Implementing Real-Time Video Deblocking in FPGA Hardware

by

Martin Hansen

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Video compression techniques are commonly used to meet the increasing demands for the storage and transmission of digital video content. Popular video compression techniques such as MPEG video encoding make use of block-transform coding algorithms which are susceptible to blocking artifacts. These artifacts can be reduced using a deblocking process, of which there are many. However, those deblocking algorithms which provide noticeable improvements in visual quality also tend to be computationally expensive and unsuitable for real-time video use.

This dissertation selects and examines an appropriate algorithm for real-time video deblocking applications, and describes its hardware implementation on a Altera Cyclone II FPGA. The chosen algorithm is based on the concept of shifted thresholding; it reduces computational complexity by several means, such as by using only integer arithmetic and by replacing division operations with bit shifting. The implementation leverages the reduced hardware complexity of the chosen algorithm to cost-effectively implement real-time video deblocking.

Acknowledgements

I would like to thank my co-supervisor, Dr. Bill Bishop, for his continuous support and assistance, as well as my co-supervisor Dr. Wayne Loucks for his ongoing feedback and insightful suggestions. This work would not have been possible without both of their contributions as well as both of their support for my academic career.

I would also like to thank Altera Corporation for their substantial hardware contributions to the University of Waterloo's Electrical and Computer Engineering Department, without which this work would not have been possible. Finally, I would like to thank the members of the Parallel and Distributed Systems Group at the University of Waterloo, particularly Alex Wong.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation describes and examines a novel hardware implementation of real-time video deblocking. Although many types of deblocking implementations exist, most have either high computational cost or provide lower quality results. This project implements a recently proposed algorithm which delivers high quality deblocking with reduced hardware complexity. Reducing the hardware requirements allows for effective deblocking to be accomplished using low-cost hardware devices. This result is ideal for integration into consumer electronic devices.

1.1 Motivation

Deblocking algorithms are a type of image manipulation which is useful when an image (or a frame of video) has been compressed using block-based compression techniques. These types of compression, such as JPEG for images [Wal91] and MPEG for video [ISO94] are extremely prevalent in most consumer electronic de-

vices, such as phones, personal data assistants (PDAs), and digital television systems. Compression allows for efficient usage of storage space and transmission bandwidth. However, to greatly reduce the amount of data used in these devices, often media are too greatly compressed.

When overly reducing image and video file sizes, the quality of the media suffers. To address this loss in quality without affecting the desirable savings in space that have been achieved, deblocking can be used. Deblocking aims to correct one prevalent artifact caused by block-based compression, that of discrepancies at the block boundaries. Figure 1.1 displays an image that has been compressed using the JPEG standard, resulting in blocking artifacts, and the corresponding image after deblocking.

The deblocking process can be time-consuming and computationally intensive. To achieve real-time video deblocking, 24 frames (images) per second must be processed. Attaining any significant increase in image quality at this speed can be quite challenging. This difficulty, along with the goal of targeting consumer electronics, make this project suitable for hardware development, as opposed to software development. Hardware allows a customized solution, one which can be designed specifically for low-cost devices such as Field Programmable Gate Arrays (FPGAs).

The usefulness of such hardware implementations has not gone unnoticed. VIA Technologies' PN800 mobile chipset includes a video deblocking tool [VIA07], and ATI's Radeon X800 consumer video card contains video deblocking functionality as well [Alt07]. H.264-compressed video in particular has several proposed hardware deblocking architectures [PH06] [SCL06] [SZZ04].

The development of a real-time hardware video deblocker allows a wide range of

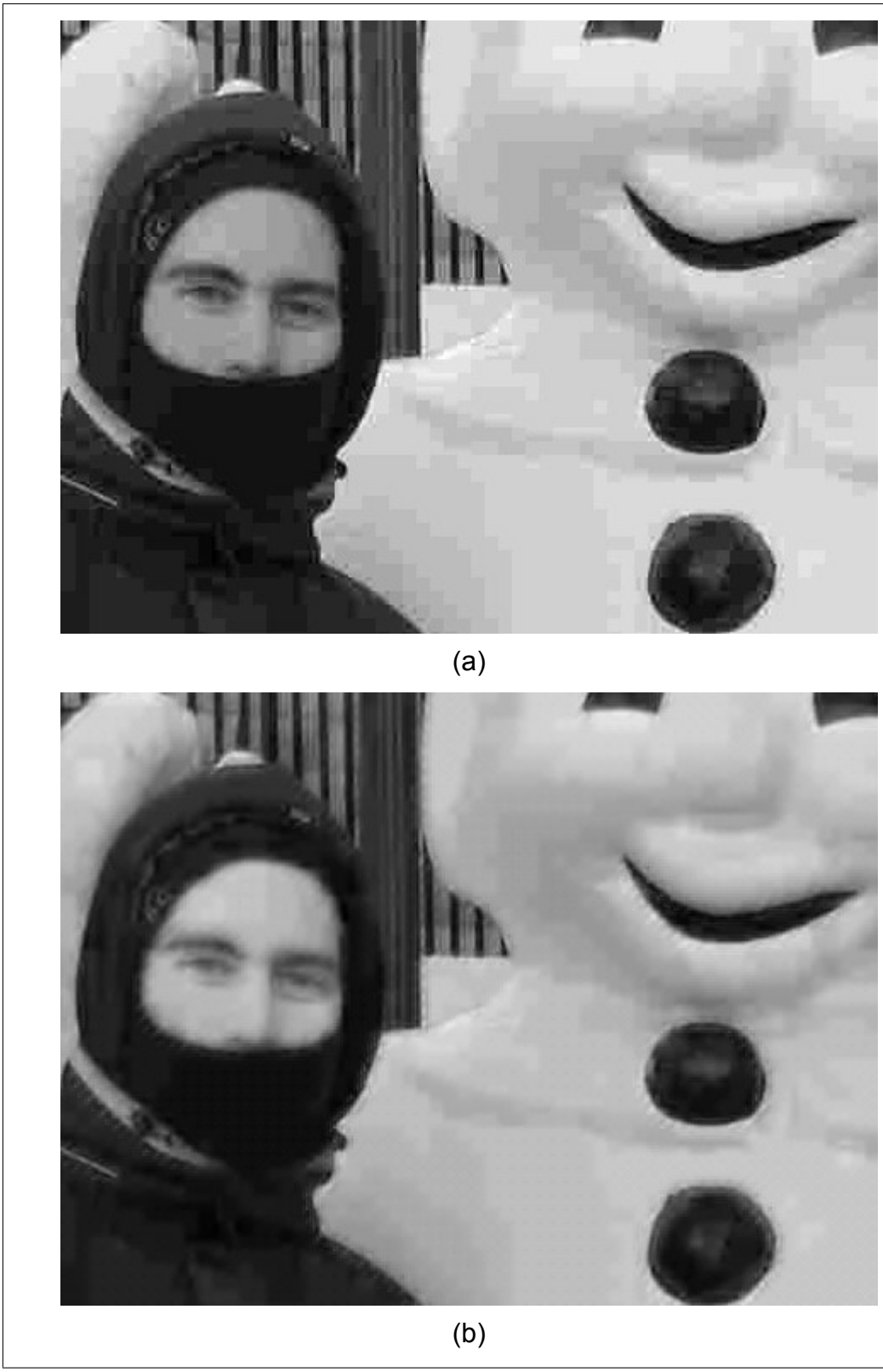

Figure 1.1: JPEG-compressed image, before (a) and after (b) deblocking

devices and applications to transmit and store video (as well as image) data more efficiently, with enhanced media quality and end-user satisfaction.

1.2 Statement of Thesis

It is my thesis that a reasonable quality of real-time video deblocking can be implemented in low-cost hardware, suitable for use in consumer electronics. This implementation would allow a reduction in media file sizes through block-based compression, while limiting visual quality degradation. Real-time deblocking allows for more efficient use of digital storage and data transmission, which, notably for mobile devices, can be a major concern.

One relevant application is high definition digital television. A real-time video deblocking system could be integrated in the end-user signal processor, allowing transmission costs to be reduced, which would allow bandwidth to be used for other applications such as enhanced digital television content or higher resolution television.

1.3 Contributions

This thesis makes the following significant contributions:

1. selects a preferred deblocking algorithm for implementation in hardware;
2. describes the modifications necessary to the chosen deblocking algorithm for further applicability to a hardware environment as opposed to a software environment;

3. details an implementation of the algorithm on a low-cost FPGA, including further optimizations which are required due to area and throughput requirements for real-time applications; and

4. analyzes the feasibility of the given implementation for real-time video deblocking, including the feasibility for usage in high-definition television.

1.4 Thesis Outline

Chapter 2 introduces the field of image and video compression, showing the importance of video deblocking and the situations in which it is applicable. This chapter also introduces some hardware considerations, and includes information on the hardware available for this project. Chapter 3 selects a preferred algorithm for implementation, based on its quality of results and on its perceived suitability for hardware development. Chapter 4 details the implementation itself, including explanation of the optimizations used, and some analysis of the limitations of the hardware. Also included is a commentary on means of improvement for the hardware design process. Finally, Chapter 5 analyzes the performance of the implementation and provides suggestions for future work, and Chapter 6 concludes the document with a discussion of the results.

Chapter 2

Background

Video and image data is heavily used in consumer applications and entertainment, among other areas. To supply consumers with this data, efficient representation of the multimedia information can be an important requirement [GES+99], and is often indispensable [CKL06]. Unlike the compression of textual and other data, compression of video and images, as well as audio, can be lossy. That is, data can be lost with the end product still appearing acceptable and reasonably unchanged to the end user. This approach allows a greater reduction in data size. Since lossy compression achieves noticeably smaller file sizes than lossless techniques, the former has become a much more common method of compressing video and images. The primary challenge becomes to find the optimal balance between file size and image quality; in other words, to find the optimal compression ratio.

In many areas of consumer entertainment, such as in video transmitted over the internet, quality becomes a minor consideration in comparison to the data transfer rate, and thus, the amount of data to be transferred. The result is that greater compression ratios are used, which improves user satisfaction in the primary area

(transfer rate), but lowers it in the secondary (quality). To address this quality issue without increasing transfer rates, much research has been done in image processing. The goal becomes to continue transmitting small (lower-quality) images but to then improve the quality through post-processing once the image is contained locally.

This chapter introduces techniques used for image compression, extends this knowledge to describe a method of quality improvement (reduction of blocking artifacts), and describes the role that hardware plays in this field. The descriptions refer specifically to image manipulations, with the implication that all techniques are extendable to video applications, as video can be viewed as a collection of images (frames) in this context.

2.1 Image Compression

Block-based image compression schemes are very widely used as a way of obtaining a large (for example, 20-fold) reduction in image file size [CAGM94]. These schemes are particularly useful in images which contain gradual changes, such as photographs, as opposed to those with distinct boundaries and solid surfaces, such as textual data and frames of animation. Although this compression can still be used on the latter cases, image degradation is more perceptible. Despite this issue, block-based schemes are still often used for all types of image data.

Generally, block-based compressions operate on a (most commonly 8 pixel by 8 pixel) block of the image at a time, transforming the data into a different domain and retaining only the useful components. The widely used JPEG (Joint Photographic Experts Group) standard uses a Discrete Cosine Transform (DCT) to calculate 64 frequency components from the original 64 pixels [Wal91]. These

frequency components can be selectively quantized; low-frequency coefficients are likely to be larger than high-frequency coefficients [Wal91]. Those coefficients which are zero can be eliminated, greatly reducing the total volume of data. To reconstitute the image, the inverse process is applied, using an Inverse DCT (IDCT). The JPEG encoding system can be seen in Figure 2.1.

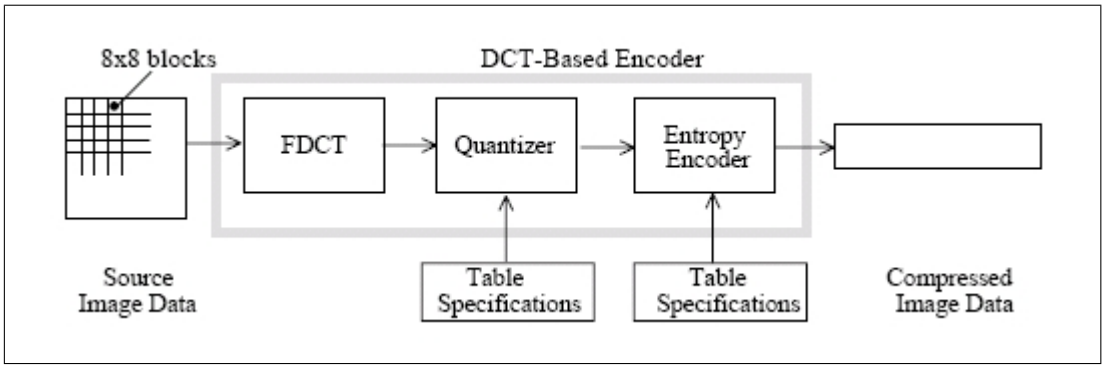

Figure 2.1: JPEG encoder processing steps [Wal91]

Although JPEG can be used for lossless compression, where the original image can be exactly recreated, it is more commonly used in a lossy manner to obtain great reductions in file size with (hopefully) little reduction in image quality. This lossy nature can manifest itself in different ways, through blocking artifacts, striping, blurriness, and so on. As can be seen in Figure 2.2 (details in Table 2.1), these effects become more apparent as the compression ratio is increased. The Peak Signal to Noise Ratio (PSNR) is a measure of the image quality, measured relative to image a (larger values are better)[1]. As mentioned, artifacts can be particularly obvious in non-photographic data, where the assumption of gradual change is less valid. For example, in drawn images, the transition between two surfaces is an immediate change, which can be difficult to represent accurately using a JPEG

[1]Measurements were done with MSU Video Quality Measurement Tool

image format with a high compression ratio.

