$, echoing the conclusion from the previous plot. However, this plot shows the variation in detail. The deviation is in fact normally distributed, with variance only slightly increasing with decreasing bit shift: a bit shift of 6 has a standard deviation of 11, whereas a bit shift of 7 has a standard deviation of 10. It makes little sense then to use a quantisation of precision greater than six bits. The corresponding quantisation matrix is given in Equation 3.15.



(a) Absolute value of the error.



(b) Deviation from $\mathbf{Q}_{70}$.

Figure 3.9: CDF plot of accuracy of quantisation against the precision of quantisation matrix.

$$\mathbf{Q}'_{70} = \begin{bmatrix} 11 & 7 & 6 & 11 & 13 & 21 & 32 & 32 \\ 7 & 7 & 8 & 11 & 16 & 32 & 32 & 32 \\ 8 & 8 & 11 & 13 & 21 & 32 & 32 & 32 \\ 8 & 11 & 13 & 16 & 32 & 64 & 64 & 32 \\ 11 & 13 & 21 & 32 & 32 & 64 & 64 & 64 \\ 13 & 21 & 32 & 32 & 64 & 64 & 64 & 64 \\ 32 & 32 & 64 & 64 & 64 & 64 & 64 & 64 \\ 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 \end{bmatrix} \tag{3.15}$$

The most significant elements, those towards the top left, match $\mathbf{Q}_{70}$ more closely than the less significant elements towards the bottom right.

3.2.1 Quantisation Implementation and Testing

The implementation of the quantisation module consists of a counter, look-up-table, multiplier and an element buffer. On each clock cycle the 6-bit counter increments, overflowing every 64 cycles. The counter provides an input to the look-up-table which is used to select the appropriate multiplicand for the current element of the DCT-MCU. This look-up-table, shown in Table 3.4, is made up of the elements of $\mathbf{Q}'^{-}_{70}$ reordered to match the order of the elements output from the 2D-DCT (see Equation 3.12). The input, from the 2D-DCT, is multiplied by the multiplicand from the look-up-table and is bit-shifted down before being stored in a buffer which is the output of the module. As there is only a single buffer, the latency of the quantisation module is 1 clock cycle.

The VHDL implementation went through a similar testing process to the 2D-DCT. A file of random sample data is produced. A VHDL testbench reads the data from the sample file, feeds it into the quantisation module, and stores the output stream to another file. A MATLAB script performs the same function and compares the results with those in the output file of the VHDL simulation. Once this test is passed, a new module is developed to link the 2D-DCT and the quantisation modules.

Table 3.4: Elements of the matrix $\mathbf{Q}'^{-}_{70}$ implemented as a look-up-table reordered to match the output of the 2D-DCT module.

Addr	0	1	2	3	4	5	6	7
0x00	6	11	5	2	9	6	3	2
0x08	8	6	3	2	8	5	2	2
0x10	6	3	2	1	5	2	1	1
0x18	2	1	1	1	2	1	1	1
0x20	9	8	4	2	9	6	2	2
0x28	8	5	2	1	6	4	1	2
0x30	5	2	1	1	3	2	1	1
0x38	1	1	1	1	1	1	1	1

The combined DCT and quantisation system goes through the same testing process. The only significant change this test introduced is to alter the initial value of the counter in the quantisation module so that it reaches 0 at the same time as the first element of the MCU arrives from the 2D-DCT.

Once the system reached a point where JPEG encoded files were able to be produced, it was discovered that there were noticeable visual artefacts in many MCUs. Further testing attributed the cause to the quantisation stage. The rounding that occurred during the quantisation was a truncation; the value was always rounded down. The artefacts were eliminated by ensuring that the rounding was always towards zero (i.e. negative numbers rounded up). The MSB is the signed bit, so is 1 for negative numbers and 0 for others. By treating this as an integer, either 1 or 0, and adding it to the output all negative numbers are incremented. This counteracts the rounding down that is inherent in the system.

No further buffering is required for this operation so the latency remains at 1.

3.3 Zig-Zag Buffer

The zig-zag buffer module changes the order of the elements within each MCU. The order in which they arrive is the ordering introduced by the 2D-DCT as depicted in Equation 3.12. The order in which they must leave is the zig-zag order of Figure 2.3.

To avoid any breaks in the pipeline stream a sufficient number of elements must be buffered. If insufficient elements are buffered either data that has not yet been read will be written over or data will be read that has not yet been updated. To simplify this, two MCU sized buffers ($8 \times 8 \times 9$ bits) are used in ping-pong mode; one is written to while the other is read, then they switch. A 6-bit counter and a look-up-table are used to select the buffer addresses in correct order. The two buffers are implemented in VHDL as a single block of RAM. This module introduces a latency of 65 clock cycles to the pipeline.

A testbench for the zig-zag buffer is written in VHDL. As this module is significantly simpler than the previous ones, the testing is also simpler. Two MCUs consisting of ascending values starting at zero are piped into the buffer module. In simulation the waveform of the outputs of the module is examined to check that the elements are in the order expected.

The zig-zag buffer module is added to the module linking the 2D-DCT and quantisation modules. An RBF is constructed and as in the previous tests goes through both a MATLAB script and VHDL simulation and the outputs are compared. As in the case of the quantisation module the counter of the zig-zag buffer is offset so that it equals zero on the arrival of the first element of an MCU.

3.4 Huffman Encoder

The Huffman encoder is distinct from the previous three modules in that it does not produce an output for every input. Furthermore the number

of inputs of the module upon which an output depends varies between one and sixty-three. Here is a summary of the operation of the Huffman encoder as described in Section 2.2.4.

1. The first element of each MCU, the DC element, is subtracted from the DC element of the previous MCU. The difference is encoded as a DC Huffman code concatenated with a DC Huffman value to form the DC Huffman element of variable length.

2. The rest of the elements of the MCU are AC elements. If an AC element is zero then a *zero counter* increments. Once a non-zero AC element occurs, or the zero counter reaches 16, an AC Huffman element is produced and the zero counter is reset. The AC Huffman element is formed by the concatenation of the AC Huffman code, derived from the AC element and zero counter, and the AC value derived from the AC element.

3. If there are no non-zero elements left in the MCU then a special end-of-block (EOB) Huffman code is used, and no further AC Huffman elements are generated for the MCU. Note that in the event that the final element of the MCU is non-zero then no EOB Huffman code is required for that MCU.

The output of the Huffman encoder must convey three items of information: the Huffman element, which utilises between 2 and 32 bits; the length of the Huffman element; and a trigger to indicate that an output has occurred. To this end, three output signals are used for each of these three pieces of information. The Huffman element output is a 32-bit bus to accommodate the extreme case. The Huffman length is a 5-bit unsigned integer which equals one less than the length of the Huffman element.

The components and flow of this system are illustrated in the block diagram of Figure 3.10. Some buffers and logic details are omitted for

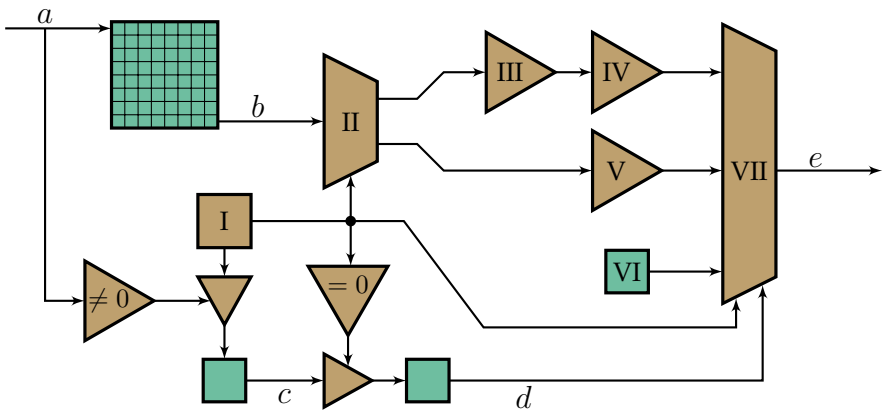


Figure 3.10: Block diagram of Huffman encoder module.

simplicity, but this diagram forms the basis of the Huffman encoder used in this JPEG encoder.

The signal paths of the Huffman encoder, as labelled in Figure 3.10:

a Next MCU. The elements of the *next MCU* arrive sequentially and are stored in a 64 element FIFO buffer.

b Current MCU. The output of the FIFO buffer are the elements of the *current MCU*.

c Index of latest non-zero element of *next MCU*. This gets updated with the current counter value whenever a non-zero element appears in the *next MCU*. This is reset whenever the counter reaches zero and another MCU arrives.

d Index of final non-zero element of *current MCU*. Once the end of the next MCU is reached, the value c becomes the index of the final non-zero element of the *current MCU* and is stored in a separate buffer.

e The Huffman encoded data. This consists of three signals: the data itself in a 32-bit bus, an unsigned integer specifying the length of the signal and the data valid trigger.

The functional blocks of the Huffman encoder, as labelled in Figure 3.10:

I 6-bit counter. This is used for tracking the index of the MCUs. This is used to distinguish DC elements from AC elements and mark the position of the final non-zero element of the MCUs.

II Demultiplexer. This determines whether the current element of the *current MCU* is DC or AC by the current counter value. If it is AC then the data is streamed towards the zero counter. If it is DC it is streamed towards the DC element encoder, and the enable line for the DC element encoder is pulsed high.

III Zero counter. Checks AC elements as they arrive for zeros. If a zero occurs a zero count is incremented. If a non-zero element arrives then it is passed along with the current zero count to the AC element encoder. The zero counter is reset after every valid element. If 16 consecutive zeros occur then it enables the AC element decoder with the element 0 and a zero count of 15.

IV AC element encoder. This takes the AC elements and zero run counts and produces the appropriate Huffman code. It also outputs the length of the code, and a high on the output trigger.

V DC element encoder. Similar to the AC element encoder. Finds the difference between the current DC element and the previous one and encodes it.

VI EOB element. This is effectively a ROM stored in an LE. This contains the end-of-block Huffman code, and code length for outputting if the multiplexer determines it is necessary.

VII Output multiplexer. This multiplexer enters four modes. It outputs the DC Huffman data on the first clock cycle of the current MCU. It

then outputs the AC Huffman data until the last non-zero element is reached, or a new MCU begins. Following the final non-zero element the multiplexer selects the EOB buffer. For the rest of the MCU the multiplexer holds the output data trigger low.

The necessity of buffering an entire MCU means that this Huffman encoder design has a total latency of 67 clock cycles. The functional blocks **I**, **II**, **III**, **VI** and **VII** are each comparatively simple. The AC and DC element encoders are, however, a little more complex, so their design is explained here in more detail.

3.4.1 AC and DC Encoder Modules

The role of the AC and DC encoder modules is to produce the individual Huffman data elements. Both encoder modules are distinct, but they have sufficient in common to be described together. Unless otherwise specified this discussion applies to both. As stated in Section 2.2.4 the encoder may use whatever Huffman tables it wishes for encoding. The investigation of an optimised Huffman table for the application considered here is beyond the scope of this project, therefore the use of the example Huffman tables contained in Annex K of the JPEG specification [14] were selected. These are reproduced in this document for convenience: the DC Huffman table is Table 2.5, and the considerably longer AC Huffman table is in Appendix A.

The design for the AC and DC element encoders is shown in Figure 3.11.

Each block has a quantised element input in addition to reset and enable lines (not shown), and the AC encoder also has a zero count input. The DC encoder stores each element, and subtracts it from the previous element before passing the difference on to the two look-up-tables. The AC encoder sends the quantised element directly to both of its look-up-tables, along with the zero count. The two look-up-tables are here referred

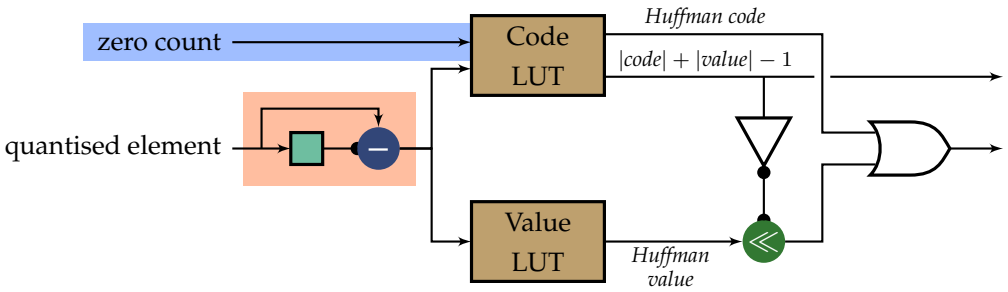


Figure 3.11: Block diagram of AC and DC element encoders used in the Huffman encoder module. Blue marking denotes AC only, and red marking denotes DC only.

to as the *value LUT*, and the *code LUT*.

The *value LUT* takes the quantised element, a signed 9-bit integer, and outputs an equivalent Huffman value according to Table 2.4. The Huffman values vary in length, so when they are piped into the 32-bit bus the most significant bits are padded with 0's. Due to the nature of the Huffman value table, positive integers require no modification; they are concatenated with twenty-three 0's. The conversion of the negative numbers is a little more complex; first an addend must be selected. Possible addends come from the series $2^i - 1$ where $i \in \mathbb{N}$. The smallest element of this series, greater than or equal to the absolute value of the quantised element, is selected. For example the quantised element -5 would correspond to an addend of 7, while the quantised element -8 would correspond to an addend of 15. The addend is added to negative quantised elements and the result is padded with zeros as with the positive elements.