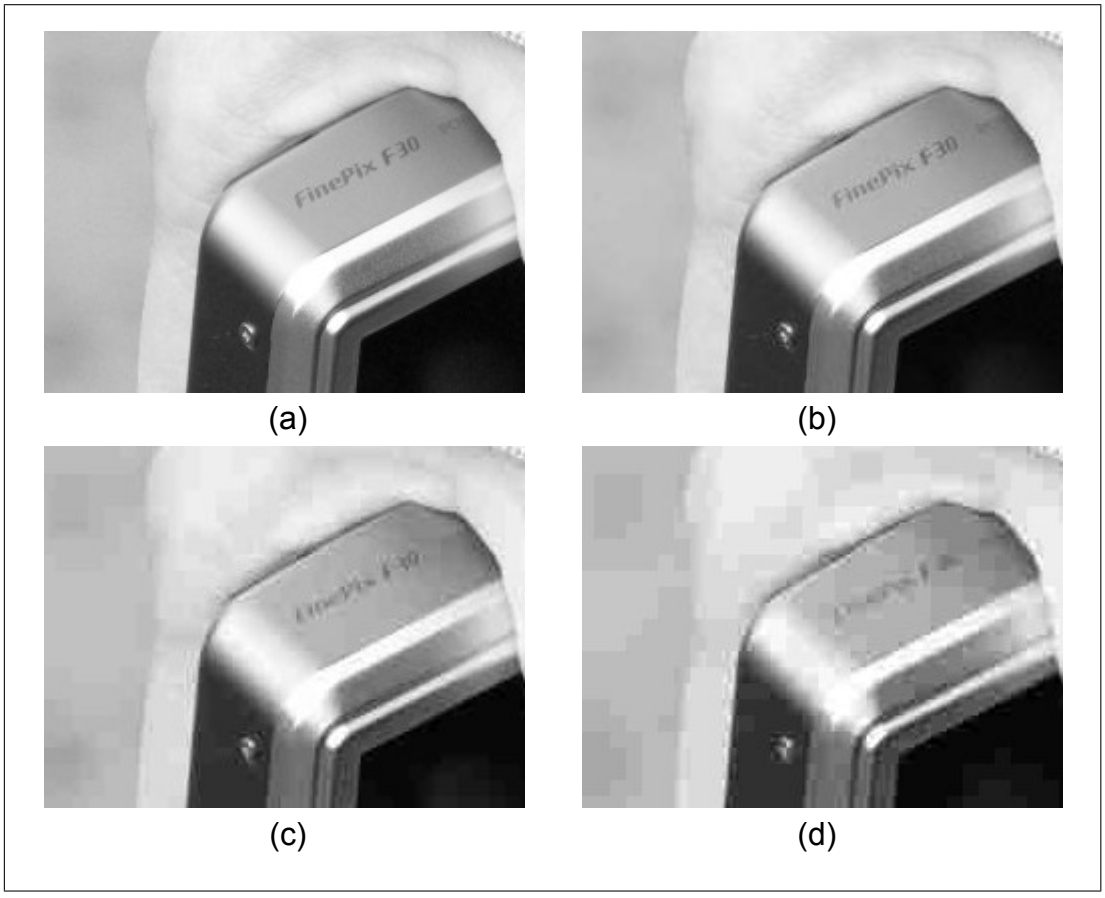

Figure 2.2: JPEG compression at different ratios (partial image)

2.1.1 Implementations of Block Operations

Block-based compression is based on the principle of transforming blocks of pixels into a different domain, which can represent the same visual image using a smaller amount of data. Commonly, 8 by 8 blocks of pixels are used. Most block transform algorithms use the DCT or variants thereof, although wavelets are used in some

Image	PSNR (dB)	Image Sizes (bytes)		Compression Ratio	Comments
		Initial	Final		
a	n/a	175 824	175 824	1:1	Initial JPEG
b	38.4	175 824	33 072	5.3:1	Negligible quality loss
c	34.9	175 824	20 430	8.6:1	Noticeable quality loss
d	32.4	175 824	16 079	10.9:1	Questionable quality

Table 2.1: JPEG compression data for Figure 2.2

newer algorithms such as JPEG 2000. In JPEG images, the DCT is used as the transform operation (Equation 2.1).

$$\mathbf{F}_{\text{out}}(u,v) = \tfrac{1}{4} C(u)\, C(v) \times \left[\sum_{x=0}^{7} \sum_{y=0}^{7} \mathbf{F}_{\text{in}}(x,y) \cos\tfrac{(2x+1)u\pi}{16} \cos\tfrac{(2y+1)v\pi}{16} \right], \qquad C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}}, u, v = 0 \\ 1, otherwise \end{cases} \tag{2.1}$$

The DCT can be implemented in various ways. A simple representation is using matrix operations: two matrix multiplications involving the 8×8 transform matrix $\mathbf{B}$ (Equation 2.2). This approach is particularly intuitive as the input is already a two dimensional array (matrix) of values.

$$\mathbf{F}_{\text{out}} = \mathbf{B}\mathbf{F}_{\text{in}}\mathbf{B}^{\mathbf{T}} \tag{2.2}$$

Ideally, these calculations can be applied losslessly; using a DCT followed by an IDCT will mathematically return an exact copy of the original block. In practical

terms, the calculations can not be done with infinite precision and so some loss can occur. However, again in practical terms, JPEG is most often used as lossy compression, using quantization between the DCT and IDCT operations, and thus these calculation losses are irrelevant.

2.2 Image Deblocking

As explained previously, extensive compression results in lower visual quality of the image. The quality can be affected in many ways, such as with blurriness or loss of detail. Only one symptom will be addressed here: blocking artifacts.

Blocking artifacts come about from compression schemes that use blocks. The artifacts are discrepancies in the continuity of the image, and occur at the block boundaries; since each block is compressed independently of the others, the pixels at the right edge of one block are affected differently by the compression process than their neighbouring pixels which exist at the left edge of the adjacent block. Thus, the pixel transitions within the block are acceptable but the transitions from block to block have noticeably worse continuity, which degrades the subjective visual acceptability of the image (as shown in Figure 2.3).

There are several approaches that can be taken to reduce these artifacts. Some techniques achieve quite good results but require a great deal of computation. In these cases there is much less possibility of a real-time implementation for video deblocking, due to the time constraints for processing each frame. Other techniques have very low costs but only produce marginal improvements to the image. Again this situation is not ideal. The preferred technique should have reasonably low computation requirements but still deliver a noticeable (although not necessarily

Figure 2.3: Example (partial) image with noticeable blocking artifacts

the best) reduction in blocking artifacts.

2.2.1 Deblocking Techniques

A large number of video and image deblocking methods have been introduced. These methods have been categorized [WB06] as follows:

1. projections onto convex sets (POCS) methods [WLY02];
2. spatial block boundary filtering methods [CCR98];
3. wavelet filtering methods [XOZ97];

4. statistical modeling methods [OS95];

5. constrained optimization methods [HCS95]; and

6. shifted transform methods [Nos01].

Although these algorithms greatly enhance image quality, the majority are unacceptable for real-time deblocking, since they require large amounts of computation time or memory. To develop an inexpensive hardware deblocker suitable for operating on real-time video, these costs must be addressed.

Modern shifted transform algorithms have been shown to offer better performance than techniques such as those based on POCS and wavelets [WB06]. They are often very expensive in terms of computation for a real-time application, although not in comparison to many other methods. Despite requiring less calculation than methods based on constrained optimization, there is still a great deal of calculation required, including 64 DCT and IDCT operations per 8×8 pixel block. Recently however, an algorithm based on shifted transforms was proposed by Wong, which offers competitive results while reducing much of the computation required by earlier methods [WB06].

2.2.2 Comparison of Techniques

To implement a deblocking system in hardware, the chosen algorithm must be computationally efficient enough to provide the opportunity for video deblocking. In other words, the algorithm should be simple enough to be able to deblock 24 images per second using off-the-shelf hardware. Within this constraint, there still needs to be a noticeable subjective reduction in blocking artifacts.

Table 2.2 summarizes the complexity and results of leading methods from various techniques. Although several of the techniques offer good results, their complexity makes them unsuitable for implementation in a low-cost hardware device.

Deblocking Method	**Mathematical Workload**	**Quality of Deblocking**
Projection Onto Convex Sets (POCS)	high	high
Spatial block boundary filtering	low	med
Wavelet filtering	high	high
Statistical modeling	high	high
Constrained optimization	high	med
Shifted transforms	med	high

Table 2.2: Comparison of deblocking techniques

Shifted transforms is a promising deblocking approach. Although historically computationally intensive, the shifted transform implementation proposed by Wong uses much less calculation than in previous implementations [WB06]. Not only is the number of DCTs and IDCTs greatly reduced, hardware-friendly features are also included, such as removal of floating-point operations.

2.3 Hardware Considerations

There are various hardware platforms that can be used in implementation of an algorithm. Both application-specific and non-application-specific devices are available. The target hardware platform for this project is a low-cost FPGA device.

A FPGA is a programmable hardware device that can implement digital logic customized by the designer, which is downloaded directly to the device without additional manufacturing [Kim00]. An Application-Specific Integrated Circuit

(ASIC), however, is manufactured with a specific design already implemented; it comprises hardware that runs pre-determined logic which cannot be easily modified after production [ZRG+02]. Both of these solutions are customized to be application-specific. Non-application-specific solutions are those such as Digital Signal Processors (DSPs) or microprocessors.

2.3.1 Non-Application-Specific Devices

Although a variety of general use processors are available, the only general device under consideration for this project is a DSP. Since DSPs are designed for digital signal applications, they are more appropriate than general-purpose processors for image and video processing.

A DSP would be required to fetch pixels from memory, perform calculations, and save out the results. Although these operations could be done efficiently, it is not possible to attain the level of customization inherent to an application-specific device. However, DSPs are often clocked at higher speeds than FPGAs, giving them a possible advantage in terms of latency of operations.

Despite the speed advantages of a DSP, they generally do not achieve the data rate possible with a FPGA for video processing operations [SVMJ95]. The higher data rate in an application-specific device results from the customization that is possible. Parallelism, pipelining, and detailed optimization can be used to a great extent to increase the throughput of a specific project.

2.3.2 Application-Specific Devices

ASICs are often used for high-production goods, and for high-speed applications or designs which have stringent area or power constraints. FPGAs, in general, require more power and are limited to slower clock frequencies, as well as consuming more transistors and more physical area. They do have the advantage of flexibility; they can be continually reprogrammed by the developer. Both devices can be found at various costs. However, ASICs are only cost-effective when produced in very large quantities, for example requiring 250 000 or more units in a 2-pass process before attaining a die price which is competitive with comparable FPGAs [ZRG+02]. Because of their availability in smaller quantities and their ability to be continually reprogrammed, FPGAs are often used for prototyping of products during design phases.

For these same reasons, a FPGA is the preferred device for development of this project. This device can allow the designer to move beyond simulation results and view hardware behaviour at every stage of the design without needing to manufacture a new integrated circuit each time. Although area usage is greater and the operating clock speed is slower, a successful design on a FPGA can demonstrate the feasibility of the chosen algorithm without incurring the high costs of producing an ASIC in small quantities.

It should be noted that in addition to an FPGA, the project requires memory to store the initial and final images. For the image size being used (640×480 1-byte pixels), 307 200 bytes are needed. Therefore the minimum size of memory needed is 512 kByte, given that memory is only available in sizes of powers of two. Either one or two memories of this size could be required, depending if the initial image and final image can occupy the same space.

2.3.3 Hardware Implementation

In contrast to software development, hardware development has a higher likelihood of incurring more difficulties and of requiring a longer time to complete. For implementation of the same algorithm, software can often rapidly prototype the required functionality whereas hardware may take substantially more time to develop. Even using an FPGA, where much of the timing and routing complexity is handled by the compiler, there are still difficulties.