The *code LUT* is purely a look-up-table. Depending on the quantised element, and in the case of the AC encoder the zero count, the appropriate Huffman code is selected from the table and output, along with a length. The length is the sum of the length of the Huffman data element (the concatenation of the Huffman code and value) minus one. The length is stored as a 5-bit unsigned integer, ranging from 0 to 31, and is an output of the

quantised element $= x$
zero count $= y$
Huffman code(x, y) $=$ CCCCC
= CCCC C000 0000 0000 0000 0000 0000 0000
Huffman code length $= 5$
Huffman value(x) $=$ VVVVVVVVV
= 0000 0000 0000 0000 0000 000V VVVV VVVV
Huffman value length $= 9$
total length $-1 = 13 =$ 01101
inversed length $=$ 10010 $= 18$
Huffman value$(x) \ll 18 =$ 0000 0VVV VVVV VV00 0000 0000 0000 0000
Huffman data element $=$ Huffman value$(x) \ll 18$ OR Huffman code(x, y)
= CCCC CVVV VVVV VV00 0000 0000 0000 0000

Figure 3.12: An example of the operation of the AC element encoder.

encoder. The Huffman code itself ranges in length from 2 to 16 bits and uses a 32-bit line with the least significant bits padded with 0's.

The length is subtracted from 31 by way of a bitwise inversion. This resultant number is used as the input to a left bitshift operation upon the Huffman value. Bitshifting by a fixed quantity is simple in hardware; the signal bus is routed to select the appropriate bits of the data and fill the rest with 0's. A variable bitshift must be performed by way of a look-up-table selecting which of the hardcoded fixed length bitshift operations to route the signal through. Such hardware systems are referred to as barrel shifters.

The bitshifted Huffman value is combined with the Huffman code by way of a bitwise OR. The resultant signal is the concatenation of the two parts filling the most significant bits of the 32-bit signal, and padded to the right with 0's. Why this works is best demonstrated by way of example as in Figure 3.12.

3.4.2 Huffman Encoder Testing

Compared with the other JPEG modules, writing an automated test system for the Huffman encoder is more challenging. The test scripts for these modules were achieved in MATLAB using the built in matrix operations. On the other hand the Huffman encoder requires manipulation of individual bits, a task that MATLAB is not well equipped for. An automated test could have been written in another language or environment, but the long Huffman tables would need to be hard coded, itself a lengthy process. Instead it was decided to eschew an automated process in favour of manually encoding data and comparing the results by hand.

Errors in the encoder module are likely to come from two sources. First, a bug in the logic and control of the encoder would be obvious within a single MCU in the case of the AC encoder, or within a couple of MCUs in the case of the DC encoder. The second source of error is from a miscoded table value, which would only be discovered by going through every single possible table entry. If a mistake were made in the tables it would become apparent in due course and is easily remedied.

Due to the nature of the potential errors, and that the testing process would be manual, a much smaller test sample was prepared for the Huffman encoder than the previous modules. A sample of 3 random MCUs, which had had DCTs performed on them before being quantised, were used. A VHDL testbench piped the sample data through the Huffman encoder and the results are compared with those calculated by hand. While there were some errors in the control systems of the encoder, these are isolated by comparing the simulated waveforms of the internal signals of the Huffman encoder with the working that had been done to calculate the Huffman elements by hand. Once isolated, errors are fixed.

3.5 Testing JPEG Encoder

Having tested each constituent module of the JPEG encoder, and tested the first three modules together, a test of the overall system was necessary. It was decided that an appropriate test would be to encode images taken by the Blackeye II camera. To perform this operation a VHDL testbench is written. To make them simple for the testbench to read, the bitmap image files are reformatted into text files forming a space-separated list of integers with line breaks at the end of each row. Thus the testbench under simulation pipes continuous pixel data in to the encoder module.

The significant problem is how to extract the output data of the encoder. The data is stored in a text file as it comes out of the encoder, a space separated list that alternates between the data itself, in hexadecimal, and the data width, as an integer. This text file can be processed by another program to create a JPEG file.

The text file produced by the testbench contains the JPEG data, but not in a form readable by image viewing software. A program to reformat the JPEG data and add the appropriate file data is written called *jfifwrapper*. The program is written in C due to the nature of the data manipulation required and the familiarity of the language.

The nature of the JFIF file is described in Section 2.2.5, and the exact contents used in this case is detailed in Appendix B. The *jfifwrapper* program writes this file data to a file followed by the condensed JPEG data and terminates on an end of image marker (0xFFD9). The file data is hard-coded, leaving the condensing of the JPEG data as main task of the program. The condensing process is a case of taking all the data output from the JPEG encoder, stripping all the padded 0's and concatenating it all in to a continuous data stream. Furthermore it has to keep track of byte boundaries; a byte entirely of 1's requires an extra byte of 0's stuffed immediately afterwards. Finally it has to pad the end of the stream with 1's so that the stream is an integer number of bytes in length.

The successfully encoded images are viewable by a wide range of software. Initial JPEG files were not viewable at all due to formatting errors caused by *jfifwrapper*. Once these were resolved, images were viewable but still corrupt; they contained errors in the data that made the image either distorted or unrecognisable. To gain insight in to the cause of the errors in the files, a program called *JPEG Snoop* is used. This software is used for JPEG file analysis, and can pinpoint down to the bit where an unexpected value occurs, and what the nature of the error is. This is of great use in removing bugs from *jfifwrapper* and the JPEG encoder.

After several iterations three successfully encoded images are produced by a single testbench run, the images being encoded sequentially.

3.6 Overview of JPEG Encoder Implementation

The JPEG encoder module consists of four sub-modules: 2D-DCT, quantisation, zig-zag buffer, and Huffman encoder. These were all implemented in VHDL, and successfully tested in simulation to produce a JPEG encoding of a 752×480 8-bit greyscale image. The IJG quality level of the encoding process is hardcoded to be 70%.

The encoder requires the pixels to arrive sequentially without break row by row. The output data is of variable bit width up to a maximum of 32 bits, therefore the output data uses a 32-bit signal, padding the least significant bits with zeros. In parallel another output signal provides the width of the output data, this width is a 5-bit unsigned integer. An enable signal is also output to indicate when data is available. Clock and reset signals are required; each module has elements that must be reset immediately prior to the commencement of encoding. These are counters used to track the state of the data, or in the case of the DC encoder the storage of previous data.

The form of the JPEG encoder module is summarised in Figure 3.13.

Of note is the latency, which totals 6164 clock cycles. This latency is

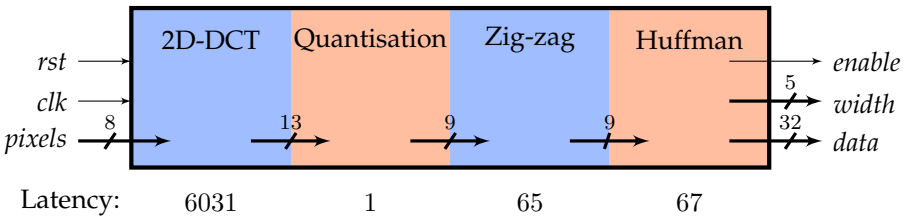


Figure 3.13: The JPEG encoder: its constituent modules and its interfaces.

primarily due to the necessity of buffering so many rows of data during the first stage of the encoding process. The JPEG encoder at this point is ready to be integrated in to a greater hardware system.

Chapter 4

System Implementation

The previous chapter describes the design of an FPGA based JPEG encoder. This chapter puts the encoder into the context of the wider system and begins by describing the image sensor and its interface, in Section 4.1. Section 4.2 outlines the role of the FPGA in the camera system, and describes the hardware used for testing this project. This is followed by Section 4.3, which describes the design of the FPGA modules that interface with the system's microprocessor. A discussion of methods to reduce the system's dynamic power consumption can be found in Section 4.4, and one of the proposals is implemented. The chapter ends with an exploration of the potential for image processing operations to be added to the FPGA system in a modular fashion and as an example, an image sharpening process is implemented.

4.1 Image Sensor

The Blackeye II camera system is available in various versions, which employ a variety of CMOS active pixel sensors (APSs). This project focuses on the sensor most commonly employed: the MT9V034 from Aptina. If the FPGA system needs to be employed with a different image sensor in the future, it is anticipated that this design will be able to be adapted to

that sensor.

Electronic image sensors based upon nMOS, pMOS and bipolar technology were developed in the 1960s [40]. Due to innate noise in these early sensors alternative technologies were sought, and one of promise was proposed in 1970: charge-coupled device (CCD). By the late 1980s this was the dominant technology in electronic imaging devices [40]. By the 1990s interest grew in CMOS based APSs which offer two significant advantages over CCDs: they use common IC manufacturing techniques, and digital and analogue processing could be integrated into individual pixels [41]. This made them cheaper to manufacture and to use in designs, as much of the interfacing electronics could be built into the device. Into the 2000s CCD based cameras maintained a superior image quality compared with CMOS sensors, but APS cameras were cheaper and therefore became increasingly used in low quality consumer electronic applications such as webcams and mobile phone cameras [42]. However, since the start of the 2010s, APS has come to surpass CCD in quality in many applications, and have therefore become more widely used [43].

Upon activation, an APS pixel will convert incoming photons into a voltage [43]. In the case of colour APS, a Bayer pattern configuration is common; half the pixels are sensitive to green light, a quarter to blue and a quarter to red [41]. The data is extracted from pixels row by row; the first pixel of each column is extracted (i.e. the first row) followed by the second pixel and so on. It is common for these sensors to include on-chip analogue to digital converters (ADCs) so that the output of the sensor can represent each pixel as a binary digital signal. This enables the sensor to directly interface with digital systems, such as those used in this project, without the need for external ADC systems, which add to the system cost and require careful design.

Besides simple control signals such as standby and reset, the MT9V034 has an I^2C interface to set and read the chip's control parameters, and an image sensor interface (ISI) to transfer the image data.

4.1.1 Image Sensor Interface

There is a lack of standardisation in the field of image sensor interfaces, and as such there are several different terms used to describe similar, and sometimes identical interfaces. In this thesis the term ISI has been chosen to refer to the interface used to transfer image data from the MT9V034 APS. The microprocessor used in the Blackeye II camera has an ISI peripheral block with dedicated pins.

The image sensor outputs the D_{OUT} bus and the *pclk*, *l_vld* and *f_vld* signals. While pixels are transmitted serially, each pixel's 10-bit binary representation is transmitted in parallel on the D_{OUT} bus. The *pclk* signal is a clock used to synchronise the interface. Because transitions in the APS's output occur on negative clock transitions, the data should be read by the FPGA on positive clock transitions. The function of *l_vld* and *f_vld* is identical to the video display signals HSYNC and VSYNC in VGA and its derivatives. Pixels are output serially along each row from left to right, going along rows from the top to the bottom of the frame. There are periods in which no data is output, called blanking periods. In the case of this sensor, these occur between rows (the equivalent of 57 pixels in duration) and between frames (the equivalent of 19 rows in duration). To distinguish between active periods and blanking periods, two signals are used. Blanking periods between rows (or lines) are indicated by the *l_vld* going low, and similarly blanking periods between images (or frames) are indicated by *f_vld* going low. This is illustrated in Figures 4.1 and 4.2.

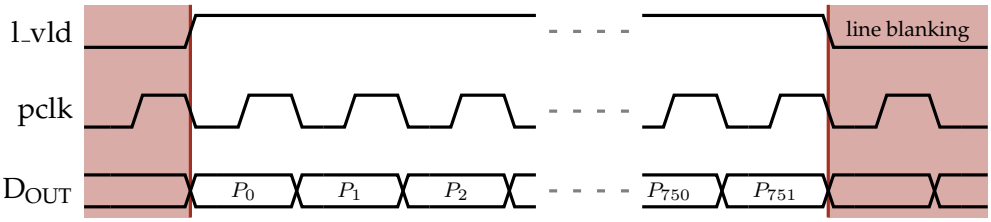


Figure 4.1: Timing examples of ISI signals: a single row. Line blanking is indicated by red.

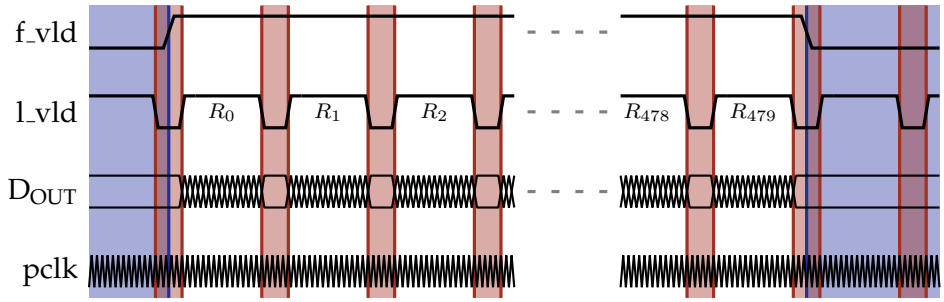


Figure 4.2: Timing examples of ISI signals: a single frame (480 rows). Line blanking is indicated by red and frame blanking by blue.

4.1.2 FPGA ISI Module

As stated in the introduction of Chapter 3, the JPEG encoder module expects a continual data stream. This is a design decision to simplify the implementation of the encoder. Furthermore the encoder needs to be reset immediately before each frame begins.

To ensure a continual data stream into the JPEG encoder, either the entire frame needs to be buffered in order to remove the blanking periods between rows, or the encoder must be paused during blanking of the input. Such a buffer is impractically large for the FPGA, and would add significantly to the latency of the system. Pausing can be easily achieved by using an enable system on the encoder module. Rather than having an enable network within the encoder module it is simpler to use the enable signal to gate the clock to the encoder. Altera recommends that clocks not be gated asynchronously, so either the enable signal should itself be clocked by the clock domain in question, or special enable systems on clock domains within Altera FPGAs should be used [44].

The FPGA enable system is utilised by the use of a *MegaFunction* within Quartus II. The enable signal is controlled by the expression (*l_vld* OR NOT *f_vld*). The clock should be disabled whenever *l_vld* is low, so that the input stream remains continuous. However, once the final row

is reached and no more data is expected, the system has no need to pause and should finish processing the data that it still holds.

In order to achieve the reset condition, a reset line for the encoder is set for one clock cycle on the first positive *l_vld* edge after a positive *f_vld* edge. This presents two problems. First, the reset signal will be set at the same time as the first data element is input. This is remedied by the simple expedient of a single element buffer on the data. Secondly, the entire module is reset at the start of each frame arriving. The latency of the encoder module is such that an entire frame will not be fully processed by the time the next frame begins, so the reset signal disrupts the previous frame.

A simple fix is to have a counter keep track of frames and block the inputs on alternate frames to allow the encoder module to finish the frame it has started. This prevents the system from processing every frame from the image sensor. However, such a frame rate is not required.

This solution is modified in Section 4.4.

4.2 Blackeye II Camera System

The Blackeye II camera system has an image sensor and a microprocessor. The image sensor outputs the image to the microprocessor which in turn performs image processing operations, compresses the image and stores the image. To provide access to images the microprocessor runs a web server for PCs to access. These roles are depicted in Figure 4.3 under the *existing system*. This project proposes an alternative system in which an FPGA supplants the microprocessor in its roles of image processing and image compression.

In the *proposed system*, the microprocessor remains the centrepiece of the design: it runs the web server for external access, it stores and retrieves the image files using an SD card, and it controls the operation of the image sensor and FPGA. The role of the FPGA is to receive image data from

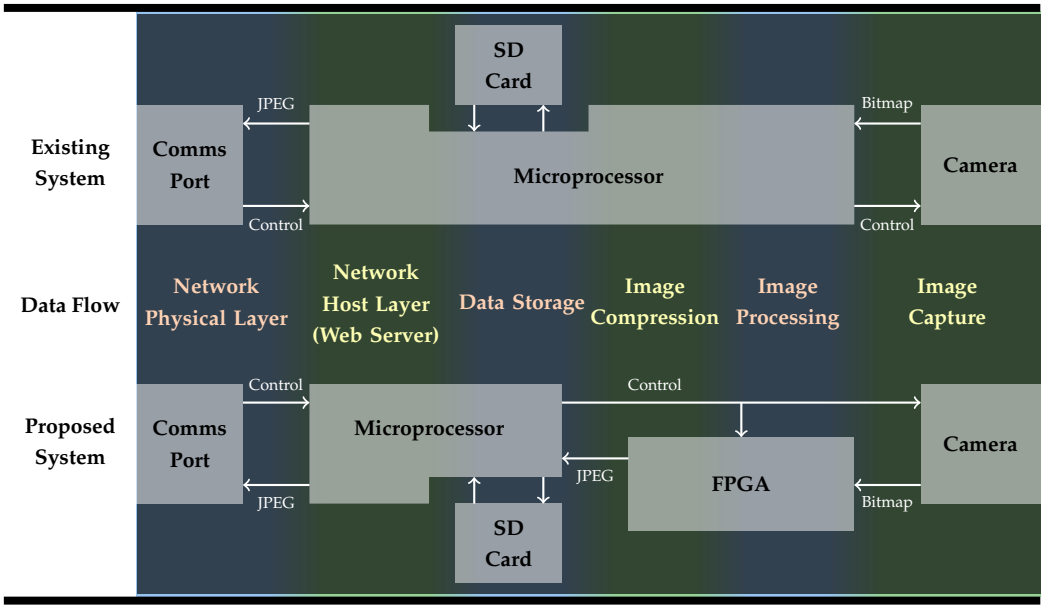


Figure 4.3: Diagram of the pre-existing Blackeye Camera system compared with the proposed system.

the image sensor, perform what image processing operations are deemed necessary by the microprocessor, encode the image as a JPEG file and send it, along with other information the microprocessor might want about the image, to the microprocessor. The necessary control structures are more complicated to design in hardware compared with software, so it is preferable for the microprocessor to retain overall control of the system.

The advantage of FPGAs for image processing are discussed in Section 2.3.2. The existing Blackeye II system is limited to recording 2 frames per second (fps) by the rate the microprocessor can process them. Shifting the image processing and compression to the FPGA has a significant potential benefit. The frame rate may be increased because the bottleneck – the image processing and encoding – is shifted away from the microprocessor. The problem with adding a new device to the system is the increased power consumption. However, reduction of the power consumption of the FPGA is a focus of the design, and is mitigated by the

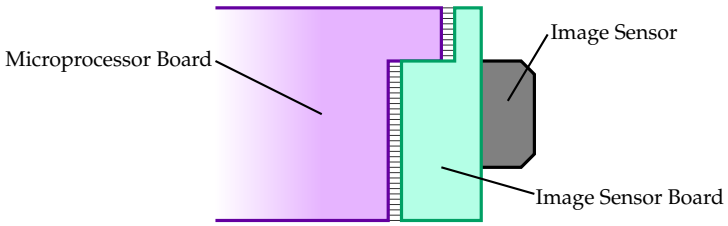
microprocessor being used less, so either the clock speed is reduced or the microprocessor is put into a low power mode between frames.

To test the FPGA system in practice, a Blackeye II camera is disassembled. The image sensor and microprocessor are on two separate printed circuit boards (PCBs) connected together. The FPGA which is used for this test is on a very small development board, the DE0-nano made by Terasic. A simple PCB is designed to fit between the image sensor board and microprocessor board and also connect to the FPGA board. Figure 4.4 shows how select signals between the image sensor and microprocessor boards are rerouted via the FPGA board, and Figure 4.5 shows a photo of the board.

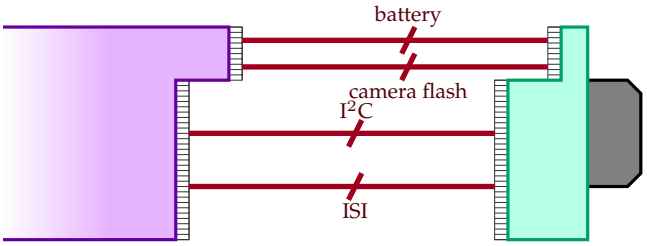
Some connections such as the battery connection and camera flash trigger are not needed by the FPGA. The battery is connected to the image sensor board but the power circuitry of the device is on the microprocessor board and these links simply connect the battery to the power circuitry. The microprocessor makes decisions on the behaviour of the system, so it decides when to and when not to use the camera flash.

In the existing system the microprocessor controls the image sensor's behaviour by using an I^2C interface to set and read registers on the chip. In anticipation of a similar register control system on the FPGA in the future, the I^2C interface is extended to it as well. While other serial communication systems used to connect ICs exist, such as UART and SPI, access to the appropriate IO ports on the microprocessor is not currently available. Furthermore, the engineers at Kinopta prefer to use a single interface for the microprocessor to communicate with the two devices, which I^2C allows because it works on a single master multiple slave basis. I^2C modules for FPGAs are common, so there are plenty of designs freely available for use, reducing the time and effort to implement the interface. Hence there is no need to develop an alternative interface.

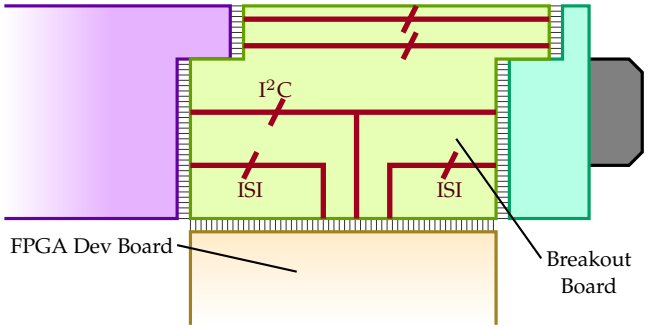
The existing system uses an ISI to transmit image data from the image sensor to the microprocessor. The signals are rerouted to the FPGA to



(a) In the Blackeye II Camera the board with the microprocessor connects to the board with the image sensor.



(b) The main signals between the image sensor board and the microprocessor board.



(c) The breakout board is inserted between the image sensor and microprocessor boards. Some signals are rerouted to the FPGA development board.

Figure 4.4: Blackeye II Camera boards, and breakout board interfacing them with the DE0 Nano FPGA development board. Not shown are power lines.

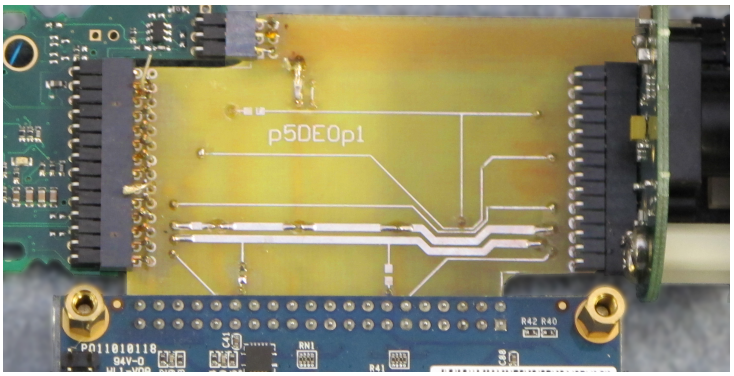


Figure 4.5: Photo of breakout board connecting to FPGA development board (bottom), image sensor board (right) and microprocessor board (left).

receive the images, and the FPGA uses a second ISI interface to transmit data to the microprocessor. Other interfaces between the FPGA and microprocessor were discussed with the other camera engineers, but on further investigation they did not provide as simple a solution as using the existing ISI of the microprocessor. The pins of the ISI port were already available on the board, and the ISI peripheral allows the microprocessor to receive data in real-time even though the software is not designed for true real-time.

4.3 Microprocessor Interface

The FPGA system provides data to the microprocessor via the microprocessor's ISI interface. The ISI peripheral system on the microprocessor chip operates independently from the CPU, and therefore may be used to receive data in real-time even though the microprocessor is not operating in a real-time fashion. Registers are used to set a frame width and height of up to 2048 each, which the ISI peripheral will buffer. When the peripheral system has finished receiving a frame it signals the CPU of its availability, which can then access the image as a block of data in memory.

The peripheral system receives *f_vld* and *l_vld* lines as well as an 8-bit

D_{OUT} bus (in fact the bus is 12 bits wide, but 4 of these are unused). In the existing system the ISI peripheral receives data from the image sensor ISI interface, so the FPGA output must conform to the image sensor's ISI interface. A constant data stream is expected, read on positive edges of *pclk* and separated at regular intervals by blanking periods as indicated by *f_vld* and *l_vld*.

4.3.1 Data Format

The dimensions of the ISI frame must be fixed before transmission. The peripheral requires *l_vld* to go high for exactly the number of clock cycles as the predefined width, and *f_vld* for the number of rows of the height. However, it does not require any uniformity in the blanking periods.

The JPEG data is not of fixed length, nor can it be divided into parts representing individual image rows. Therefore the frame size is set to dimensions sufficient to store any JPEG file likely to be generated by the FPGA system; for testing purposes this is set to 2048×256. The data to be transmitted is the JPEG encoded image in JFIF file format, and a low resolution (94×60) bitmap. The low resolution bitmap is used by an algorithm on the CPU for calculating settings for the image sensor.

The transmission begins with the JFIF/JPEG file and ends with the bitmap. The ISI data elements between these two are set to 0xFF. Because the JPEG file is not of fixed length the microprocessor must search through the data to find the EOI (end-of-image) marker (0xFFD9) that will precede the block of 0xFF. The bitmap is simple to identify as it is located in the same data address range each time.

The JFIF file format is described in Section 2.2.5. A single scan baseline sequential encoded file can be divided into three distinct parts:

Header

This contains the SOI (start-of-image) marker, JFIF header, Huffman and quantisation tables, frame header and scan header.

Data

The JPEG data is the concatenation of the output of the JPEG encoder module (the Huffman encoded data). This data will always be an integer number of bytes in length.

Footer

The footer is the EOI marker, so is always two bytes long.

If the encoding method, quantisation table and Huffman tables do not change, then the Header need not change either. It is hardcoded into the system in a ROM. See Appendix B for details of the Header.