A primary time cost is the slowness of the debugging process. Since a complete hardware design for a complex system can often take 30 minutes to 60 minutes to compile (in comparison to 2 minutes to 10 minutes for a similar software project on the same computer), making incremental changes to locate errors can be an arduous and sometimes even infeasible process. Furthermore, the avenues available for debugging information are often less convenient. Often there are only simple outputs such as LEDs that can be used for user feedback. The signal values within hardware can be accessed through simulation and other means, but this approach can often be time consuming to initiate. Finally, although the system can sometimes be partitioned into smaller pieces for debugging, this is not always convenient due to the highly parallelized and interdependent nature of many hardware implementations.

This interdependent nature also creates further difficulty in debugging, since it implies additional complexity. To generate high throughput it is common to create a very concurrent style of solution, which can require quite a bit of time to design and understand. This complexity influences the debugging process, making even simple errors often very time-consuming to diagnose.

2.3.4 Hardware in Video Processing

Given the slower development inherent with implementing hardware [AM02], there must be suitable justification for undertaking a hardware design as opposed to simply designing a software solution; there need to be some benefits to a hardware solution, such as performance or cost.

Indeed there are many examples of the preference for hardware designs in video and image processing. When JPEG images must be compressed and/or decompressed in real-time, conventional software implementations are inefficient when used in hardware devices [EPG+05]. In general, digital signal processing algorithms in embedded applications, including image processing algorithms, require high processing power which can be more readily achieved through hardware parallelism [AM02]. On slower consumer products, such as digital television receivers, using a software design run on a microprocessor would not allow the kind of customization and parallelism that is helpful for obtaining real-time throughput. Even having a separate general-purpose microprocessor dedicated to a given algorithm might be insufficient in terms of throughput, and inefficient in terms of power usage, depending on the application.

Using an application-specific hardware device rather than a microprocessor can resolve these performance concerns without needing to increase the clock speed of the system. FPGAs have been used for several different image processing applications such as edge detection [THAE00] [HLC+05], rank-order filtering [Nel00], and convolution [Nel00]. Intuitively, this style of implementation makes sense for many complex algorithms that would be run continually, in real-time, and with hard timing constraints. One such example is a high-speed edge and corner detector [THAE00]:

> "Several computer vision applications involve the computation of a large number of repetitive operations over the entire input image in order to analyze the image contents and recover useful information. However, this task could be complex and computationally intensive. An alternative solution to software implementation is the design of specific hardware for computer vision in order to perform a high rate of operations per second."

There are existing hardware video deblockers, particularly for the H.264 compression standard. Some offer excellent performance, but less noticeable qualitative improvements to the deblocked video [SCL06]. Others meet real-time requirements but only for smaller frame sizes such as 352×288 and smaller [PH06] [SZZ04]. These deblockers also have been implemented at clock speeds greater than the 50 MHz available for this project's prototype.

The proposed architecture is novel in two ways. First, it is targeted at low-cost hardware. Second, the algorithm itself is one which has not previously been implemented in hardware. Furthermore, the deblocking rates achieved are competitive with or better than those of existing architectures.

2.3.5 Altera DE2 Board

The Altera DE2 Educational Development Board (Figure 2.4) is a new hardware platform available for use in the Department of Electrical and Computer Engineering at the University of Waterloo. One of these boards was used to implement this project.

The DE2 has a wealth of features, most of which are not of immediate use to

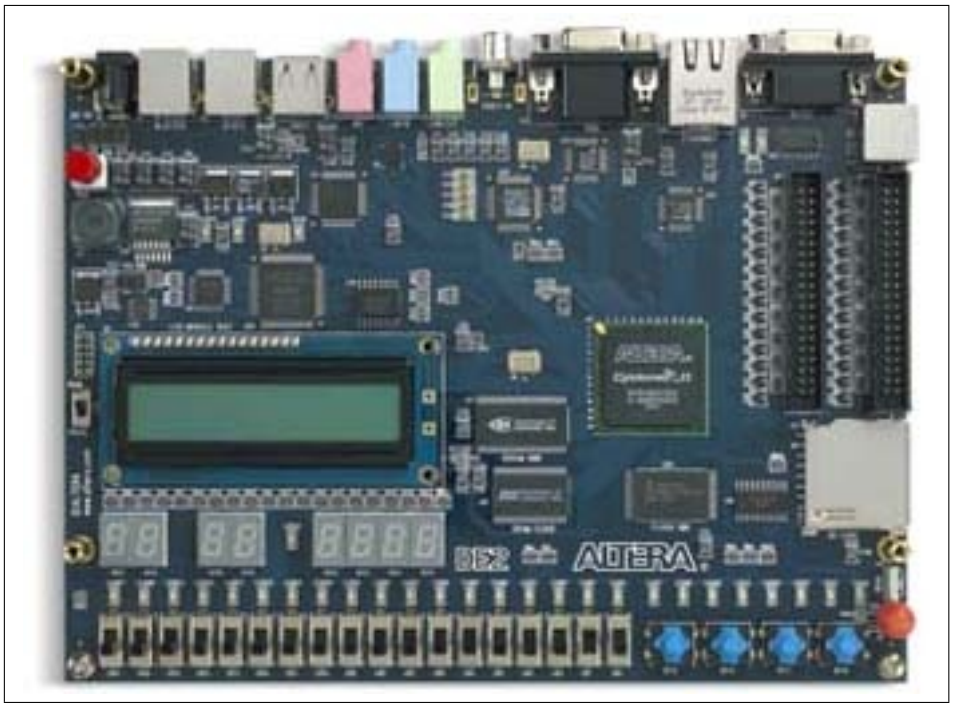

Figure 2.4: Altera DE2 development board [Alt06]

the project, but many of which could be used to extend the prototype design in the future. These features include a USB (Universal Serial Bus) connector, ethernet connector, VGA (Video Graphical Adaptor) output, audio connectors, and a small LCD (Liquid Crystal Display). Also available to the designer for input and output are an 8 character Seven Segment Display (SSD), 18 switches, 4 pushbuttons, and 27 LEDs (Light Emitting Diodes) in green and red. The many features of this board make it suitable for the rapid prototyping of hardware designs.

The central component of the board is the Altera Cyclone II FPGA; in this case, a model EP2C35F672C6 device. It contains 33 216 logic elements (LEs), 105 M4K memory blocks (comprising a total of 60 480 bytes of memory), and 35 embedded multipliers (Table 2.3).

Feature	Quantity
LEs	33 216
M4K RAM blocks (4 kbits plus 512 parity bits)	105
Total RAM bits	483 840
Embedded multipliers	35
PLLs	4
Maximum user I/O pins	475

Table 2.3: EP2C35 Cyclone II FPGA features

The memory capacity on the board consists of three chips: an 8 MByte SDRAM (Synchronous Dynamic Random Access Memory), a 512 kByte SRAM (Synchronous RAM), and a 1 MByte FLASH memory. There is also a SD card slot on the board that can be used for storage, when a SD card is installed. A summary of these memory options, in addition to the on-chip M4K memory, can be seen in Table 2.4.

Memory	Size (MB)	Speed (MHz)	Data Width (bits)	Notes
SDRAM	8.00	100	16	Requires precharging and other operations
FLASH	1.00	10	8	Limited lifespan
SRAM	0.50	100	16	
M4K	0.06	50	n/a	On-chip
SD slot	n/a	25 to 50	16	Requires SDRAM operations and SD card

Table 2.4: DE2 board memory options comparison

The DE2 is a capable system that contains the base requirements for development of a real-time hardware image deblocker: a FPGA of reasonable size as well as several memory options large enough to store an input test image and output image. The board has several advantages, including its availability for this project. It has a large number of simple inputs and outputs which can be used for debugging purposes. There also are plentiful options for expansion of the base project. For

example, deblocked images could be sent to the VGA output for immediate display on an external monitor.

Chapter 3

Shifted Deblocking Algorithm

Before implementing deblocking in hardware, an algorithm must be selected that meets the goals of the project. The primary concern is ease of implementation: an algorithm which takes into account the limitations of hardware is preferable to one that is much more complex but produces better results. The algorithm must provide noticeably positive results while being feasible to fit physically on the FPGA, as well as being able to run at the desired speed. Thus the design priorities are:

1. low area, in order to have the design fit on the available hardware;
2. high throughput, producing 24 deblocked images per second; and
3. high quality, providing a noticeable reduction in blocking artifacts.

As described briefly in the previous chapter, the best fit to these requirements is the shifted thresholding method designed by Wong [WB06]. Wong's algorithm contains many features that make it suitable for hardware implementation. The

primary benefit is that the overall algorithm is reasonably simple, implying a greater chance of completing the deblocking process quickly and therefore implying a greater chance of deblocking video in real-time. It is also designed with hardware considerations in mind by removing floating point operations, replacing division operations with bit shifting, and reducing multiplications for the transform and inverse transform (down to 19.5 per pixel as compared to 2016 per pixel in earlier shifted transform methods). Furthermore the algorithm is well-suited to pipelining.

3.1 Overview

A structural overview of deblocking algorithms that utilize shifted transforms is shown in Figure 3.1. Wong's method follows this structure. The image is shifted based on n shift patterns and the outputs of these shifts ($S_1 \dots S_n$) are transformed into another domain with transform operator T. The shifted transforms are then filtered using operator F, inverse transformed with T^{-1} and inverse-shifted based on the corresponding shift pattern. Finally, the images are averaged together to form the final output image. Wong's algorithm contains 4 shift patterns: $(\Delta x, \Delta y) = (-3,-3), (-1,-1), (1,1), (3,3)$. The transform used is a Discrete Cosine Transform (DCT), with a corresponding Inverse DCT used as the inverse transform. The filtering operation is a thresholding based on the image's quantization matrix.

3.2 Image Shifting

The first stage in the algorithm is the shifting stage. The image is traversed block by block, first for the first shift pattern $(-3,-3)$ and subsequently for the others, to

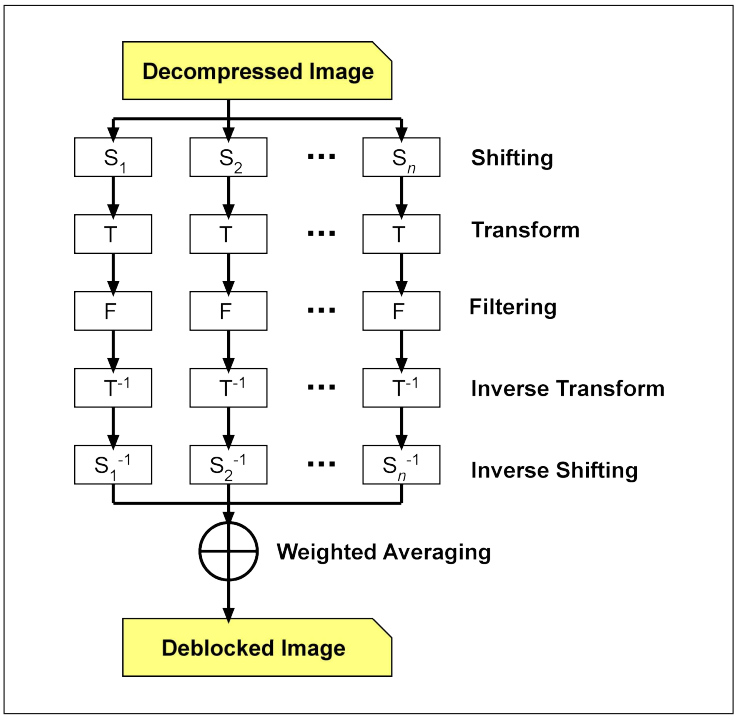

Figure 3.1: Overview of shifted transform deblocking [WB06]

result in 4 scans of the full image (Figure 3.2). Therefore, the bulk of the deblocking process must be applied over 4 different copies of the image, once for each shift. Since the shifting operations traverse the boundaries of the image, the exterior of the original image is zero-padded by 3 pixels in each direction. The selection of only 4 shift patterns is a vast improvement over other shifted deblocking methods, which can use up to 64 shift patterns and therefore require approximately 16 times more computation to be done.

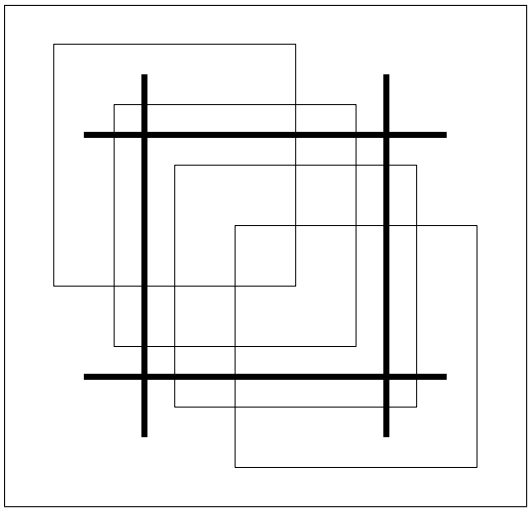

Figure 3.2: Image shift patterns: (-3,-3), (-1,-1), (1,1), (3,3)

3.3 Transform and Inverse Transform

The second stage is the transform stage. A DCT is used to transform each block into the frequency domain. In general, the operations required for this are 2 matrix multiplications on the 8×8 block. For Wong's algorithm the matrix multiplications have been greatly simplified at the expense of adding two additional operations: a scalar multiplication and a scalar division by a multiple of 2. This change results in each of the two matrix multiplications containing almost exclusively operations on multiples of 2, which can be implemented using left bit shifts. The more difficult multiplicands are relegated to the final scalar multiplication, which is by nature a simpler operation and can accomplish these multiplications more easily.