This specification of the FPGA output will be used to design the output module.

4.3.2 Output Module — Structure and State Machine

The output module must perform many tasks, therefore it is divided into modules as shown by the block diagram in Figure 4.6. To distinguish these modules from other items of similar name they are prefixed by *output/* (for example *output/header*).

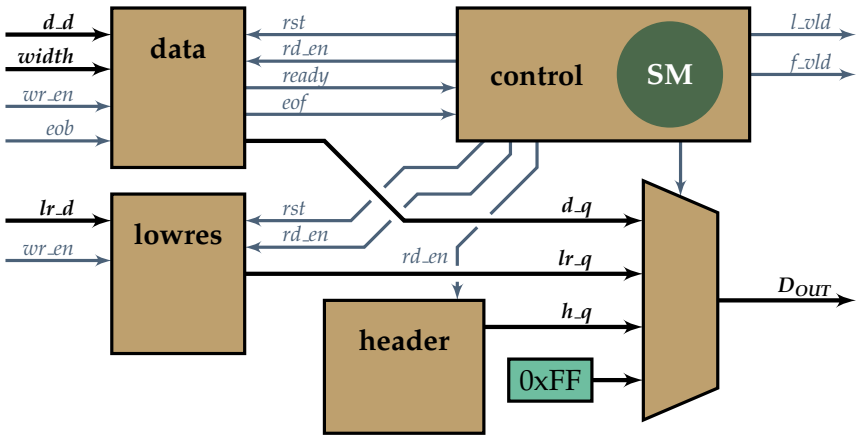


Figure 4.6: Block diagram of output module.

The module *output/control* keeps track of the output process; it sends signals to the other modules and controls what data is sent to the output. A cyclic finite state machine (SM in Figure 4.6) is devised upon which *output/control* is based. Table 4.1 outlines the different states and the flow between them.

Table 4.1: Output module state machine.

State	*f_vld*	Output	Transition
s_idle	**0**	Does not matter (0xFF).	**s_header**: Sufficient data has been received from the JPEG encoder (*ready* is high).
s_header	**1**	Data from header ROM (**h_q**).	**s_data**: Final address of ROM has been reached.
s_data	**1**	JPEG data (**d_q**).	**s_fill**: Data buffer indicates the end of JPEG frame (*eof* is high).
s_fill	**1**	0xFF	**s_lowres**: Only 5640 data elements remain in ISI frame.
s_lowres	**1**	Low resolution bitmap (**lr_q**).	**s_idle**: End of bitmap has been reached.

Each state only has one potential state to transition to. The transitions *s_idle*→*s_header* and *s_data*→*s_fill* are triggered by signals from *output/data*. The other transitions are triggered by ISI addresses: the count of the data elements in D_{OUT}. The state *s_header* transitions to *s_data* once the address 329 is reached (the size of the header). Each ISI frame consists of 524287 elements. *s_lowres* begins 5640 bytes previous to this at the address 518647. Therefore, *output/control* must count the data elements output.

The simplest of the modules is *output/header*. This is a ROM, utilising block RAM, filled with the 329 bytes of the file header. For each clock cycle in which the read enable (*rd_en*) signal is high, the internal address

counter increments, rolling over once the final address is reached. Thus, by maintaining the read enable high, the output cycles through the header data.

The low resolution bitmap required by the microprocessor does not need to be very precise. The resolution is reduced by 8 in each dimension from the original image, so each pixel of the low resolution bitmap is the average pixel value of one of the MCUs of the JPEG encoder. This simplifies the acquisition of the low resolution bitmap: the DC element of the quantised-MCU is directly proportional to the average pixel value of that MCU. The data has lost precision by this point, but is sufficient for the requirements. The Huffman encoder module is modified so that on the arrival of a DC element it is directed to the output module along with a write enable (*wr_en*) signal. The Huffman encoder is used for this purpose as it already keeps track of which elements are DC and which are AC.

Block RAM is used for buffering the 5640 elements of the low resolution bitmap in *output/lowres*. On reset the internal write address is set to zero. On a write enable the data on the *lr_d* line is written to the RAM, and the write address is incremented. On a read enable the read address is incremented, so that the output cycles through the data stored in the RAM. A reset from *output/control* is required between frames.

The *s_fill* state sends 0xFF along the data line as fast as it can (minimising the blanking periods between rows). It should be noted that this state begins by outputting 0xFFD9 (EOB marker).

Far more complex is the *output/data* module. It receives data of varying width from the JPEG encoder, which it must concatenate and store. This module requires careful design to ensure its successful operation and prevent it from using excessive resources.

4.3.3 Output/Data Module

The structure of *output/data* is built around a large segment of block RAM used to store the JPEG data until the output requires them. The block diagram in Figure 4.7 is used to discuss the module's design.

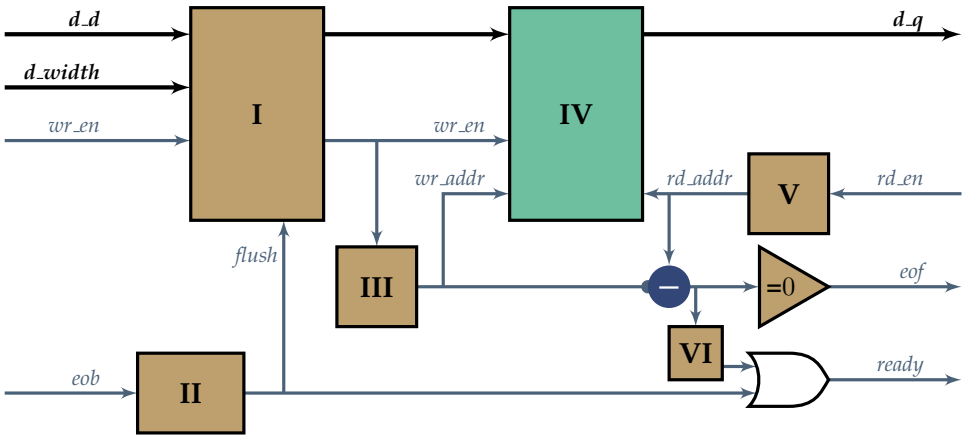


Figure 4.7: Block diagram of *output/data* module.

The functional blocks of *output/data* as labelled in Figure 4.7:

I Concatenate and accumulate unit (CAC). Concatenates the data received from the encoder using the data width bus to determine how the data fits together. Its operation is detailed below.

II Block counter. Receives a pulse from the encoder module at the end of each MCU. The output of the block remains low until the count reaches 5640 (the end of frame is reached) and it is set to 1.

III Write address counter. A counter increments the buffer write address whenever a write enable is sent from the CAC to the buffer. This starts at zero upon reset, but is allowed to overload during its operation.

IV Buffer. A large section of block RAM with independent read and write functionality.

V Read address counter. Identical in operation to the write address counter but is triggered by the read enable line.

VI Comparator. The output of the comparator is high whenever the input is greater than the ISI frame width (2048).

The input data, *d_d*, is a 32-bit bus, and the width of the data of interest is indicated by *d_width*. The CAC is designed to accumulate input data until at least 32 bits of data is concatenated together, at which point it outputs the 32-bit concatenation along with a pulse on the buffer's *wr_en* line. Once this occurs the accumulation process starts over, beginning with the remainder (any data bits that did not fit in 32-bit output). This is clarified by the example in Table 4.2. Each row corresponds to a *wr_en* signal being received. Note that this example is simplified to a 16-bit system, but the actual system is 32-bit.

Table 4.2: Example of the operation of concatenate and accumulate unit.

d_d	accumulation	output	remainder
011 1010	**011 1010**		
01 1011	**0 1101 1**011 1010		
1010 1101	**1 0101 101**0 1101 1011 1010	1010 1101 1011 1010	1 0101
011	**011**1 0101		

The CAC unit also has a flush input. This is set once the end of the frame occurs, indicating that no more data is expected. In this case the CAC module outputs its current accumulation stuffed with 1's. The concatenation that occurs is similar to that of the AC and DC encoders of the JPEG encoder (see Section 3.4.1). A look-up-table consisting of all possible combinations of widths of the accumulation and the input data is used to select which bits are selected from which source.

The input to the buffer is 32 bits, whereas the output is 8 bits. This means that read and write addresses do not directly correspond. To subtract one address from the other, the write address is bit-shifted by 2. Not

indicated in the block diagram is that the two LSBs (least significant bits) of the write address, after bit-shifting, are supplied by the CAC. They will always be 0's except on the final write of the frame in which case they indicate how many of the four bytes written to the buffer are used.

To buffer the entire JPEG image would require a substantial amount of RAM. Instead only two ISI rows of data are buffered. The ready output of *output/data* is set high whenever the write address of the buffer is 2048 (one row) greater than the read address (or the end of the frame has been reached). It does not matter if the write address has overflowed and the read address has not, because the subtraction will overflow.

The state machine description of Table 4.1 contains a simplification: the *s_data* state is in fact two states. The first state sets *l_vld* high and transmits one row of data (2048 elements), the other state sets *l_vld* low and does not transmit any data. The microprocessor does not require blanking periods between rows to be even, so data is sent as it is available saving on the amount of RAM required. There are localised periods during which data arrives in *output/data* faster than it is sent out. The primary concern of this is that it will lead to the buffer filling up. In testing of the final system the number of data elements buffered was measured. These were typically about one ISI row, and had a maximum of 37 elements more than one row. It is highly unlikely that a state will be reached in which two rows of data need storage.

Once the end of the frame has been reached the ready signal is maintained high so that output/control will continue to request data. Once the read address reaches the write address (the difference is zero), the end-of-frame (*eof*) signal is set so the state machine of *output/control* transitions to the *s_fill* state.

The JPEG specification requires that any 0xFF byte in the JPEG data is followed by a stuffed 0x00 byte. To this end *output/control* monitors the output of *output/data* and on an 0xFF byte temporarily sets *rd_en* to zero while it outputs a 0x00 byte.

4.4 Power Reduction

CMOS-based devices, such as FPGAs, primarily dissipate power during the switching of gates. The dynamic power consumption of the device during a given period is proportional to the number of gates being switched during that period. When digital systems are primarily synchronous their dynamic power consumption is effectively eliminated when the clocking source stops.

In Section 4.1.2 a clock enable system was implemented for the purpose of removing gaps in the incoming data stream. It is proposed that this system is extended to ensure that systems in the image processing pipeline, such as the JPEG encoder, are not clocked when not in use.

In order to do this, a system must be devised to recognise when a module is in use. A given module, M, has a fixed data-path latency, m_L, measured in clock cycles. Therefore each module can be considered to be in use from the time data first arrives at its inputs until M_L clock cycles after data stops arriving at its inputs. To track this progress a signal, called data valid (dv), is created as an input and output of every module along the image processing path. The dv signal at the input of a module, M, is denoted as dv_{Mi} and at the output as dv_{Mo}.

Supposing the image processing path consisted of modules $A \rightarrow B \rightarrow C$. The dv signal entering module A is referred to as dv_{Ai}, and the dv signal exiting the module is dv_{Ao}. The dv_{Ao} will respond to dv_{Ai} by mimicking it with a delay of A_L. For the purposes of this example $A_L = 3$, $B_L = 1$ and $C_L = 2$. A timing diagram depicting the responses of the relevant dv signals of this example is shown in Figure 4.8.

Each module begins processing data as soon as dv_{Mi} is high, and stops processing data once dv_{Mo} is low. It is during this period of module operation that the clock is required, the rest of the time it is not. A clock enable for the module is expressed as (dv_{Mi} OR dv_{Mo}). Figure 4.9 shows how this functions.

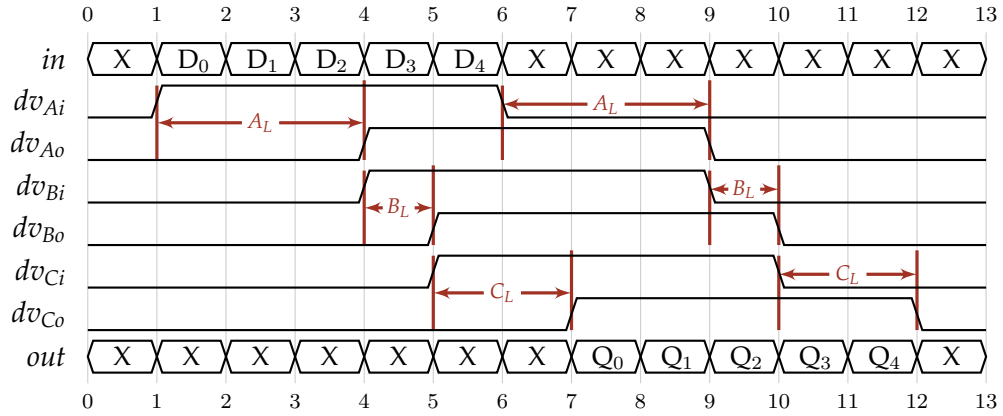


Figure 4.8: Timing diagram of data valid responses through an example image processing path.

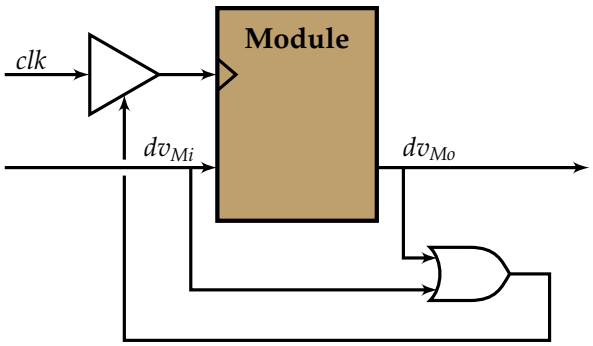


Figure 4.9: The data valid input and output lines of a module are used as clock enables for that module.

There are insufficient hardware clock enable systems in the FPGA for every module, so the alternative is to have a clocked enable line lead into an AND gate with the clock. This adds a latency of one clock cycle to the enable, therefore the data valid lines should lead the data by one clock cycle to compensate. This system is implemented upon the JPEG encoder as a single module as a proof of concept.

The mechanism by which the *dv* signal is delayed as it passes through a module is by a component module called *dv_counter*. This module is encoded in VHDL using *generics* so that different instantiations of the mod-

ule has a different delay property. This way each module in the image processing pipeline need only contain a *dv_counter* module with the delay set to equal that module's latency. The operation of each *dv_counter* is, as the name suggests, by a counter that simulates the latency of the module.

4.5 Image Processing Operations

The system thus far forms a pipeline of three blocks: input, JPEG encoder and output. It is intended that eventually several image processing operations be included in the FPGA system. These must be performed before the image is encoded, leading to the overall system structure shown in Figure 4.10.

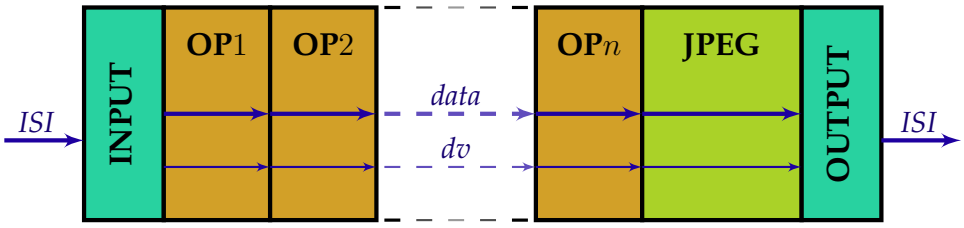


Figure 4.10: FPGA system flow showing the integration of image processing operations (*OP*).

The pipeline is highly modular: image processing operation (*OP*) modules can be added between the input module and JPEG encoder module. The number of modules in this pipeline are only limited by the available resources of the FPGA. Each module receives the pixel data from the previous module along with a *dv* signal. Each module utilises the *dv* signal to switch off when not in use and put itself in a reset state in preparation for the next frame.

The advantage of such a system is that no module depends on the presence of any other module: a module can be removed from the pipeline, *data* and *dv* are connected directly from the output of the previous module to the input of the subsequent one, and no part of the system would notice

any difference in behaviour. The exception to this is if the module changes the dimensions of the image. However, this can be anticipated by setting *width* and *height* parameters in subsequent modules. Section 4.2 discusses the use of an I^2C connection from the microprocessor to control settings on the FPGA. Such settings might be used to turn different *OP* modules on and off, or to select different parameters for their operation.

There are several image processing operations which are performed by the microprocessor in the existing system, or are planned for future implementation. Many of these are commercially sensitive (and are not part of this project), but one that is not sensitive is a simple image sharpening algorithm, which is implemented in this project as a proof of concept of image processing pipeline.

4.5.1 Image Sharpening Operation

An image sharpening filter, also known as an edge enhancement filter, operates as a high pass filter on local regions of an image: it accentuates the difference between a pixel and those surrounding it. The operation is performed by a linear transformation on a 3×3 window.

$$\begin{aligned} R_{m,n} =& k_{-1,-1}P_{m-1,n-1} + k_{0,-1}P_{m,n-1} + k_{+1,-1}P_{m+1,n-1} \\ &+ k_{-1,0}P_{m-1,n} + k_{0,0}P_{m,n} + k_{+1,0}P_{m+1,n} \\ &+ k_{-1,+1}P_{m-1,n+1} + k_{0,+1}P_{m,n+1} + k_{+1,+1}P_{m+1,n+1} \end{aligned} \tag{4.1}$$

Equation 4.1 gives the equation associated with such a linear transformation, where $P_{m,n}$ is the input pixel, and $R_{m,n}$ is the corresponding output pixel. The other pixels are the ones surrounding the pixel of interest (either the same row/column, one left/above, or right/below). The coefficients, labelled k, determine the nature of the transform. Another notation for the operation is displayed in Figure 4.11.

It is common for the coefficients to be normalised. The consequence of not normalising them is to apply a linear increase or decrease in brightness

$k_{-1,-1}$	$k_{0,-1}$	$k_{+1,-1}$
$k_{-1,0}$	$k_{0,0}$	$k_{+1,0}$
$k_{-1,+1}$	$k_{0,+1}$	$k_{+1,+1}$

Figure 4.11: A generic 3×3 linear window operation corresponding to Equation 4.1.

to the whole image, and thereby decrease the overall contrast. Normalisation is simply the requirement that the coefficients sum to 1.

Image sharpening processes usually place large positive coefficients on the original pixel of interest, and negative coefficients on the surrounding pixels. The complementary blurring operation is a Laplacian filter which has a large negative coefficient on the pixel of interest, and positive coefficients on the surrounding pixels [6]. In this instance the coefficients are taken from the image sharpening filter of the microprocessor in the existing system, and are shown in Figure 4.12.

0	-1	0
-1	5	-1
0	-1	0

Figure 4.12: Coefficients of the image sharpening operation.

Note that the coefficients do add to 1, and that the corner pixels are not used at all.

The implementation of window operations requires the buffering of rows, as described in Section 2.3.2. A 3×3 operation requires that two rows be buffered as well as the window itself. Each row is stored as a FIFO buffer: as a new element enters, the element that has been stored the longest exits. Instead of multiplying by 5, which requires a hardware multiplier, a bit-shift and addition occur: $5P_{m,n} = P_{m,n} \ll 2 + P_{m,n}$. Bit-shifting operations are effectively 'free' in hardware because they are only a reroute of the signals corresponding to the bits.

Additional logic is required for edge cases; where the pixel of interest is on the edge of the frame then the window goes over the edge. Any missing input pixel of a window operation is commonly filled with a constant, the adjacent pixel within the frame, or the pixel opposite it in the window [45]. Using a constant for this filter leads to unpredictable results because the constant will not correspond to the rest of the pixels within the window. The adjacent edge pixel in this case will always be the pixel of interest, and using it as a contrasting neighbour pixel counteracts the effect of sharpening. The opposite pixel within the window is the suitable solution. Logic is added to the operation so that when the window is in the top row of the frame then the top pixel of the window is a duplicate of the bottom pixel of the window.

Although two rows are buffered, output occurs only one row after the corresponding pixel has entered the operator because the pixel of interest is in the centre of the window. This means that the total latency of the module is 756 clock cycles.

(a) Original **(b)** Sharpened

Figure 4.13: Comparison of image before and after the application of the image sharpening filter.

Figure 4.13 demonstrates the effect of this image sharpening filter. While edges and details become more defined so does image noise. A trade-off is made in the design of image sharpening filters between the improvement to edges and details and the exacerbation of noise; in this case the design decision was previously made by other Kinopta engineers.

4.6 System Summary

The existing Blackeye II system comprises a CMOS image sensor and a microprocessor. The microprocessor sends control signals to the image sensor, which in turn sends 752×480 greyscale bitmap images back via the ISI. The microprocessor performs image processing operations upon the image then encodes it in a JPEG based file and stores it on an SD card. The microprocessor also operates a web server, which provides retrieval of the images from the SD-Card to an external PC. Although the image sensor can produce 60 fps, the microprocessor cannot process more than 2 fps under full load, severely limiting the capabilities of the system.

This project proposes to supplement the microprocessor with an FPGA. The task of the FPGA will be to receive the data directly from the image sensor's ISI, perform image processing operations upon the images as they arrive, encode them as JPEG files and send them to the microprocessor for storage. The microprocessor maintains overall control of the system, so infrastructure is put in place for the FPGA to receive control signals.

The FPGA system forms a pipeline through which the data from the image sensor passes to the microprocessor. This pipeline is divided into four distinct stages:

Input

Receives data from the image sensor via ISI. It forwards the data on to the rest of the system, stopping the clock to the image processing and JPEG encoding modules during row blanks in the incoming data.

Image Processing Operations

These independent modules receive a stream of pixel data and outputs pixel data in the same form. They modify the pixel values to achieve some visual effect. In addition to the *data* line there is also a *dv* line which is used to turn modules off when not in use. Modules can be disabled by multiplexing the data line to bypass the module.

JPEG Encoder

Receives pixel data and encodes it into a JPEG data stream. It then outputs this data stream, along with a low resolution bitmap of the image to the output module.

Output

Upon receiving data from the JPEG encoder the output module begins to send a frame to the microprocessor via ISI. It sends out a complete JPEG file by prepending the JPEG metadata to the data from the JPEG encoder. This is followed by empty data packets, and finishes with a low resolution bitmap.

This system is implemented in VHDL on an Altera Cyclone FPGA development board attached via a breakout board to the microprocessor and image sensor boards of the Blackeye II camera.

Chapter 5

Evaluation and Conclusion

This chapter begins by describing the implementation and testing of the system in hardware. Section 5.2 goes on to evaluate the JPEG encoder by comparing it with commercial designs. The system is assessed in Section 5.3 against the project's objectives. This is followed by an outline of how Kinopta intends to build on this project for their product. Section 5.5 provides a summary of the project and its success. The chapter ends with a statement from Kinopta's CEO.

5.1 The System in Operation

The entire system is tested in simulation because bugs are easier to track and identify in simulation than in hardware. A VHDL testbench is written that provides stimuli to the system imitating the image sensor. Actual bitmaps previously taken from the Blackeye II's image sensor are used as test data. The testbench saves the output of the system to files, which are inspected to determine if the data is as expected. The output files should each consist of a JPEG JFIF file followed by a run of bytes of value 0xFF and ending in a 5640 byte bitmap. In each case the JPEG file was manually extracted from the output data and saved separately. To test the JPEG file it is opened with a program called JPEGsnoop. This software is designed

for the debugging of JPEG encoders and is useful for identifying where errors occur in the encoding.

Once the system performs without error in simulation, the design is loaded onto the FPGA development board, which is connected to the image sensor board and microprocessor board. To test the camera, it is plugged into a PC via a USB cable, and a web browser on the PC is used to view the images from the camera in real time.

Although simulations are able to accurately test the performance of an FPGA, the stimuli provided by the testbench can't always cover every situation that may arise in deployment. Therefore it is not entirely surprising that further bugs were encountered in the hardware implementation. To facilitate debugging of systems in hardware, Altera developed the *SignalTap II* system. SignalTap II provides the means to analyse internal signals on the FPGA. It does so by supplementing the FPGA design with additional logic during the synthesis of the design. SignalTap II systems are limited in their complexity and amount of data they may acquire by the availability of resources.

A SignalTap II system is built using Quartus II, and is implemented by selecting the signals that are of interest and defining trigger conditions for the system to report data back to the PC. When the trigger conditions are met, the waveforms of the signals of interest are sent to the PC. They extend a predefined number of clock cycles before and after the trigger event. By selecting appropriate signals and triggers, bugs are traced to their source and solved. Figure 5.1 shows the Blackeye II with FPGA system operating successfully.

During testing the system operated without issue, and achieved a frame rate of 20 fps. The usage of FPGA resources is shown in Table 5.1.

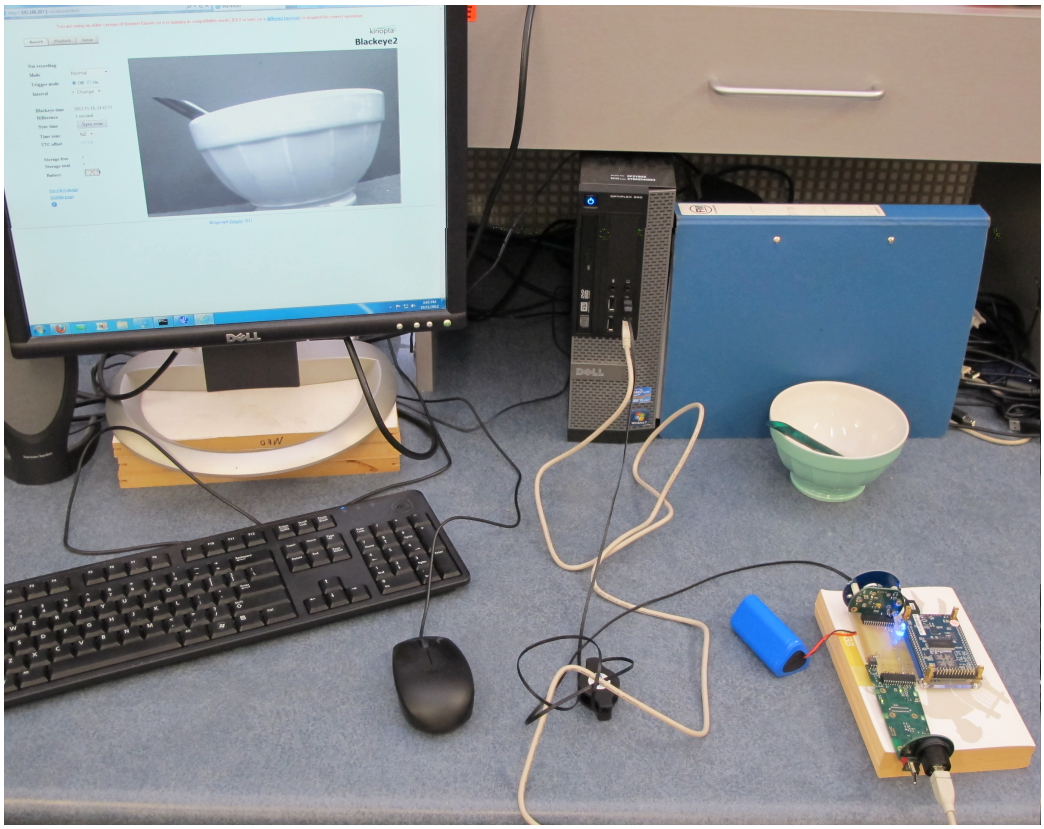


Figure 5.1: Camera system in operation with the FPGA-based image processing pipeline. The web browser on the PC is displaying images from camera in real time.