The scalar division is incorporated to maintain integer arithmetic throughout the transform. After the many multiplications the terms are all scaled down, using a division by 2^{18}, which again can be implemented using bit shifting. Equation 3.1

shows the definition for the transform and for the integer matrices M, P, and E, which are derived from the original transform matrix B. The derivation for Wong's implementation of the transform can be seen in Equation 3.2, and the matrices themselves are shown in Equation 3.3 through Equation 3.5. X represents the input matrix and Y is the resulting output matrix, $\otimes$ represents scalar matrix multiplication, and >> denotes binary bit shifting.

$$\begin{aligned} \mathbf{Y} &= \mathbf{B}\mathbf{X}\mathbf{B}^{\mathbf{T}} \\ \mathbf{M} &\equiv \text{round}(\alpha\mathbf{B}) \\ \mathbf{M} &\equiv \mathbf{P} \otimes \mathbf{E} \end{aligned} \tag{3.1}$$

$$\begin{aligned} \mathbf{Y} &= \mathbf{B}\mathbf{X}\mathbf{B}^{\mathbf{T}} \\ \alpha^2\mathbf{Y} &= \left(\mathbf{M}\mathbf{X}\mathbf{M}^{\mathbf{T}}\right) \\ \alpha^2\mathbf{Y} &= (\mathbf{P} \otimes \mathbf{E})\,\mathbf{X}\,(\mathbf{P} \otimes \mathbf{E})^{\mathbf{T}} \\ \mathbf{Y} &= \left(\mathbf{P}\mathbf{X}\mathbf{P}^{\mathbf{T}}\right) \otimes \left(\mathbf{E} \otimes \mathbf{E}^{\mathbf{T}}\right) \big/ \alpha^2 \\ \mathbf{Y} &= \left(\mathbf{P}\mathbf{X}\mathbf{P}^{\mathbf{T}}\right) \otimes \left(\mathbf{E} \otimes \mathbf{E}^{\mathbf{T}}\right) >> (2\log_2\alpha) \end{aligned} \tag{3.2}$$

$$\mathbf{M} = \begin{bmatrix} 181 & 181 & 181 & 181 & 181 & 181 & 181 & 181 \\ 256 & 206 & 128 & 64 & -64 & -128 & -206 & -256 \\ 256 & 64 & -64 & -256 & -256 & -64 & 64 & 256 \\ 206 & -64 & -256 & -128 & 128 & 256 & 64 & -206 \\ 181 & -181 & -181 & 181 & 181 & -181 & -181 & 181 \\ 128 & -256 & 64 & 206 & -206 & -64 & 256 & -128 \\ 64 & -256 & 256 & -64 & -64 & 256 & -256 & 64 \\ 64 & -128 & 206 & -256 & 256 & -206 & 128 & -64 \end{bmatrix} \tag{3.3}$$

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 256 & 206 & 128 & 64 & -64 & -128 & -206 & -256 \\ 256 & 64 & -64 & -256 & -256 & -64 & 64 & 256 \\ 206 & -64 & -256 & -128 & 128 & 256 & 64 & -206 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 128 & -256 & 64 & 206 & -206 & -64 & 256 & -128 \\ 64 & -256 & 256 & -64 & -64 & 256 & -256 & 64 \\ 64 & -128 & 206 & -256 & 256 & -206 & 128 & -64 \end{bmatrix} \tag{3.4}$$

$$\left(\mathbf{E} \otimes \mathbf{E}^{\mathbf{T}}\right) = \begin{bmatrix} 32761 & 181 & 181 & 181 & 32761 & 181 & 181 & 181 \\ 181 & 1 & 1 & 1 & 181 & 1 & 1 & 1 \\ 181 & 1 & 1 & 1 & 181 & 1 & 1 & 1 \\ 181 & 1 & 1 & 1 & 181 & 1 & 1 & 1 \\ 32761 & 181 & 181 & 181 & 32761 & 181 & 181 & 181 \\ 181 & 1 & 1 & 1 & 181 & 1 & 1 & 1 \\ 181 & 1 & 1 & 1 & 181 & 1 & 1 & 1 \\ 181 & 1 & 1 & 1 & 181 & 1 & 1 & 1 \end{bmatrix} \tag{3.5}$$

The fourth stage of the algorithm, an inverse transform, occurs after the thresholding process in stage three. It is accomplished using an IDCT and proceeds similarly to the DCT; thus it is optimized in the same manner. The primary differences are that the operations proceed mostly in the inverse order, and that the transformation matrix used is the transpose of the one used in the DCT. The IDCT is shown in Equation 3.6.

$$\mathbf{X} = \left(\mathbf{P^T}\left(\mathbf{Y} \otimes \left(\mathbf{E} \otimes \mathbf{E^T}\right)\right)\mathbf{P}\right) >> (2\log_2 \alpha) \tag{3.6}$$

3.4 Thresholding

The thresholding takes place in the frequency domain, in stage three, between the DCT and IDCT. The frequency coefficients are compared with the values in the threshold matrix, and those coefficients below their respective threshold are set to zero. The threshold matrix $\mathbf{T}$ is derived from a scaled integer approximation of the quantization used to decompress the image (see Equation 3.7). This relationship allows the image to be adaptively filtered depending on the image quality. This stage is designed to filter the highest frequencies of each block.

$$\mathbf{T} = \mathbf{Q} >> 1 \tag{3.7}$$

3.5 Reassembly

Following the inverse transform the 4 shifted images must be recombined with each other, and with the original image. This recombination is accomplished using two

different averaging schemes. The first is a global average which assigns a weight of 1/2 to the original image and a weight of 1/2 to the sum of the shifted deblocked images. Therefore each shift pattern is given a weight of 1/8.

The second form of averaging is based on the location of each pixel within its block in the original image. The pixel's weight is calculated from the Euclidian distance between the pixel in question and the centre of its block. The implementation of this operation results in each block of pixels being weighted using one of two weight matrices, depending on whether the block is from the original image or from a shifted deblocked image. These weight matrices are designed such that the sum of their two values at any position is 256. Therefore, after this weighted averaging process the pixels can be normalized using a scalar division by 256, which is equivalent to a shift right by 8 bits.

The first averaging scheme is designed to give suitable weight to the original image. The second averaging scheme is designed to give added weight to the original image at the center of each block, and added weight to the deblocked images at the block boundaries, where the deblocking is most needed.

3.6 Image Format

The steps required have now been defined but the system requires a specific data format in order to create an implementation. This format will affect most facets of the implementation, such as the width of the datapath, and most importantly the specific details of the first and last stages, where the image is loaded and saved. In order to simplify the width of the datapath and the difficulty of recombination of the image, the original image is in greyscale as opposed to colour and is composed

of 8-bit pixels. The image format is a bitmap, for simplicity in accessing the pixel data. Therefore, to apply the algorithm no further conversion must be done; the file already explicitly contains the data value of each pixel. It is also implied that the output (deblocked) file will be of the same format, and thus the deblocked pixel data can simply replace the data of the original pixels without any change to the image file structure.

3.7 Hardware-Beneficial Features

As mentioned previously, Wong's algorithm contains several features that contribute to its ease of implementation in a hardware environment. A main benefit of this algorithm is its conceptual simplicity and perceived ease of implementation. Although shifted deblocking by its nature does not produce the best qualitative image results (as compared to other forms of deblocking), it does greatly reduce blocking artifacts with relatively low computational cost. Furthermore the selection of only 4 shift patterns (in comparison to as many as 64) allows for many less iterations of the deblocking process in comparison to earlier techniques. Another example of implementation simplicity is the algorithm's ease of discretization: it is quite straight-forward to divide the process into smaller steps, and even into different combinations of steps, which is ideal for a pipelined implementation as well as for load-balancing the pipeline as a means of optimization. The algorithm can be viewed as operations on one single block of pixels at a time, which is a further benefit.

Another major contribution of Wong's work is the large reduction in arithmetic, most notably in the DCT and IDCT operations. In comparison to the original

Nosratinia algorithm, his "requires approximately 103 times less multiplications and approximately 16 times less additions [...] making it suitable for real-time video deblocking purposes." [WB06] This reduction is accomplished through the optimizations to the transform matrices, allowing most of the multiplications to be reduced to trivial shift operations. Additionally, all divisions are designed such that their divisors are powers of 2, again resulting in shift operations as opposed to much slower divisions.

A third simplification is the elimination of floating-point operations. This improvement is accomplished mainly using scaling, which results in increased datapath width to match the increasing scale of the values being calculated. However the datapath still does not exceed 31 bits for any values since the operations are normalized at logical points using the above-mentioned divisions. One such example has already been discussed: the weighted pixel averaging in the Reassembly stage, which is normalized using division by 256. Overall the increased width is a relatively minor penalty in return for the ease of integer operations (both in terms of speed and area).

Chapter 4

Implementation

It is often beneficial, when implementing an algorithm in hardware, to deviate from the original sequential perception of that algorithm. Even with an algorithm that is explicitly designed to be "hardware-friendly", such as Wong's shifted thresholding algorithm, changes are often needed before implementing it. These changes can range from minor arithmetic manipulations to the regrouping of entire steps in ways that are conceptually different. Since the deblocking algorithm chosen does have many features that are conducive to a hardware implementation, it becomes interesting to note how many changes still have to be made throughout the hardware design process to accommodate restrictions such as memory bandwidth and physical FPGA area.

It is worthwhile to note that the chosen algorithm can indeed be implemented directly (without optimization or modification) on hardware that does not suffer from the above restrictions. In other words, there can still be a direct mapping from the initial design to the FPGA, resulting in a processor that modifies the input image as required. When this is done experimentally, two problems are

immediately apparent: low throughput and insufficient physical area. If the clock frequency of the hardware could be increased indiscriminately then throughput could be improved; since this is not the case, optimization is needed to process the data in a timely manner. Similarly, if the available area could be increased indiscriminately then there would be no need for the improvements that are required on this real platform. As described in Chapter 2, the hardware being used is clocked at 50 MHz and contains 33 216 logic elements.

Therefore, changes must be made. To improve throughput, pipelining is used, and to address area limitations various optimizations are made to the mathematics and the construction of the required operations. From a development standpoint, these optimizations can be challenging. Often it can be useful to implement functionality first and optimize second, to verify a baseline of functional correctness before modifying the original algorithm. For this project, the area constraints prevent this approach from taking place: in some cases even implementing just one unoptimized portion of the algorithm exceeds the available area, let alone implementation of the entire system. Therefore implementation and optimization must go hand in hand, and must be done iteratively, observing the tradeoffs that occur at different stages. Since compiling for this hardware at 90% or greater LE usage takes approximately 60 minutes, reducing the number of compiles is a significant concern in the development process and places great emphasis on developing the system in an efficient way.

Figure 4.1 illustrates a high-level view of the data flow within the hardware. At each stage, one 8×8 block of pixels is manipulated, and the design iterates over the entire image multiple times (five full passes are required). The implementations of the pipeline scheme, the memory systems, the transform operations, and the other

components are discussed in the following sections.

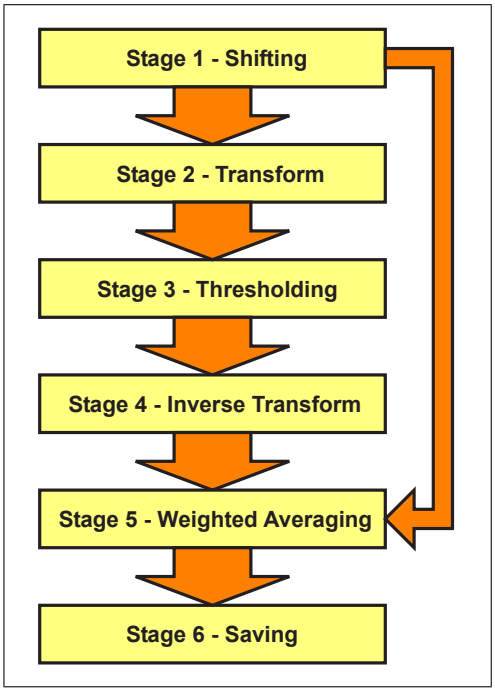

Figure 4.1: Data flow of the hardware implementation

4.1 Pipelining

Arguably the most important high-level architectural decision is the segmentation of the algorithm into pipeline stages. Pipelining is required in order to generate efficient throughput of the data. A comfortable balance needs to be found between pipeline stage length (clock cycles) and pipeline length (number of stages). Although dividing the algorithm into stages is trivial due to the sequential nature of its steps, more complexity is introduced when selecting their optimal arrangement.