5.2 Comparison of JPEG Encoder

The JPEG encoder presented in this thesis is referred to as the *project encoder* for this section to prevent ambiguity.

There are several commercial JPEG encoder IP blocks (see Section 2.4), which are compared with the project encoder to evaluate it. Only the data that is publicly available for these encoders can be retrieved, so there are gaps that are noted by blanks in tables. Table 5.2 presents a summary of these encoders.

Most of the encoders are labelled as baseline JPEG standard compliant.

Table 5.1: Usage of FPGA resource by module.

Entity	LEs	RAM Blocks	RAM (Kbits)	Multipliers
Image Sharpener	226	2	12	0
JPEG Encoder	1 960	25	182	9
2D-DCT	618	23	180	8
Quantisation	46	0	0	1
Zig-Zag Buffer	76	1	1.4	0
Huffman Encoder	1 177	1	0.7	0
Output Block	1 054	13	102	0
Total	3 286	39	297	9
FPGA	22 320	66	594	66

The CAST encoder claims to support all JPEG formats, so it is expected to require more resources than encoders that do not. All the encoders provide JFIF header data. All except the Sundance encoder describe the colour formats that they offer. They all have flexible frame sizes, for example VISENGI and CAST encoders specify a maximum frame size of 64k×64k pixels (as per the maximum in the JPEG standard). Three of them specifically state that the user may configure the quantisation and Huffman tables. They all have greater capabilities than the project encoder, but this is necessary for their deployment in unknown and varied applications. There will be trade-offs in their design, which make them less optimal for most applications than a purpose-built encoder does.

The encoders are advertised as providing fast encoding speeds; for example, the Sundance encoder claims to be able to encode up to 660 megapixels per second in a Virtex II FPGA. Speed comparisons are very difficult, as they depend on the device the system is implemented in. No clock speed performance measurements were made of the project encoder, because the data rate it is capable of is far in excess of the rate it receives data. The architecture of the system restricts the project encoder to 20 fps,

Table 5.2: Comparison of commercial JPEG encoder IP blocks [31, 32, 33, 34, 35].

Company	Product	Tables	Input	Resources
Entner Electronics	JPEG CODEC Encoder			*Cyclone IV* 2250 LEs 5 RAM Blocks 8 Multipliers
VISENGI	JPEG Encoder			*Cyclone III* 8125 LEs 4 RAM Blocks 3 Multipliers
CAST, Inc	JPEG-C	Configurable		*Cyclone III* 10154 LEs 9 RAM Blocks 36 Multipliers
Sundance Multi-processor Technology	FC-JPEG04	Configurable	10 pixels	*Virtex II* 10750 Slices 45 RAM Blocks 34 Multipliers
Barco Silex	BA116	Configurable	64 pixels	

which is equivalent to 7 megapixels per second.

Next to four of the encoders in Table 5.2 are resource usage of an FPGA. This resource usage will be dependent upon the device itself and the software that generates the FPGA design. However, resource usage of FPGAs in the same family of devices is comparable, which is why the device family is mentioned under the resource usage. Note that the Sundance encoder is only quoted on a Xilinx Virtex device which makes it difficult to compare with Altera devices. The architecture of the logic elements and the architecture multipliers and the size of the RAM blocks does not vary between the Cyclone III and Cyclone IV devices, so they make good com-

parisons. A summary of resource usage of the encoders is presented in Table 5.3.

Table 5.3: Comparison of resource usage of the JPEG encoder from this thesis with commercial encoders. The number of resources available in the FPGA used for testing (Cyclone IV EP4CE22), is included for comparison.

Encoder	LEs	RAM Blocks	Multipliers
Project	1 960	25	9
Entner	2 250	5	8
VISENGI	8 125	4	3
CAST	10 154	9	36
FPGA	22 320	66	66

An initial appraisal of the data would suggest that the project encoder uses the fewest logic elements, a moderate amount of multipliers and by far the most RAM blocks. The importance of these figures is enhanced by discussing them in terms of their availability, which can be achieved by expressing them as a percentage of the available resources on an FPGA: the project encoder uses 9% of LEs, 38% of RAM blocks and 14% of multipliers. If the proportions of the different resources used are unbalanced, waste occurs; the device will have excess resources of types in less demand by the system. Excess resources cost money and power consumption, particularly static power consumption. This assumes that FPGAs will typically have a comparable ratio of different resources available.

The most obvious issue with the project encoder is that it utilises so much of the block RAM on the device. Referring back to Table 5.1 the main use of the block RAM is the 2D-DCT module which uses 23 of the 25 blocks. In fact all 23 of these blocks are used by the transpose buffer between the two parts of the DCT. The need for such a buffer is due to the difference between the ordering of the data from the image sensor (pixels are output row by row), and the ordering required by the JPEG standard (pixels in 8×8 square blocks). Considering that the size of this buffer de-

pends on the frame, and that the commercial encoders are designed to deal with frames larger than the ones in this application, the encoders would require at least as many RAM blocks in order to do the necessary reordering. This suggests that the commercial encoders do not include such a buffer, and instead expect the data to be in a more JPEG-friendly order to begin with. This conclusion is bolstered by the Barco Silex encoder stating it takes an 8×8 block of pixels as input. Using image sensors like the one in this project, the pixels would need to be reordered for use with these commercial encoders, adding 23 RAM blocks to their utilisation, making them in fact more RAM intensive than the project encoder design.

Steps were taken to reduce the use of multipliers, such as bit-shifting followed by addition, in the design of the project encoder. However, in Section 3.1.1, a 1D-DCT design was selected that used more multipliers than the alternatives considered, in exchange for less logic, and therefore LEs, as well as a simpler development process. It appears that this was a good design decision as the utilisation of multipliers did not prove excessive: only 14% of those available. While the VISENGI design uses a third of the number, the project encoder is on a par with the Entner Electronics design and uses significantly less than the CAST design.

Thus far it appears that the project encoder compares very favourably in terms of resource usage with the commercial alternatives. This is to be expected of a system focused on a single application when compared with general purpose systems. However, the comparisons are not of like systems; each of these (although it is not clear in the case of the Entner Electronics system), include the generation of the JFIF file header. The data of these systems is output as an 8-bit bus in an unbroken stream for each file. In the system presented here, these features are included as part of the *output* module and not in the JPEG encoder itself. The *output* module adds a further 1054 LEs and 13 RAM blocks to total resource usage, but part of this is used for other purposes, such as the control of the ISI, and the low resolution bitmap storage. If the resource of *output* are added to

those of the project encoder the multiplier usage remains the same, the LEs increase to 3014 and RAM blocks to 38. In this case the LE utilisation is still favourable, being only 34% greater than the best competitor. However, the RAM usage is significant, so should be a primary focus of future revisions of the system.

5.3 Evaluation of System

The aim of this project was to design a hardware-based system to improve the performance and capabilities of the Blackeye II camera system. The existing Blackeye II system utilises a microprocessor for all its processing, and this is proving insufficient for some applications of the camera. Features that Kinopta would like to add to the camera include a faster frame rate, more real-time image processing operations and improved power management of the system.

An FPGA-based hardware accelerator system was designed for the Blackeye II. This hardware system is intended as a proof of concept; it implements key aspects of the final system to demonstrate its potential, so it was designed with further development in mind. Because the design of a proof of concept is only a limited commitment to a system, this system is designed to be tested with the existing camera hardware with no modification; only a simple breakout board was required.

The previous camera system could only achieve a frame rate of 2 fps with a 752×480 greyscale frame. Applications such as wildlife monitoring and traffic flow surveying require the camera system to achieve a frame rate of 12.5 fps. Each frame requires certain image processing operations, and the frame is encoded in the JPEG format for compression. Kinopta would like to add more image processing operations to the camera, but this would lead to a further reduction in frame rate. This project implemented an image processing pipeline, including a JPEG encoder, on an FPGA so that the system may achieve a higher throughput. Because FPGA

systems are pipelined, the addition of further image processing stages do not have an adverse affect on the throughput of the system, though they do add slightly to the latency of frames. Frame latency is not a primary concern of the design because the microprocessor will store the images for later use in most cases.

The FPGA system in the form presented in this thesis is able to send encoded JPEG images to the microprocessor of the Blackeye II at a rate of 20 fps. This significantly exceeds the requirements of the system (12.5 fps), and there is scope with further revisions to increase this frame rate even more if future applications require it. As current applications do not require such a high frame rate the system clock speed may be reduced, resulting in power savings.

The system receives data from the image sensor, applies image processing operations as required and outputs it as a JPEG image. Currently the only image processing operation that has been implemented is an image sharpening filter, but the modularity of the design makes it simple to add modules. Furthermore, modules can be removed and added back in to the processing pipeline on-the-fly by way of a bypass multiplexer. The modularity extends to the power control system; each module is only clocked when in use. This reduces the power consumption of the FPGA system, which is of great significance in the camera's applications.

Data is transferred from the FPGA to the microprocessor via ISI. The operating system of the microprocessor is not real-time, so without dedicated hardware the FPGA would need to wait for the microprocessor to poll the data. Instead, by using the microprocessor's hardware ISI, the FPGA can push the data directly to system memory, where it is stored as a block of memory for the operating system to access when it is ready. The ISI specifies that data is transferred in *frames* of a predefined size. The size of an image after JPEG encoding cannot be accurately predicted, so this size was set to one which is expected to be sufficient for any JPEG image. In addition to the JPEG file, the ISI can also be used to transfer other infor-

mation the microprocessor might need. For example, the microprocessor uses lighting conditions to make changes to the image sensor settings and uses a very low resolution image as input to the algorithm. Therefore a low resolution bitmap (1 pixel per 8×8 block of original pixels) is sent in conjunction with each JPEG frame. The low resolution image is obtained by the FPGA with very little extra overhead by using the processes of the JPEG encoder to extract an average of each 8×8 pixel block.

By adding the FPGA system to the Blackeye II it can be used in new applications where previously the frame rate had been insufficient. And the camera has the potential to be developed further for other planned applications. This project succeeded in the goals it set out to achieve: the development of a suitable architecture, including a JPEG encoder, and successful testing with the Blackeye II hardware. It exceeded its goals by implementing a power management system that pauses modules whenever they're not in use. The camera originally could only achieve 2 fps, but this project achieves a frame rate of 20 fps with the Blackeye II hardware.

5.4 Future Work

This project developed a proof of concept system for the improved performance of the Blackeye II camera. This system is intended to be the basis of further development and this was borne in mind during the design process. These future developments can be divided into three categories: further refinements and improvements to the system architecture ; new image processing operations can be added to the pipeline; and a degree of on-the-fly configurability is added to the current components.

5.4.1 Improvement

In remote surveillance cameras power consumption needs to be minimised as much as possible. Any chip added to the camera will increase the power

consumption, but the added power consumption of the FPGA will be mitigated by a decrease in the demand on the microprocessor. Nevertheless, any way in which the power consumption may be reduced should be considered.

One way to reduce the power consumption of the FPGA system is to use a smaller FPGA, that is, one which has fewer resources. As was noted in Section 5.2 the JPEG encoding system uses a disproportionately high amount of RAM compared with other FPGA resources. Therefore it should be a goal in revising the system to reduce the RAM requirements where possible. One immediate step would be to change the ISI frame size between the FPGA and microprocessor. The required size of this frame is uncertain due to the variation in file size produced by the JPEG algorithm. Further investigation is required to establish how much this frame may be reduced by. In any case, the width of the ISI frame may be decreased by increasing the height without affecting the total capacity of the frame. The size of the buffer in the *output/data* module is proportional to the width of the ISI frame, so a reduction in the frame width will result in a reduction in the required RAM. There will be a limit to this reduction, and this needs to be investigated.

Future versions of the Blackeye camera system may use a different microprocessor. Dedicated ISI peripherals are not common features on microprocessors, so unless an alternative transmission scheme is developed, the choice of microprocessor is greatly restricted. To remove this restriction, alternate transmission mechanisms should be investigated. One potential solution is to use a RAM chip to buffer data between FPGA and microprocessor.

Some applications may require colour images. The JFIF file format requires colour data to be in a YCbCr format (in practice very similar to YUV): one luminance component, Y, and two chrominance components, Cr and Cb. This format is also particularly useful in image processing operations, and is commonly used by colour image sensors. The other com-

mon colour format is RGB, but there is a transform for converting RGB data to YCbCr data. The JPEG standard treats each of these components similarly and separately. This means that three instantiations of the JPEG encoder module are used and the data is processed in parallel. The only difference between the encoder modules are the quantisation and Huffman tables. This makes the implementation of a colour encoder a comparatively simple process. The biggest change that would be required of the system would be in the *output/data* module, which would have to order the data from the three encoders.

5.4.2 Image Processing

The architecture of the FPGA system allows for the insertion of image processing operations. One of the strengths of the Blackeye II is its low-light performance, this can be further enhanced by contrast and brightness adjustment techniques. Traffic monitoring requires the that number plates on cars are clearly readable. These applications and others have the potential for improvement by such operations as pixel binning, stuck pixel repair, thresholding and morphological filters.

Pixel binning involves the grouping of pixels, for example in 2×2 blocks. The operation takes each block as an input, sums the intensities of the constituent pixels, and outputs this number as a single pixel representing the whole block. The effect is to increase intensity resolution and reduce high spatial frequency noise, but at the cost of reducing the spatial resolution. High spatial frequency noise is a problem endemic to CMOS image sensors [42], and this provides a solution where the trade off is acceptable. However, the most important benefit for the Blackeye II camera is in night vision operation. The increase in intensity resolution enables details in the dark, that would otherwise be obscure, to be seen.

Stuck pixels refer to particular pixels of an image sensor that no longer respond to light, and instead always output the same value. This is not

an uncommon occurrence, but once identified, a filter can be designed to approximate a stuck pixel by taking the mean of the surrounding pixels.

Thresholding refers to a type of filter that performs a transformation on each individual pixel, usually producing a binary output. A common example would be to output a 1 when the pixel value is above a preset threshold, otherwise output 0. Thresholding is often used for selecting regions of interest for other processes.

In their simplest form, morphological filters work on binary data, so pair well with thresholding, but the principles can be extended to greyscale images. A morphological filter affects the shape of the image and is particularly useful for smoothing and separating object boundaries.

5.4.3 Configurability

The breakout board in the test setup included an I^2C communications link from the microprocessor to the FPGA. The FPGA does not currently utilise this connection, but it is intended for use as a mechanism for the microprocessor to set control parameters on the FPGA. An I^2C interface would provide a simple control scheme to allow the microprocessor to set and read a group of registers on the FPGA. The content of these registers will be used by the FPGA to set control signals used by different modules.

One such control signal, which has been previously mentioned, is an image process bypass signal. This signal would be used to multiplex the outputs of an image processing module, to use either the module's functionality, or to bypass it by directly outputting its inputs.

The system currently implements the JPEG encoder using an approximation of the IJG 70% quality quantisation table. Additional tables could be added for other IJG quality settings and the appropriate table selected by the configuration register. This also involves a change to part of the JFIF file header because this includes the table.

There is a possibility that different frame sizes may be used in the fu-

ture. In such a case the only changes required are to adjust parameters for counters (address counters, pixel counters etc.) and the size of buffers. Parameters can be changed globally through the use of VHDL *generics*, which were used in the implementation for this reason. If an application requires a frame size other than 752×480 pixels then the parameters and buffer sizes can be changed during the generation of the configuration file. If the application requires the frame size to change during operation, for example if pixel binning is turned on or off, then the buffer must be generated to cope with the maximum size, and the counter parameters will read from an active signal. Changes in frame size do not affect the frame rate, unlike software, which has an inverse relationship between frame size and potential frame rate. Therefore the hardware system has an advantage over software in scalability.

5.5 Conclusion

Kinopta developed the Blackeye II camera system for use in wildlife observation, rural security, traffic monitoring and other applications. These types of application require constant operation in anticipation of events that can occur at any time, so frames must be taken at regular intervals. The cameras are often in remote locations so must operate without human intervention for extended periods, which means that power consumption is of great importance to the design. They must also store all the images that they take, so the images must be compressed. For compression, a JPEG encoder is used. The cameras are designed to not require any specialised software to retrieve images – only a web browser on a modern operating system. Therefore any processing that needs to be performed on the images before the viewer sees them must be done in the camera as the images are retrieved and encoded. Kinopta wants to improve the performance of the Blackeye II in its current applications, and expand into new applications. They have reached a point where the cur-

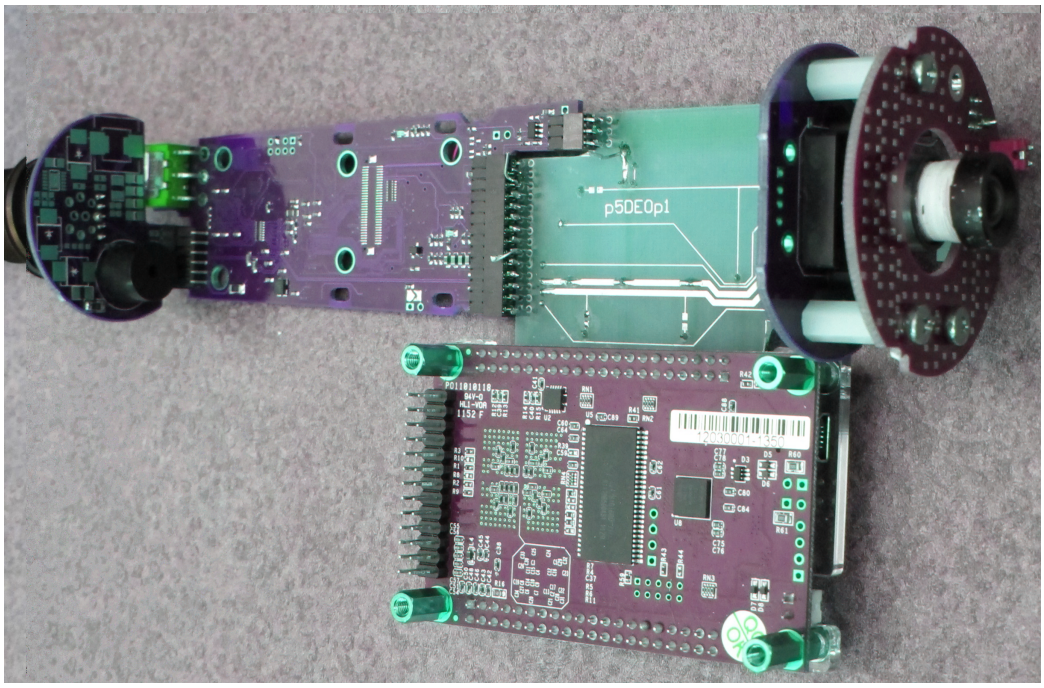


Figure 5.2: Blackeye II camera with breakout board connecting the FPGA development board to the image sensor board and microprocessor board.

rent microprocessor-based architecture is unable to provide the improvements they now seek. This project provides a solution to this problem in the form of an FPGA-based hardware accelerator.

The existing microprocessor system is unable to achieve a frame rate greater than 2 fps. The addition of planned image processing operations will cause it to slow down even more. Planned applications of the Blackeye II require a frame rate of 12.5 fps. A solution is required that enables the camera to operate at higher frame rates while performing additional image processing operations. Furthermore, the solution must be designed with power consumption in mind.

In this project a solution was proposed, and from this a proof of concept system was designed, built and tested. The solution is an FPGA-based hardware accelerator. For operations that can be performed in parallel,

hardware solutions provide improved performance for low energy costs compared with software. Such a system is an image processing pipeline because it is highly parallel. FPGAs are an ideal platform for the design of digital hardware solutions, particularly in low volume production, such as the Blackeye II, due to the low NRE costs compared with other ASICs. JPEG encoders for FPGAs are available for licensing, but a custom solution is preferred by Kinopta. This project implements a JPEG encoder which improves upon existing designs. The project then creates an image processing pipeline that receives data from the camera's image sensor, performs image processing operations upon it, encodes it as a JPEG file before transferring it to the microprocessor of the camera.

The JPEG standard is based around blocks of 8×8 pixels, referred to as MCUs. Each MCU is transformed into a frequency domain representation by way of a 2D-DCT. The frequency precision is reduced by way of a quantisation operation. This data is reordered by a zig-zag buffer for encoding. The final stage is a Huffman encoder, which is the stage where data compression occurs. These four sections form distinct modules in the FPGA design. The 2D-DCT is the most process-intensive operation, and therefore is the focus of much research. The DCT algorithm in this project was based upon the work of Woods et al. [1]. It was found that the coefficients used in this approximation of the algorithm did not provide accurate results, and new coefficients were found. The trade-off between precision and accuracy in operations in the DCT and quantisation modules was explored. By identifying this relationship the design minimised the resources it required for the desired accuracy. The JPEG encoder was designed in VHDL, and was found to work successfully in simulation.

The FPGA system receives data from the camera's image sensor via ISI. This provides a greyscale bitmap of the image, which is forwarded on to the image processing pipeline. The ISI includes blanking periods in which no data is sent, so a mechanism was developed to pause the image processing pipeline until data transmission resumes. Data is tracked by a

data valid signal as it passes through the image processing pipeline so that modules may be identified as in use or not. A clock gating mechanism is used to pause modules whenever they are not use in order to reduce the power consumption of the design.

The architecture of the pipeline is designed to allow for convenient sequential insertion of image processing modules. For this project an example image processing module was implemented. This module performs an image sharpening operation, and operates successfully within the system architecture. The image processing pipeline concludes with the JPEG encoder. To transfer the data to the microprocessor, the microprocessor's dedicated ISI peripheral is used. A dedicated output module was developed as an interface between the JPEG encoder modules and the ISI. A state machine controls the output module and makes sure that the output data conforms to the microcontroller's ISI specifications. A module, *output/data*, concatenates and buffers data from the JPEG encoder in preparation for the output module to transmit it in bursts. With the completion of the output module the system was able to be tested in hardware.

An FPGA development board was used for testing the system. A breakout board was used to interface the FPGA development board with the boards of the Blackeye II camera. The camera's webserver was used to observe the images as they arrived from the FPGA system to the microprocessor. The system was found to operate successfully at frame rates of up to 20 fps.

The JPEG encoder compares favourably against the commercially available encoders, and provides the flexibility and performance that the project required. The proof of concept system required minimal changes to the existing camera hardware in its implementation. It provided a tenfold increase in frame rate, which exceeds the project's requirements. The system also demonstrates how image processing operations can be added without affecting the frame rate. The power consumption of the FPGA system is minimised by the use of control mechanisms to turn off modules when

not in use, and by reducing the resource usage through careful analytic design.

This project produced a proof of concept of a hardware accelerator for the Blackeye II camera. The architecture this system introduces allows a small company to increase the performance of its product. The camera will provide a more competitive solution in its existing applications, and be able to offer solutions in new applications. Kinopta intends to use the work done in this project as the basis for upcoming revisions to its cameras, and is currently engaging in a hardware redesign to incorporate it. The features of the system developed in this project enable this redesign to be achieved with minimal effort. They have hired me to continue developing the FPGA system for them.

5.6 Statement from Kinopta

The following is a statement about this project from the founder of Kinopta, Don Peat:

> Alexander Kane has engaged in a project for Kinopta Ltd, to implement a JPEG algorithm on an FPGA, suitable for inclusion into our Blackeye II camera system.
>
> Kinopta is a company that had its origins in a different technology business (satellite communications), but moved into remote imaging as a strategic business decision. The company developed two initial cameras systems for remote resource management and protection, such as coastal fisheries. First Farsight, then Blackeye1, using OEM camera modules, and a simple controller to store images on an SD card. It soon became clear that there were other markets in this space, and that a much more sophisticated and more automatic camera platform was needed, to give both the image quality and a simple user experience.
>
> The company is still on this trajectory – the Blackeye II camera system went a good distance to reaching these market and strategic requirements. But always, there are the drivers for faster imaging, lower power and longer endurance, better photos, and very importantly, better tools to find the photos of interest in very large archives. On that last point, Kinopta has a proprietary and 'patent applied for' method of indexing and rapidly locating individual images in massive photo archives, which can easily contain millions of individual images.

Although we have a relatively fast processor for an embedded system, a 400MHz ARM9 computer, the JPEG algorithm in particular, takes a good amount of time, and limits the sustained photo rate to about 2 frames per second (fps) for our standard WVGA image sensor. The JPEG algorithm for a 5 megapixel sensor we have been testing takes seconds. The company has long recognised that an FPGA could do this and other image processing tasks at the native frame rate of the sensor, allowing faster performance, and/or lower power by virtue of the fact that the system can potentially sleep between image tasks. But we are a small company with limited resources at this time, so we were grateful when Alexander embarked on a project to integrate an FPGA into our Blackeye II system specifically to implement an efficient JPEG algorithm for the reasons just mentioned. It needed someone with talent and commitment to own and champion this process to see it through.