For simplicity, the length of the stages in the pipeline, measured in clock cycles,

will be referred to as the *width* of the pipeline. More specifically, this term will refer to the length of the longest stage: the limiting value for the clock cycle length of the pipeline as a whole, or in other words, the time required for all data in the system to propagate one stage further in the pipeline. The term *pipeline length* will refer to the number of stages in the pipeline. Therefore the latency of one block of data can be calculated by length×width, and the throughput of one block of data can be given simply by the width.

One assumption that is made is that the datapath width of the pipeline must be one full block of 8×8 pixels. This assumption is made due to the nature of the transform and inverse transform operations. Since the transform of one pixel is dependent on the values of all 64 pixels in that block, the complexity of processing pixels individually is qualitatively far greater than the complexity of processing one full block at a time. Furthermore, processing pixels individually would result in greatly reduced parallelism and a great number of duplicated memory accesses. Therefore it is assumed that each pipeline stage will process 64 pixels.

The challenge of allocating operations to pipeline stages can be seen as similar to a knapsack problem [GJ79]. The knapsack problem is a well-known nondeterministic polynomial (NP-complete) problem, which is described by trying to maximize the possible value of a subset of objects while observing a given cost constraint. In this case, the goal is to maximize throughput while minimizing physical area. A primary way of decreasing area is to decrease parallelism, lowering the circuitry required to perform multiple tasks simultaneously. A primary way of increasing throughput is decreasing the pipeline width. So, one way in which this optimization goal can be expressed is to minimize pipeline width while minimizing parallelism.

The variables to be decided when designing the pipeline are the number of pipeline stages, the width of the pipeline, and the optimal arrangement of operations within these stages. As defined above, there are two primary objectives to consider: maximizing throughput and minimizing area.

First, to maximize throughput, the pipeline width should be minimal, allowing a fast rate in the data exiting the system. The width should be reduced as much as possible: it will be limited in this respect by the longest single block operation, which can be assumed at this point to be memory access as all other operations can theoretically be optimized to be as short as required (without considering area costs).

Second, reducing physical area is required in order to fit the design on the FPGA. Once the minimum pipeline width is set – it is convenient to estimate it to be 64 clock cycles, allowing 1 cycle per pixel for memory operations – it is difficult to quantify the remaining tradeoffs that occur in allocating pipeline stages. The allocation of deblocking operations to the stages is influenced by several factors in regards to area (which conflict with high throughput), as shown in Table 4.1.

Since a 64-pixel block of data is being manipulated at each stage, each stage requires at least 64 registers, each between 8 bits and 31 bits wide, simply for storage of the pixel data. These registers are a considerable area cost that can be reduced by designing a pipeline with a fewer number of stages. However this smaller length is not optimal for throughput, since having fewer pipeline stages implies that these stages will be longer, resulting in a large pipeline width and lower throughput.

An important note is that clock cycles can be "wasted". This term (which uses n_m to denote to the number of wasted clock cycles in stage m of the pipeline) refers to the difference in clock cycles between the pipeline width (PW) and the time

Proposed Tradeoff	**Throughput Effect**	**Area Effect**	**Limitations**	**Overall Effects**
Decreasing pipeline width	*Beneficial:* Increased throughput results directly from decreased width.	*Possibly detrimental:* Decreased width implies an increased number of stages, which requires additional storage registers.	Memory accesses require an estimated minimum of 64 clock cycles.	Increasing the width allows more options for allocating multiple operations to one stage, but also provides the potential for more wasted clock cycles.
Decreasing pipeline length	*Possibly detrimental:* Decreased length implies increased width, resulting in lower throughput.	*Beneficial:* Fewer storage registers will be required.	A minimum of 3 stages are required: 2 for memory access and 1 for computation.	As additional stages are added, the area overhead consumed by storage registers becomes increasingly more costly.
Increased parallelism	*Beneficial:* Increased parallelism allows pipeline width to decrease, improving throughput.	*Detrimental:* Generally, increased parallelism requires greater area.	The LEs and multipliers available limit the amount of parallelism possible.	Parallelism reduces latency of operations, allowing multiple operations per stage, and possibly increasing the number of wasted clock cycles in other stages.

Table 4.1: Implementation tradeoffs for throughput and area

required to complete the operations in a given stage (c_m), as shown in Equation 4.1. It is not expected that the number of wasted clock cycles can be reduced to zero, but reducing this number implies a more efficient pipeline scheme, or in other words, a more efficient balance between pipeline length and width. Wasting clock cycles implies that area usage is not efficient, since the wasted cycles could potentially be used for reducing parallelism in some operations, allowing area usage to decrease.

$$n_m = PW - c_m \tag{4.1}$$

An example of pipeline tradeoffs can be observed by analyzing the transform operation. It is fairly simple to break down the required calculations for each block into a highly parallel sequence of additions, multiplications, and shifts, which lasts 18 clock cycles. Since the pipeline width cannot be any shorter than 64 clock cycles, one approach is to add the thresholding operation to this stage. In fact, since the thresholding is a fairly simple comparison operation, it would be possible to combine not only these two operations but the inverse transform as well, with a resulting $18 + 1 + 18 = 37$ clock cycle latency, which still would allow $64 - 37 = 27$ clock cycles for additional operations or optimization.

In practice however, the above allocation is extremely inefficient, resulting in area requirements far in excess of the available hardware. The difficulty occurs in the area reduction process. To improve the area costs, these three operations (transform, thresholding, inverse transform) should be optimized, which will increase their latency and result in them being implemented in two or three pipeline stages, resulting in the additional area overhead of the pipeline storage registers. Ideally, each pipeline stage should have as few wasted clock cycles as possible, since those clock cycles should theoretically be taken advantage of to improve the area

usage of that stage. However in the development effort it is often unclear as to which optimizations will actually result in area improvements, and extra clock cycles are only helpful up to a certain extent. In the end, trial and error was often used for this kind of design work.

After several design and optimization cycles the final design consisted of a six-stage pipeline [HWB07], shown previously in Figure 4.1. The first stage loads the initial data from the first memory and the sixth stage stores the output data to the second memory. The second and fourth stages perform the matrix multiplications for the DCT and IDCT, respectively, and the intermediate (third) stage performs the scalar matrix multiplications for both transform operations, as well as the thresholding operation. Finally, the fifth stage is responsible for the averaging processes.

4.2 Memory Access

Although the DCT and IDCT operations can be thought of as the most computation-intensive parts of the implementation, memory access is by far the main throughput bottleneck due to its finite bandwidth. The two main considerations in designing the memory access scheme are: the selection of appropriate memories from those available on the DE2 hardware, and the examination of the latency required to transfer the data that is needed.

In analyzing the chosen deblocking algorithm it is apparent that the entire first stage of the pipeline must be designed: there must be some process to fetch data from memory and to manage the order in which blocks are processed and in which shifts occur. Furthermore, in the algorithm it is implied that the entire

original image is accessible at the user's convenience; clearly this is not the case. For example, in the averaging stage the pixels from the original image are used in conjunction with the shifted deblocked pixels.

These considerations are related to the broader issue of memory selection. It is clear that to maximize throughput different physical memory should be used for loading the initial pixels and for storing the processed data. This solution would allow the input and output memories to function simultaneously: one pipeline stage could read in new data in conjunction with another stage providing output to the other memory, a feat that would be impossible if only one (single-port) memory was being used.

4.2.1 Memory Selection

The DE2 board has four memory options: FLASH, SRAM, SDRAM, and an SD card slot (as presented previously in Table 2.4). The SRAM was obvious as an ideal choice due to its simplicity in interfacing and timing, its acceptable capacity (512 kB), and its reasonable throughput (16 bits per clock cycle). It was chosen as the output memory for the system. The choices for input memory were less appealing. Because of its simplicity, the FLASH was used for the initial design but was unacceptable in the long term due to the very low throughput of 8 bits every 5 clock cycles. This type of memory also has a limited lifespan, only being certified for 100 000 program/erase cycles [Fuj03], which makes it undesirable for this type of usage. Of the remaining two options, their complexity is comparable but the SD card slot is slower and requires the added purchase of an SD card; thus the SDRAM was chosen.

It should be mentioned that there is indeed one further option for memory implementation: the on-chip M4K memory blocks. However the maximum storage available by this means is only 52.5 kB. Although some of this space is useful for other areas of the design such as read-only memory (ROM) lookup tables, the images being processed are 307 200 bytes, and as such this option was discarded.

4.2.2 Maximum Throughput

Once the most appropriate memories for this project have been selected, an upper-bound on the throughput of the entire system can be calculated. This bound will not necessarily be achievable by the implementation, but can be used as a target, as well as being used to demonstrate the suitability of the given hardware for real-time usage (for this algorithm). The calculations are done for both input and output, and the worst of these values will be the fastest theoretically achievable throughput of the design.

First, the input memory will be considered. Being an SDRAM, this calculation has several components. The basic case itself is fairly straight-forward: to read an 8 byte block (i.e. one row of pixels from an 8×8 pixel block), the memory row must be selected, 4 words (of 2 bytes each) are read, and the row is precharged. This process takes a minimum of 8 clock cycles. However, this case is the one most seldomly used. Since most reads are for shifted blocks of pixels, and since all of the shift values are odd, the bytes to be read overlap with the word boundaries and therefore 5 words must be read instead of 4. This second case requires 9 clock cycles.

The shift patterns cause another complication. Not only are the word bound-

aries compromised but the row boundaries are as well. When those cases occur, 5 words must still be read but the row selection and precharge operations must each be done twice. The overhead latency in the read operation itself also increases, since this pipelined operation is now done twice. This third case requires 13 clock cycles. The three SDRAM accesses are summarized in Table 4.2.

Case	**Row Selection**	**Data Transfer**	**Pre-charge**	**Total**	**Occurrences**
Basic	2	5	1	**8**	All non-shifted rows
Shifted	2	6	1	**9**	63 of every 64 shifted rows
Boundary	4	7	2	**13**	1 of every 64 shifted rows

Table 4.2: SDRAM memory request latencies (durations in clock cycles)

To calculate a final throughput value, these three cases must be combined. The first case, requiring 8 clock cycles, occurs for all loads of the original (non-shifted) image. Since this operation loads 8 pixels, the cost is 1 clock cycle per pixel, resulting in 307 200 clock cycles for the entire 640×480 image.

The second and third cases both occur when loading the four shifted images. Since each read operation fetches 8 pixels and since a row in the SDRAM contains 512 bytes (pixels), 1 out of every 64 reads will occur at a row boundary, comprising the third read case. The other 63 reads in that memory row constitute the second case. For one full shifted image, the total duration is 348 000 clock cycles (t_{CC}), or 1.13 per pixel ($t_{\mathrm{CC/P}}$).

$$\begin{aligned} t_{\mathrm{CC}} &= (1\,(13) + 63\,(9)) \times \tfrac{640\times480}{64} = 348\,000 \\ t_{\mathrm{CC/P}} &= \tfrac{t_{\mathrm{CC}}}{640\times480} = \tfrac{348\,000}{307\,200} = 1.13 \end{aligned} \tag{4.2}$$

An assumption can be made that the pipeline width should not be fixed. Since

read operations (for one row) vary between 8 and 13 clock cycles, the memory access time for one pixel block will vary between $(8\,(8)) = 64$ and $(6\,(9) + 2\,(13)) = 80$ clock cycles. If the pipeline scheme is event-driven as opposed to having fixed timing, the pipeline width is not limited to the worst case, in this case 80 clock cycles. The pipeline width can instead be determined by the memory access times for that specific block. The other (internal) pipeline stages must then conform to a maximum of 64 clock cycles, to keep up with the best-case memory time. It should be noted that this best-case requirement is only necessary for the stages that are actually used in those cases: since the best-case time calculated here is only relevant to non-shifted blocks (which are not subject to the transform operations), stages 2, 3, and 4 are not subject to that specific best-case value.

Second, the output memory is considered. The SRAM is considerably simpler than the SDRAM, but as it is also comprised of 2 byte words it again suffers from the word boundary issue. The initial image is easy to process, requiring 4 clock cycles to write a pixel block row of 8 pixels. For the 307 200 pixels needed the duration is $307\,200/2 = 153\,600$ clock cycles. The latency for the shifted images must be calculated as well; they take longer for two reasons.

The first issue is again the overlap in word boundaries. Each shifted row must access 5 memory words instead of 4. The second issue is that the output data is cumulative: new results are summed with the values from earlier iterations. Each word must first be read, then the sum of the existing word and the new result is written back to the same location. The fastest this process can be accomplished is 2 clock cycles per word. When saving pixel data for the shifted images, each pixel block row of 8 pixels requires $2 \times 5 = 10$ clock cycles. For one full shifted image, the total duration is 384 000 clock cycles, or 1.25 per pixel.

$$\begin{aligned} t_{\mathrm{CC}} &= (2\,(5)) \times \tfrac{640\times480}{8} = 348\,000 \\ t_{\mathrm{CC/P}} &= \tfrac{t_{\mathrm{CC}}}{640\times480} = \tfrac{384\,000}{307\,200} = 1.25 \end{aligned} \tag{4.3}$$

Now the total throughput of the system can be calculated from the worst-case memory latency of each block (see Table 4.3). Considering only the non-shifted image, it is clear that the input memory is slower and thus constitutes the maximum latency of 64 clock cycles per block. For the shifted images, the input memory requires either $(8\,(9)) = 72$ or $(6\,(9) + 2\,(13)) = 80$ clock cycles, depending on the block location, while the output memory consistently requires $(8(10)) = 80$. Therefore for these cases the maximum duration is 80 clock cycles. Considering the full algorithm which processes 4800 blocks for a 640×480 pixel image, the total duration is 1 843 200 clock cycles (see Equation 4.4).