For a number of reasons we strongly believe this is an important strategic capability for the company, not just for the initial advantages of a hardware JPEG processor, but ultimately (and in conjuction with the CPU) for other ways to analyse images on-the-fly to look for certain characteristics. Number plate recognition with our traffic survey camera, is a good example of the potential of this.

Alexander has worked extraordinarily well and diligently on this project. It would have been tough for him working with a small company that is still trying to develop its market, but with a good and resourceful Kiwi spirit, he took it all in his stride.

And, at the end of it all, it worked. The hardware configuration of the proof of concept design is almost identical to the actual system that is now being used on the final printed circuit board which is now in the final stages of design. I have no doubt that this will work, the final process of a custom PCB is really just a CAD job, the hard engineering has already been done by Alexander. But we all look forward to this working in the final camera housing for real field trials.

We believe that this is just the beginning of FPGA-based applications for us. It has always been my view as CEO, that we need to develop cameras that solve real problems, and that are significantly different from the miriad of low cost off-the-shelf products that are available out of China. And we believe that packing enough technology into our cameras, in order to take the right photo at the right time is what will make our cameras different, and commercially successful. It has been a pleasure to work with Alexander. This was my first experience with hosting a postgraduate project, and we've certainly learnt a few things about this, but I will certainly be keen to do this again. I hope it has worked for Alexander – it was a tough and ambitious project, but one in which he has ultimately succeeded.

I would also like to commend the VUW engineering school, on a job well done. I had to make myself remember that Alexander had just finished an honours degree, and didn't have years of professional experience behind him, yet I always worked with him at the demanding level of an experienced professional engineer. This was a big ask for a newly graduated engineer, but in my view, he certainly stepped up

to that mark, and beyond it.

I am most appreciative,

Sincerely,
Don Peat, (BE Hons 1st class, Cant'y 1979)
CEO Kinopta Ltd.

06/12/2012

Bibliography

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

[2] S. Ahmed, G. Sassatelli, L. Torres, and L. Rougé, "Survey of new trends in industry for programmable hardware: FPGAs, MPPAs, MPSoCs, structured ASICs, eFPGAs and new wave of innovation in FPGAs," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pp. 291–297, IEEE, 2010.

[3] G. R. Stewart, "Implementing video compression algorithms on reconfigurable devices," Master's thesis, University of Glasgow, Jun. 2009.

[4] V. A. Pedroni, *Circuit Design and Simulation with VHDL*. The MIT Press, second ed., Sept. 2010.

[5] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, pp. 171–210, June 2002.

[6] D. Bailey, *Design for embedded image processing on FPGAs*. Singapore: John Wiley & Sons (Asia), 2011.

[7] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu, "An overview of reconfigurable hardware in embedded systems," *EURASIP J. Embedded Syst.*, vol. 2006, pp. 13–13, Jan. 2006.

[8] Q. Jin, D. Thomas, and W. Luk, "Automated application acceleration using software to hardware transformation," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 411–414, Dec. 2009.

[9] C. Valderrama, L. Jojczyk, and P. Possa, "Convergence in reconfigurable embedded systems," in *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, pp. 1144–1147, Dec. 2010.

[10] S. Liu, R. N. Pittman, A. Form, and J.-L. Gaudiot, "On energy efficiency of reconfigurable systems with run-time partial reconfiguration," in *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pp. 265–272, July 2010.

[11] R. Frazer, "Adding hardware accelerators to reduce power in embedded systems," White Paper 01112, Altera Corporation, San Jose, Sept. 2009.

[12] A. Amara, F. Amiel, and T. Ea, "Fpga vs. asic for low power applications," *Microelectronics Journal*, vol. 37, no. 8, pp. 669–677, 2006.

[13] Y. Zhang, J. Roivainen, and A. Mammela, "Clock-gating in FPGAs: A novel and comparative evaluation," in *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pp. 584–590, Sept. 2006.

[14] Joint Photographic Experts Group, "Information technology digital compression and coding of continuous-tone still images requirements and guidelines," tech. rep., The International Telegraph and

Telephone Consultative Committee (CCITT), Sep 1992. JPEG Standard Specification.

[15] K. Malik Mohamad and M. Deris, "Visualization of jpeg metadata," in *Visual Informatics: Bridging Research and Practice* (H. Badioze Zaman, P. Robinson, M. Petrou, P. Olivier, H. Schrder, and T. Shih, eds.), vol. 5857 of *Lecture Notes in Computer Science*, pp. 543–550, Springer Berlin Heidelberg, 2009.

[16] C. Kemerer, C. Liu, and M. Smith, "Strategies for tomorrow's "Winners-Take-Some" digital goods markets." Carnegie Mellon, 2011.

[17] J. Miano, *Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP*. Siggraph Series, Addison Wesley, 1999.

[18] J. Mannos and D. Sakrison, "The effects of a visual fidelity criterion of the encoding of images," *Information Theory, IEEE Transactions on*, vol. 20, pp. 525–536, July 1974.

[19] R. Reininger and J. Gibson, "Distributions of the two-dimensional DCT coefficients for images," *Communications, IEEE Transactions on*, vol. 31, pp. 835–839, June 1983.

[20] W. Osberger, *Perceptual vision models for picture quality assessmnet and compression applications*. PhD thesis, Queensland University of Technology, Brisbane, Mar. 1999.

[21] D. Salomon, *Data compression: the complete reference*. Springer, 2007.

[22] L. Agostini, S. Bampi, and I. Silva, "High throughput architecture of JPEG compressor for color images targeting FPGAs," in *Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on*, pp. 180–183, Dec. 2006.

[23] N. Ahmed, T. Natarajan, and K. Rao, "Discrete cosine transform," *Computers, IEEE Transactions on*, vol. C-23, pp. 90–93, Jan. 1974.

[24] H. Bauschke, C. Hamilton, M. Macklem, J. McMichael, and N. Swart, "Recompression of JPEG images by requantization," *Image Processing, IEEE Transactions on*, vol. 12, pp. 843–849, July 2003.

[25] J. Ahmad, K. Raza, M. Ebrahim, and U. Talha, "FPGA based implementation of baseline JPEG decoder," in *Proceedings of the 7th International Conference on Frontiers of Information Technology*, FIT '09, (New York, NY, USA), pp. 29:1–29:6, ACM, 2009.

[26] R. Kirsch, "SEAC and the start of image processing at the National Bureau of Standards," *Annals of the History of Computing, IEEE*, vol. 20, pp. 7–13, Apr./June 1998.

[27] Y.-L. Zhaog, D.-C. Juang, and C.-H. Horng, "A color video camera using FPGA video processor," in *ASIC Conference and Exhibit, 1991. Proceedings., Fourth Annual IEEE International*, pp. 16–17, Sept. 1991.

[28] C. T. Johnston, K. T. Gribbon, and D. G. Bailey, "Implementing image processing algorithms on FPGAs," in *Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon04, Palmerston North*, pp. 118–123, 2004.

[29] A. Downton and D. Crookes, "Parallel architectures for image processing," *Electronics Communication Engineering Journal*, vol. 10, pp. 139–151, June 1998.

[30] F. M. Waltz and J. W. V. Miller, "Efficient algorithm for Gaussian blur using finite-state machines," in *Machine Vision Systems for Inspection and Metrology VII, SPIE Conference on*, pp. 334–341, Oct. 1998.

[31] "JPEG CODEC product brief," 2011.

[32] "JPEG encoder," 2012.

[33] "JPEG-C," 2012.

[34] "FC-JPEG04," 2012.

[35] "BA116 - JPEG encoder," 2012.

[36] L. Agostini, R. Porto, S. Bampi, and I. Silva, "A FPGA based design of a multiplierless and fully pipelined JPEG compressor," in *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*, pp. 210–213, Sept. 2005.

[37] E. Kusuma and T. Widodo, "FPGA implementation of pipelined 2D-DCT and quantization architecture for JPEG image compression," in *Information Technology (ITSim), 2010 International Symposium in*, vol. 1, pp. 1–6, June 2010.

[38] C. Loeffler, A. Ligtenberg, and G. Moschytz, "Practical fast 1-D DCT algorithms with 11 multiplications," in *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*, vol. 2, pp. 988–991, May 1989.

[39] 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, Feb. 1995.

[40] E. Fossum, "CMOS image sensors: electronic camera-on-a-chip," *Electron Devices, IEEE Transactions on*, vol. 44, pp. 1689–1698, Oct. 1997.

[41] A. El Gamal and H. Eltoukhy, "CMOS image sensors," *Circuits and Devices Magazine, IEEE*, vol. 21, pp. 6–20, May/June 2005.

[42] P. Magnan, "Detection of visible photons in CCD and CMOS: A comparative view," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 504, no. 1-3, pp. 199–212, 2003.

[43] A. Lustica, "CCD and CMOS image sensors in new HD cameras," in *ELMAR, 2011 Proceedings*, pp. 133–136, Sept. 2011.

[44] Altera Corporation, San Jose, *Quartus II Handbook*, 12.1 ed., June 2012.

[45] D. Bailey, "Image border management for FPGA based filters," in *Electronic Design, Test and Application (DELTA), 2011 Sixth IEEE International Symposium on*, pp. 144–149, Jan. 2011.

Appendix A

JPEG Huffman Tables

The example Huffman tables provided in the JPEG standard Annex K and utilised in this project are in Table A.1 (DC table) and Tables A.2,A.3,A.4,A.5,A.6,A.7 (AC table).

Category	Code Length	Code Word
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

Table A.1: Luminance DC coefficient differences (JPEG Standard Annex K)

Run/Size	Code Length	Code Word
0/0 (EOB)	14	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

Table A.2: Luminance AC coefficients (part 1 of 6)

Run/Size	Code Length	Code Word
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
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

Table A.3: Luminance AC coefficients (part 2 of 6)

Run/Size	Code Length	Code Word
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

Table A.4: Luminance AC coefficients (part 3 of 6)

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

Table A.5: Luminance AC coefficients (part 4 of 6)

Run/Size	Code Length	Code Word
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
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

Table A.6: Luminance AC coefficients (part 5 of 6)

Run/Size	Code Length	Code Word
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

Table A.7: Luminance AC coefficients (part 6 of 6)

Appendix B

JPEG Header

The JPEG standard specifies the application markers in Table B.1.

Table B.1: JPEG application markers

`0xFFC0`	Start of Frame Marker: Baseline DCT (SOF_0)
`0xFFC4`	Define Huffman tables (DHT)
`0xFFD8`	Start of image (SOI)
`0xFFD9`	End of image (EOI)
`0xFFDA`	Start of scan (SOS)
`0xFFDB`	Define quantisation table(s) (DQT)
`0xFFDC`	Define number of lines (DNL)
`0xFFDD`	Define restart interval (DRI)
`0xFFE0`	Application marker 0 (APP0)

The JPEG JFIF file as created by the encoder in this project consists of the following stages:

- SOI (marker)
- JFIF Header
- Tables
- Frame Header
- Scan Header
- *data*
- EOI (marker)

Content of each section is specified in the following tables.

B.1 JFIF Header

Table B.2: JFIF Header

Content	Data
APP0 marker	`0xFFE0`
JFIF Header length	`0x0010`
JFIF in ASCII	`0x4A46494600`
JFIF version	`0x0102`
JFIF parameters	`0x00000100010000`

B.2 Tables

The tables consist of the quantisation table specification, followed by the Huffman table specifications and the restart interval.

Table B.3: Quantisation Table

Content	Data
DQT marker	0xFFDB
Section length	0x0043
Parameters	0x00
Quantisation table in zig-zag order	0x0A07070807060A0808080B0A0A0B0E17 0x100E0D0D0E1C15151117252025252020 0x2020252B3333252B332B202033403333 0x4040404040252B404040404033404040

Table B.4: Huffman Tables

Content	Data
DHT marker	0xFFC4
Section length	0x00D2
DC Parameters	0x00
DC Huffman table	0x000105010101010101000000000000000
	0x000102030405060708090A0B
AC Parameters	0x10
AC Huffman table	0x0002010303020403050504040000017D
	0x01020300041105122131410613516107
	0x227114328191A1082342B1C11552D1F0
	0x2433627282090A161718191A25262728
	0x292A3435363738393A43444546474849
	0x4A535455565758595A63646566676869
	0x6A737475767778797A83848586878889
	0x8A92939495969798999AA2A3A4A5A6A7
	0xA8A9AAB2B3B4B5B6B7B8B9BAC2C3C4C5
	0xC6C7C8C9CAD2D3D4D5D6D7D8D9DAE1E2
	0xE3E4E5E6E7E8E9EAF1F2F3F4F5F6F7F8
	0xF9FA

Table B.5: Restart interval specification

Content	Data
DRI marker	0xFFDD
Section length	0x0004
Restart not used	0x0000

B.3 Frame Header

Table B.6: Frame Header

Content	Data
SOF marker	`0xFFC0`
Section length	`0x000B`
Frame height	`0x01E0`
Frame width	`0x02F0`
Sampling parameters	`0x01001100`

B.4 Scan Header

Table B.7: Scan Header

Content	Data
SOS marker	`0xFFDa`
Section length	`0x0008`
Image component count	`0x0100`
Huffman tables	`0x00`
Scan parameters	`0x00003F00`