Block	**Memory Latency**		
	Input	**Output**	**Minimum**
Non-Shifted Image Block	64	32	64
Shifted Image Block	72 and 80	80	80

Table 4.3: Minimum memory latency (in clock cycles)

$$\begin{aligned} t_{\mathrm{CC}} &= 1\left(\tfrac{640\times480}{64}\,(64)\right) + 4\left(\tfrac{640\times480}{64}\,(80)\right) \\ t_{\mathrm{CC}} &= 1\,(4\,800\,(64)) + 4\,(4\,800\,(80)) \\ t_{\mathrm{CC}} &= 1\,843\,200 \end{aligned} \tag{4.4}$$

These calculations serve two purposes. First they provide a concrete goal for the pipeline implementation: the stages which process the unshifted image should be no wider than 64 clock cycles, and those which process the shifted images should be no wider than 80 clock cycles. Second they provide a concrete upper-bound on

the throughput of this algorithm on this hardware. With the given on-board clock speed of 50 MHz the full algorithm can be completed in 36.864 ms. Viewing this result in terms of frequency, 27.1 images can be processed per second. Since for real-time video processing a rate of 24 images per second is desired, it is now shown that for this specific hardware, algorithm, and image size, real-time deblocking is theoretically achievable.

$$\begin{aligned} T &= \tfrac{1\,843\,200}{50\times10^6} = 36.864\,\text{ms} \\ f &= \tfrac{50\times10^6}{1\,843\,200} = 27.1\,\text{Hz} \end{aligned} \tag{4.5}$$

4.2.3 Memory Implementation

The implementation of the SRAM interface is fairly straight-forward and will not be discussed. The SDRAM on the other hand is a more complex memory and deserves some explanation.

Unlike most other memories, which can be accessed in one clock cycle simply by putting appropriate control values on the few control pins (such as read/write and chip enable), SDRAM operates an internal finite state machine (FSM) which controls its behaviour. This FSM can be seen in Figure 4.2. The device accessing this memory must understand the FSM, and keep track of the SDRAM's state in order to issue relevant commands. However, the SDRAM has no output pins other than the bidirectional data bus; therefore verification of the current state is a painstaking process.

Without the ability to perform this kind of verification and debugging, the memory's documentation is indispensable in building a functioning interface. Although the documentation for this component was incomplete, a module was eventually

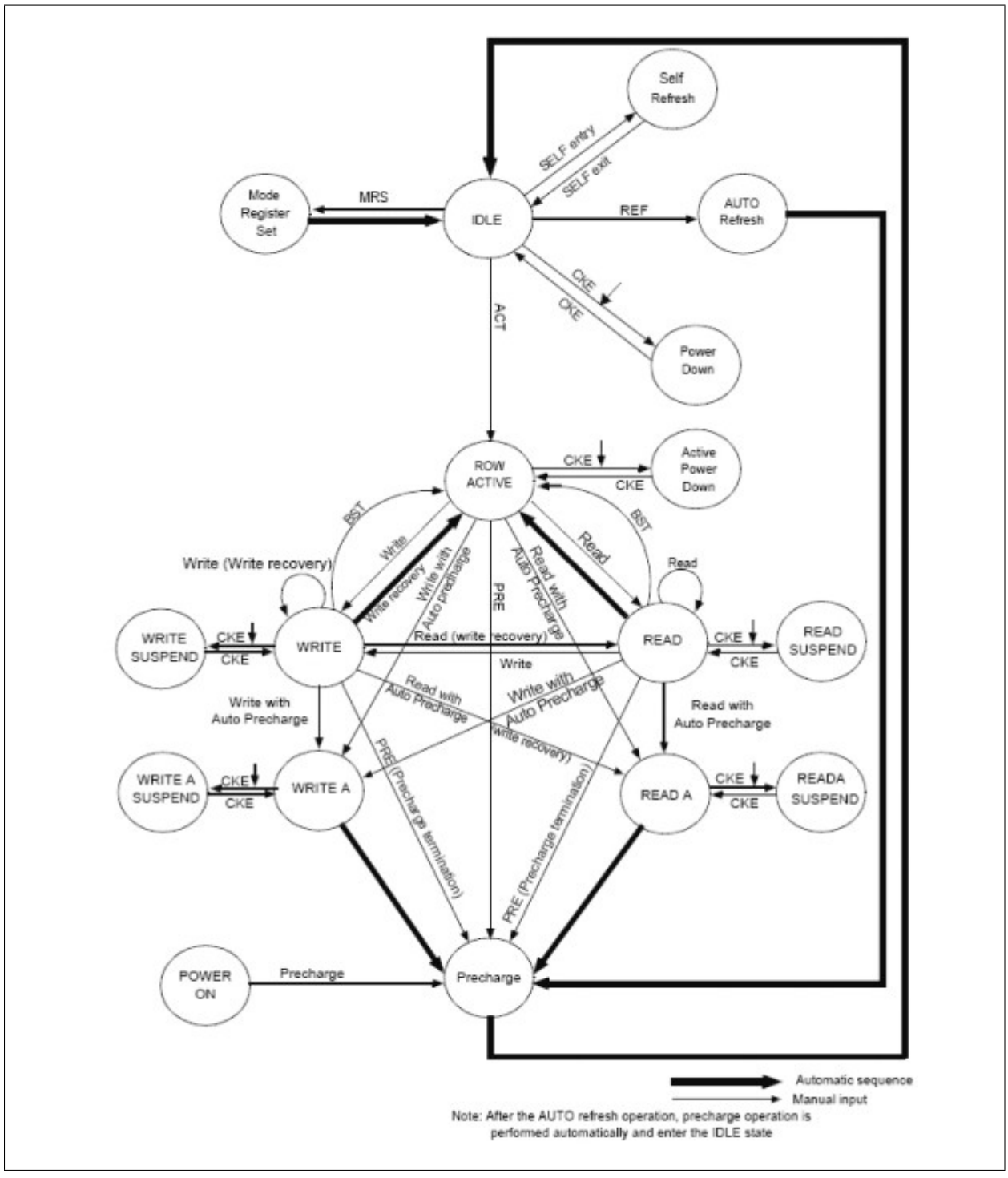

Figure 4.2: SDRAM finite state machine [Int00]

constructed with a simplified state machine, to match the timing required by the memory. The power-up time required was implemented using a simple timer, and after issuing other set-up commands (precharge, refresh, and mode set-up), the interface would wait in the idle state, as would, presumably, the memory. Upon receiving a read or write command, the appropriate row would be activated, the read or write operation performed, and then the required precharge would be requested before returning to the idle state.

Due to the complexity of the SDRAM, several non-functional cases were difficult to repair. The first case was the usage of batch mode. In the initial mode set-up, this SDRAM allows selection of a batch size; that is, selection of how many words to read or write in one operation. Despite this functionality, only the one-word size was made to operate. Therefore, to access the required batch of data, several read or write operations had to be issued sequentially within one request, by incrementing the memory address. This functionality for pipelined accesses was supported and did indeed work. Later in the development process it was realized that this approach would have been required regardless, due to the need for fetching five words at a time, a batch size which is not provided.

The second case which caused difficulty was the transition from one row to another. In all cases performing a read/write cycle on a different row than the previous cycle would result in incorrect behaviour. Since no two consecutive read instructions in this application are on the same memory row (since each row in a given pixel block is adjacent in vertical space but not in memory), this issue was a major problem. Initially every first operation on a new row was simply redone, which would then give the correct result, but which would result in doubling the latency of the first stage of the deblocker's pipeline.

Further research revealed that the clock for the SDRAM was needed to lead the clock for the rest of the design by 3 ns. To accomplish this offset one of the on-board Phase-Locked Loops (PLLs) was used. Not only did the clock have to lead, but the rest of the interface signals did as well. Once these changes were made, the SDRAM behaved as expected.

Although the SDRAM does provide high capacity (8 MByte), the interface consumes additional development time as well as (more importantly) area on the FPGA. It would be preferable for a second SRAM to be used instead, which, similarly to the output memory, would provide acceptable capacity (512 kByte or more) while retaining a great deal more simplicity in the interface.

4.3 Discrete Cosine Transform

The DCT and IDCT operations contain the bulk of the computations required for the algorithm. Due to the similarity between these operations, only the DCT is discussed, with the understanding that the IDCT can undergo the same analysis. Any relevant differences are pointed out as they are encountered.

4.3.1 Matrix Multiplications

The main issue to be addressed for the DCT (see Equation 4.6) is the exorbitant area usage that occurs when it is implemented without any modification from its description in the algorithm. There are several ways in which this operation can be optimized, the first one being the modification to the values of 206 and -206 in the transform matrix P. As all the other values are multiples of 2, these unusual values

were rounded up to 256 and -256 respectively to make them consistent with the rest of the matrix, after consultation with Wong. This change allows all multiplications within the 2 matrix multiplications to be implemented as left bit shifts.

$$\mathbf{Y} = \left(\mathbf{PXP^T}\right) \otimes \left(\mathbf{E} \otimes \mathbf{E^T}\right) >> (2\log_2 \alpha) \tag{4.6}$$

Once all multiplicands are multiples of 2, calculating one data value after one matrix operation becomes the sum of 8 shifted input data values. If all 64 data values are calculated in parallel, one full matrix multiplication can be completed in 8 clock cycles. However, this implementation does not allow all four matrix multiplications (2 each for the DCT and IDCT) to fit in the FPGA's available area.

Thus, some algebraic manipulation is used to present the multiplications in a way that consumes less area. Realizing that if the two operations can be manipulated into the same form then only one implementation is needed and the area will greatly decrease, the property of $(\mathbf{AB}) = \left(\mathbf{B^T A^T}\right)^{\mathbf{T}}$ was used to make the modifications shown in Equation 4.8.

$$\begin{aligned} \mathbf{PXP^T} &= (\mathbf{PX})\mathbf{P^T} \\ &= (\mathbf{X^T P^T})^{\mathbf{T}}\mathbf{P^T} \\ &= \Phi\left[\Phi\left[\mathbf{X}\right]\right] \qquad (4.7) \\ \mathbf{\Phi}\left[\mathbf{X}\right] &\equiv \mathbf{X^T P^T} \qquad (4.8) \end{aligned}$$

Now instead of computing two separate operations, the input pixels can be sent

into the Φ computation unit (a transposition followed by a multiplication), and the result of that iteration can simply be sent into the same computation unit again for the second iteration. Although the routing for this new scheme is possibly more complex than for the previous, the great reduction in area is still a worthwhile justification. The improvement is further justified by noticing that the matrix transposition is a trivial operation in hardware.

Further area reductions are possible by optimizing routing. A noticeable contribution to the routing area comes from the transition from the pixel block matrix to the computation logic. This area can be reduced by noticing that for every data value being calculated in the multiplication, all of their nth terms are in the same row of the input matrix. Originally, every data value in the input matrix was routed to the computation unit. If only one row is required at a time however, then the entire pixel storage matrix can be implemented as a large rotation unit, with only one row being routed to the logic. VHDL pseudocode of this optimization can be seen in Figure 4.3.

A further improvement can be introduced by taking advantage of the extra clock cycles available in this stage of the pipeline. By treating each $(8 \times 8)(8 \times 8)=(8 \times 8)$ matrix multiplication as two $(8 \times 8)(8 \times 4)=(8 \times 4)$ operations, the existing input structure remains the same but the computation unit's size is reduced, as well as the routing area for the output of the computation unit. This change is further improved by implementing a rotating matrix on the output as well, allowing the output to be routed to 4 consistent columns and having the previous result rotated over to the other 4 columns when new results are available. Although this overall improvement is not able to halve the area of the computation unit (as additional logic is now required in order to change multiplicands between the two operations),

```
-- pixel block rotation implementation

if (loopflag='1') then
      for i in 0 to 7 loop
            for j in 0 to 6 loop
                  PixelMatrix(i, j) <= PixelMatrix(i, j+1);
            end loop;
            PixelMatrix(i, 7) <= PixelMatrix(i, 0);
      end loop;
end if;

-- routing to computation logic

for k in 0 to 7 loop
      LogicInputBus(k) <= PixelMatrix(k, 0);
end loop;
```

Figure 4.3: VHDL pseudocode of rotation functionality

there is still a noticeable improvement.

4.3.2 Scalar Multiplication

There is little improvement to be had in the scalar multiplication for the DCT. To increase efficiency, the optimized on-chip multiplication units are used. Customized multiplication functions are created in the Quartus II hardware compiler with one fixed multiplicand and one input multiplicand, allowing the compiler to take care of the optimization itself. One multiplication function is created for each of the data values requiring one, for a total of 28, while the data values to be multiplied by one are simply passed on without modification (see Equation 3.5).

One design improvement can be noted. Since the scalar multiplications for the DCT and IDCT both occur in the same pipeline stage and each contain the same multiplication matrix ($\mathbf{E} \otimes \mathbf{E}$), the hardware for these two operations can be

reused, including all of the multipliers. This improvement is a major justification for placing these two scalar multiplications within the same stage.

4.4 Additional Optimizations

In addition to the major components of the design (DCT, memory interfaces, etc), many other optimizations are made throughout to reduce the physical area while remaining within pipeline width (latency) limitations. These changes include utilization of the on-chip M4K memory blocks and deparallelization of area-intensive operations.

4.4.1 Thresholding

In software the thresholding operation is most simply described as a loop comparing the data values against a set of threshold values, and changing the data values if necessary. In hardware this operation can be described simply in two different ways: either by using the same approach, or viewing it as an operation done in parallel, simultaneously operating on all 64 matrix values. The former method is more efficient in terms of area; the latter in terms of latency. Since the longer latency is not long enough to affect the overall system throughput, and since area is an important concern, the software-centric approach was taken.

Area usage can be improved beyond this initial selection. The first optimization is to remove the threshold matrix from logic element storage and instead store it as a ROM lookup table in unused M4K memory space. Although some logic is then required to manage the ROM control signals, this area can be considered minor

in comparison to the logic elements required to store 64 9-bit values. Also, to improve routing, the implementation uses a rotation mechanism for the data block, similar to that used for the DCT. Since only one threshold value can be fetched from the ROM at a time, only one frequency component can be thresholded at a time. Therefore, routing to the required operation units can be greatly simplified so that only one cell in the data block is routed, and the rest of the cells rotate so that all values eventually pass through the one chosen cell. In comparison to the previous implementation there is a tradeoff in where the area is consumed. As seen in Table 4.4 there is indeed an improvement as expected.

Method	Description	Area	Memory	LE Usage
1	Initial method	3 715 LEs	0 bits	n/a
2	As 1, with ROM lookup table	2 613 LEs	1 984 bits	70%
3	As 2, with rotation mechanism	2 276 LEs	1 984 bits	61%

Table 4.4: Comparison of threshold matrix implementations

4.4.2 Averaging

The averaging to be done is a good demonstration of operations that are simple in software but introduce additional complexity in hardware design. On the surface these operations are quite simple: the pixels are weighted based on their distance from the pixel block centre, a weight which can be pre-computed and stored on-chip. However in practice there are several challenges.

The first challenge is to simplify the multiplication operations. One reason for this simplification is that by this stage in the design all of the multiplication units on the chip have been allocated to the scalar matrix multiplication in stage 3 of the pipeline. Another is that a multiplier consumes logic elements, as opposed to alter-

nate implementations which could be designed to consume otherwise unused M4K memory blocks. Implementing the multiplication in memory makes additional sense when considering that M4Ks would likely be used regardless for implementing the required weight matrix, as done for the comparison values used in the thresholding operation.

To simplify the averaging as much as possible, the entire operation is done with a read of an 8-bit value from an M4K ROM lookup table, an improvement over reading to only obtain the multiplicand. This one read operation replaces three operations: the selection of the appropriate multiplicand based on the pixel's location, the multiplication itself, and the ensuing division (right bit shift) to normalize the value. In the algorithm, the multiplicand is an 8-bit value and the division is a shift right by 8 bits. In the implementation the accuracy of these operations is much less, since the accuracy is limited by the 8-bit representation of the final value itself.

Using a lookup table becomes even more suitable when noticing that there are exactly 16 different multiplicands, if the multiplicands of 0 and 256 (out of a maximum of 256) are not included. Therefore the table can be addressed using 12 bits: the multiplicand's identifier being the most significant 4 and the pixel value being the remaining 8. At this address in the lookup table is located the 8-bit result of the pixel value (address bits 7 down to 0) being scaled by the appropriate amount (identified by address bits 11 down to 8).

The second challenge is to address the issue of shifting. The problem here is that most of the pixel blocks being processed are offset; however the pixel weights are a function of the pixel's distance from the center of a non-shifted block. That is, each pixel's weight depends on its location within the original image, not its

location in the shifted copies. To accommodate this requirement, each pixel block arriving at the averaging stage is itself shifted back into a non-shifted state, and then re-shifted to its appropriate position upon leaving the stage.

The third challenge comes in accommodating an undocumented truncation imposed after the IDCT completes. Because of the inexactness of the DCT and IDCT implementations, result values can often occur that are either less than 0 or greater than 255, which is outside of the 8-bit range for the pixel data. To check for these cases, two flags can be added to each 8-bit pixel value at the start of the averaging stage, resulting in a 10-bit value per pixel. The first bit is the underflow flag, which can be taken directly from the sign bit of the previous stage, since the result of the IDCT is a signed value whereas the pixel value itself is not. The second bit is the overflow flag, which is the disjunction of the bits between the sign bit and the least significant 8 bits (which represent the pixel itself). If the underflow flag is set, the incoming value was negative and should be set to zero. If the overflow flag is set, the incoming value required more than 8 bits and should be lowered to the maximum value of 255. Although adding this functionality does not seem overly complex, in practice the area and routing required is noticeable in the implementation of the full system.

4.5 Summary of Final Design

The final design comprises a six-stage event-driven pipeline with SDRAM input to the first stage and SRAM output from the last. A summary of the final design can be seen in Figure 4.4. Each pipeline stage is composed of one or more state machines which transition out of their idle state upon notification of the previous

state's completion. The DCT and IDCT processes are contained within stages two, three, and four; pixel blocks of the base (non-shifted) image transition directly from stage one to stage five, while shifted blocks progress through all stages of the pipeline.

The total area usage was 31 059 out of 33 216 available logic units. 35 multipliers were also used, as well as 34 244 bits of on-chip M4K memory. A summary of resource utilization can be seen in Table 4.5.

Resource	Available	Used	Utilization
Logic elements	33 216	31 059	93.5%
Multipliers	35	35	100.0%
Memory bits	483 840	34 244	7.1%
Pins	475	79	16.6%
PLLs	4	1	25.0%

Table 4.5: FPGA resource utilization

4.6 Commentary on the Design Process

Even with the apparent suitability of the chosen algorithm for hardware, implementation was still a long and involved process. Implementation of the basic functionalities was not difficult, but as optimization increased the timings and relationships between units became increasingly more complex. The iterative optimization and debugging process was a major contribution to the duration of the project. There are four recommendations that are given based on the work done in this project, for easing the transition from a software or algorithmic design to an efficient hardware product:

1. reduction of operating units;

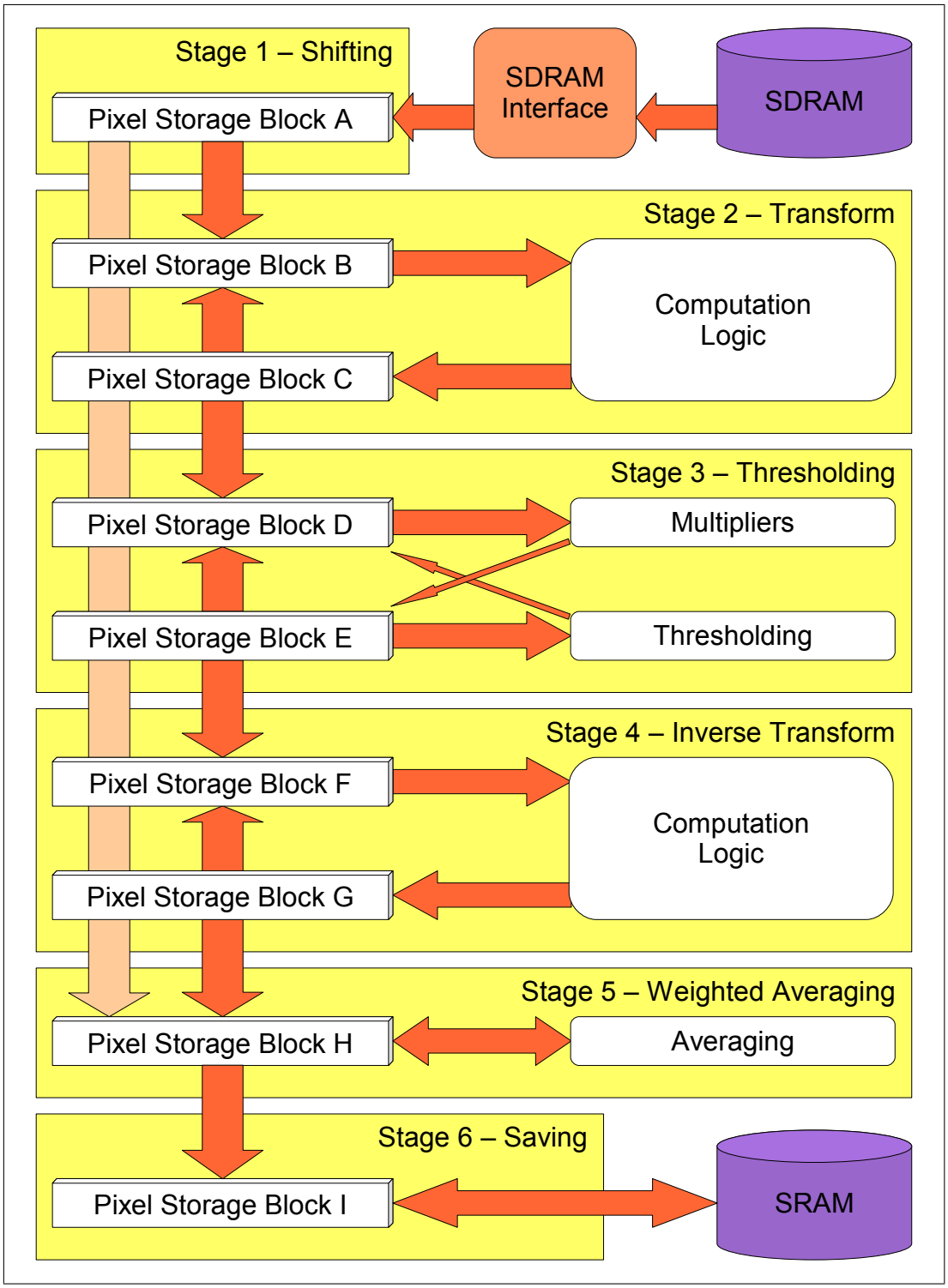

Figure 4.4: Overview of final design

2. reduction of routing requirements;

3. analysis of dataflow bottlenecks; and

4. development of a smaller compilation testbed.

The first recommendations involve the design of the algorithm, or the perception of what constitutes a design that is "simple". Using integer operations in place of floating point, and simplifying multiplications and divisions are helpful improvements, but there are further considerations that can be taken. Two improvements that can be made are to reduce operating units and to reduce routing requirements. Both of these can be addressed by analyzing the dataflow of the algorithm.

In this project, these improvements were made using the rotating matrix approach. Throughout the project, operations on an entire matrix were applied to a minimal number of matrix values (often only one), and the matrix itself was implemented to progressively rotate its values so that, for every clock cycle a new value was present at the operating location. Thus, the hardware units required were simply the matrix storage registers and the minimal number of operating units, and the routing required was from only one (or few) registers to those operating units, as well as the rotation routing. It can be noticed that these routing paths should require a very small number of multiplexors and gates, as well as traveling a very small physical distance, since they only travel between nearby logic elements. In general algorithm design, this approach can be taken by grouping operations in a manner conducive to this style of implementation.

The other recommendations involve the implementation process. First, an initial analysis of the bottlenecks in the system is essential for determining both the feasibility (best-case scenario) of the project, as well as establishing guidelines for

the timing and optimization goals for the system. This analysis can provide invaluable insight into efficient groupings of components and useful areas of optimization, mostly from a timing perspective. Second, the debugging process, the most time-consuming part of the development, can be improved by preparing an independent development system which only operates the module(s) in question rather than the full implementation. Reducing the on-chip area of a program being compiled offers a great reduction in compile time, often eliminating 40 minutes from a one hour compile for this deblocking project. Although much debugging still must be done in the full program, days can still be saved using this method.

Chapter 5

Results

The real-time video deblocking algorithm was successfully implemented on the Altera DE2 board. Further optimization is needed to attain the theoretically achievable 24 frames per second data rate required for real-time applications. The final results of the hardware implementation are shown in Table 5.1.

5.1 Speed

Achieving high throughput is an important goal for the system. Unfortunately, the current implementation falls short of the 24 frames per second that were shown to be attainable in Chapter 4. However, it is likely that a rate of 24 frames per second can be reached with some minor optimizations.

The current throughput is measured experimentally as 21.28 fps with a 50 MHz clock speed. The compilation process reports a maximum clock frequency of 63.7 MHz. Without any further changes to the design, a rate of 27.1 fps can be reached simply

Parameter	Result	Unit	Notes
FPGA logic elements used	31 059	elements	93.5% utilization
FPGA memory utilization	34 244	bits	7.1% utilization
FPGA multipliers used	35	units	100% utilization
Minimum FPGA pins required	79	pins	16.6% utilization
FPGA PLLs used	1	units	25% utilization
Off-chip memory utilization	614 400	bytes	$2 \times$ image size
Pipeline stages	6	stages	
Pipeline width	98 to 121	clock cycles	Variable (event driven)
Datapath width	8 to 31	bits	Variable (stage dependent)
Datapath storage	12 864	bits	Data registers for pipeline stages
Theoretical throughput	0.129	pixels per clock cycle	Average, due to variable pipeline width
Clock speed	50.0	MHz	On-board clock
Maximum possible frequency ($f_{\max}$)	63.7	MHz	Compilation result
Total theoretical runtime	47.59	ms	Calculated based on theoretical throughput
Total experimental runtime	46.99	ms	Obtained using a hardware clock cycle counter
Final deblocking rate	21.28	frames per second	Based on experimental runtime
Theoretical maximum rate	27.1	frames per second	Based on $f_{\max}$ clock speed

Table 5.1: Summary of final design characteristics and performance

by increasing the clock speed.

There are several optimizations possible to increase throughput. The current bottleneck is, as expected, the interface to the SDRAM in stage one of the pipeline. For design simplicity, this interface is contained in a separate module and transfers data to the main design through a request and acknowledge process. This module can be optimized by integrating it into the main design, reducing some of the delay in each memory access. For simplicity, the SDRAM interface always fetches blocks of 5 words, as required by the worst case scenario of the shifted images where pixels transgress the word boundaries in memory. For the sake of optimization, a 4-word mode could be implemented, shortening the access time for the non-shifted image.

5.2 Area

The physical area required by this system is both a major consideration and a low priority item. For the system to be successfully implemented at all, it must fit within the 33 216 logic elements available on the Altera Cyclone II FPGA. Therefore, area optimization is of major importance. However, additional area optimization beyond this point is not an important concern. It can be noted that designs with smaller area require less compilation time, which can be advantageous to the development process.

As shown in Table 5.1 the design consumes 31 059 logic elements, or 93.5% of those available. Of the other on-chip resources, the only ones heavily used are the multiplication units, all of which are consumed in stage three of the pipeline for executing the scalar matrix multiplication with the matrix ($\mathbf{E} \otimes \mathbf{E^T}$). One PLL is also used, as required for SDRAM interfacing, as well as 16.6% of the pins: 39

for the SRAM, 38 for the SDRAM, 1 for the clock, and 1 for a pushbutton which acts as an asynchronous reset signal. Finally, 7.1% of the on-chip memory is used to implement the thresholding lookup table in stage three and the pixel averaging lookup table in stage five.

It is also useful to note that pipeline storage registers for the pixel data account for 12 864 bits, or 41.4% of the logic elements used, which is, as expected, a large proportion of the area costs.

5.3 Quality of Deblocking

The deblocking algorithm produces a very noticeable improvement to the artifacts in the original image. Examples of input (original) images and output (deblocked) images are shown in Figure 5.1 and Figure 5.2.

5.4 Future Work

This project represents an excellent prototype for real-time video hardware deblocking, and has excellent potential for expansion. Some possible avenues of work are:

1. optimization and modification of the design;
2. modification to the input and output mechanisms;
3. implementation as a real-time device; and

(Original image)

(Deblocked image)

Figure 5.1: First sample result of hardware deblocking (shown at 75%)

(Original image)

(Deblocked image)

Figure 5.2: Second sample result of hardware deblocking (shown at 75%)

4. extension to colour deblocking and higher resolutions such as HDTV (720×480).

As noted in Section 5.1 several optimizations can be implemented to improve the throughput of the design. There is also the possibility of modifying the design to correct the unwanted darkness that occurs at the outer edges of the image. A simple solution is to simply ignore the outer edge of blocks over the entire image, retaining them without any deblocking. This solution could result in some small discontinuity at the edges, but it is estimated that it would still be an improvement, as well as improving the throughput of the system by effectively reducing the area to be deblocked from 640×480 to 624×464.

The input and output of the design can be modified for several reasons. One noticeable benefit would be the removal of the SDRAM, simplifying the first stage of the pipeline. This memory could be replaced with a SRAM, allowing the first stage, like the sixth stage, to adopt a minimalistic memory access scheme. The existing single-port SRAM, accessed by the sixth stage, could be replaced with a dual-port memory to streamline the required process of reading and writing data.

Beyond simplifying the two pipeline stages which interface the memories, input and output modifications can be implemented to realize the initial goal of deblocking video in real-time. The ethernet, USB, and VGA components on the DE2 board, as well as the two parallel ports, could be used to receive and transmit video frames from another device, allowing the system to operate in real-time as intended.

Finally, there is the potential to extend the design from greyscale deblocking to colour deblocking. The core implementation can be replicated or expanded to accommodate the larger data representations of colour pixels. Furthermore, different resolutions can easily be accommodated by simply changing the global image

size constants which are currently set at 640 and 480. These changes have a direct effect on the throughput of the system, and as such increasing the workload will require research into larger FPGAs (for replication) or those with faster attainable clock speeds.

Chapter 6

Conclusions

Video and image data is heavily used in both professional and non-professional contexts. Compression, primarily block-based lossy compression, is often used on this data to reduce storage and transmission costs. Block-based compression can result in undesirable blocking artifacts, which degrade the quality of the media. These artifacts can be addressed by means of deblocking techniques, which can be costly to implement for real-time video deblocking in particular due to the amount of computation required.

This dissertation presents and analyzes a novel implementation of real-time video deblocking on FPGA hardware. The implementation uses an Altera DE2 development board containing an Altera Cyclone II FPGA. The system meets all of the project requirements with the exception of a throughput of 24 frames per second, which can be attained either through further optimization or through the use of a faster external clock. Specifically, the project requirements include achieving throughput suitable for real-time video deblocking, reducing area as needed for implementation in the given FPGA, and producing a noticeable reduction in the

blocking artifacts present in the input frame of video.

6.1 Thesis Contributions

The hardware video deblocker described in this dissertation has many applications in consumer electronics. Mobile devices as well as digital home entertainment devices are well-suited for adoption of this system. The following contributions are made by this thesis research:

1. selection of a deblocking algorithm that is suitable for implementation in low-cost hardware;
2. description of the modifications necessary to the chosen deblocking algorithm for the purpose of hardware implementation;
3. design and implementation of the algorithm on a low-cost FPGA;
4. description of the optimizations necessary to achieve the area and throughput required for real-time applications;
5. analysis of the feasibility of the given implementation for real-time video deblocking, including the feasibility for usage in high-definition television;
6. presentation of general techniques which can be used for future hardware projects of a similar nature; and
7. presentation of enhancements and expansions which can be implemented in the future.

The shifted transform deblocking algorithm proposed by Wong provides many benefits such as reduced computation: fewer multiplications and simplification of divisions to bit-shifting. It also can be easily partitioned into sub-tasks which is ideal for pipelining. Wong's algorithm can be further optimized using matrix algebra. It is then partitioned into pipeline stages to increase throughput. Operations are grouped logically in an attempt to reduce area usage, such as by reusing the multiplication units in stage three. The on-board SDRAM and SRAM are selected as the input and output memories, respectively. Several techniques are used for optimization of the algorithm within the area constraints of the FPGA and the throughput constraints of the project, such as:

1. variable pipeline width (number of clock cycles per stage) and variable pipeline datapath width;
2. reuse of operational units such as the DCT unit in stage two and the multipliers in stage three;
3. partitioning of the DCT and IDCT operations to reduce wasted clock cycles in stages two and four;
4. use of on-chip memory for ROM lookup tables in stages three and five; and
5. implementation of rotation mechanisms for data storage elements within most stages.

The resulting system delivers a throughput of 21.28 frames per second. Enhancements are described for improvement of this value to the desired rate of 24 fps, including optimizations to the SDRAM interface in the first stage of the pipeline.

The deblocked images exhibit remarkably fewer noticeable blocking artifacts than the originals.

From the work on this project several techniques are suggested to improve the transition from an algorithm to a full hardware implementation. For example, optimization of operating units and routing requirements should be considered, and both throughput analysis (for bottlenecks) and smaller compilation tasks can be observed to streamline the development process.

Finally, there are opportunities for future work on this project. Further development can be done to process video in real-time, as opposed to the current single-frame operating mode. Changes to the input and output mechanisms can be done towards this goal, as well as to simplify the memory interface in stage one. The current design can easily be extended to operate on higher resolutions (such as 720×480) and colour media.

6.2 Thesis Applicability

The proposed real-time hardware video deblocking architecture is suitable for implementation in many consumer devices. Mobile devices, having constraints on data storage and transmission, are an ideal candidate for the use of the hardware deblocking architecture. Digital home entertainment is another important venue: real-time deblocking is used to cater to customer demand for high visual quality. Deblocking allows transmission costs to be reduced, which allows bandwidth to be used for other applications such as enhanced digital television content or higher resolution television.

The proposed implementation is suitable for low-cost hardware, which is con-

venient for use in consumer electronics. It allows a reduction in media file sizes through block-based compression, while limiting visual quality degradation, providing to the end user a good balance between file size and media quality.

Bibliography

[Alt06] Altera Corporation. *DE2 User Manual, Version 1.4*, 2006. `http://www.altera.com/education/univ/materials/boards/DE2_UserManual.pdf`.

[Alt07] Dave Altavilla. ATi Radeon X800 XT & X800 Pro. World Wide Web Document, May 2007. `http://www.hothardware.com/viewarticle.aspx?articleid=517`.

[AM02] Ian Alston and Bob Madahar. From C to netlists: Hardware engineering for software engineers? *IEEE Electronics & Communication Engineering Journal*, 14(4):165–173, August 2002.

[CAGM94] Navin Chaddha, Avneesh Agrawal, Anoop Gupta, and Teresa H.Y. Meng. Variable compression using JPEG. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 562–569, May 1994.

[CCR98] Jim Chou, Matthew Crouse, and Kannan Ramchandran. A simple algorithm for removing blocking artifacts in block-transform coded images. *IEEE Signal Processing Letters*, 5(2):33–35, 1998.

[CKL06] Jian-Wen Chen, Chao-Yang Kao, and Youn-Long Lin. Introduction to H.264 advanced video coding. In *Proceedings of the 2006 Conference on Asia South Pacific Design Automation*, pages 736–741, January 2006.

[EPG+05] Mohammed Elbadri, Raymond Peterkin, Voicu Groza, Dan Ionescu, and Abdulmotaleb El Saddik. Hardware support of JPEG. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pages 812–815, May 2005.

[Fuj03] Fujitsu Limited, Tokyo, Japan. *SPANSION Flash Memory Data Sheet*, 2003. `http://www.spansion.com/datasheets/e520888.pdf`.

[GES+99] Steven Gringeri, Roman Egorov, Khaled Shuaib, Arianne Lewis, and Bert Basch. Robust compression and transmission of MPEG-4 video. In *Proceedings of the Seventh ACM International Conference on Multimedia (Part 1)*, pages 113–120, October 1999.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[HCS95] Sung-Wai Hong, Yuk-Hee Chan, and Wan-Chi Siu. Subband adaptive regularization method for removing blocking artifacts. *IEEE Proceedings on ICIP*, 2:523–527, 1995.

[HLC+05] Pei-Yung Hsiao, Le-Tien Li, Chia-Hsiung Chen, Szi-Wen Chen, and Sao-Jie Chen. An FPGA architecture design of parameter-adaptive real-time image processing system for edge detection. In *Proceedings of the Emerging Information Technology Conference*, August 2005.

[HWB07] Martin Hansen, Alexander Wong, and William Bishop. A hardware implementation of real-time video deblocking using shifted thresholding. In *Proceedings of the Twentieth Canadian Conference on Electrical and Computer Engineering*, April 2007.

[Int00] Integrated Circuit Solution Inc., Hsin-Chu, Taiwan. *2(1)M Words x 8(16) Bits x 4 Banks (64-Mbit) SYNCHRONOUS DYNAMIC RAM Data Sheet*, 2000. `http://www.icsi.com.tw/english/`.

[ISO94] ISO/IEC. Generic coding of moving pictures and associated audio information, part 2: Video, 13818-2, Nov 1994.

[Kim00] Daijin Kim. An implementation of fuzzy logic controller on the reconfigurable FPGA system. *IEEE Transactions on Industrial Electronics*, 47(3):703–715, June 2000.

[Nel00] Anthony Edward Nelson. Implementation of image processing algorithms on FPGA hardware. Master's thesis, Vanderbilt University, Nashville, TN, 2000.

[Nos01] Aria Nosratinia. Enhancement of JPEG-compressed images by reapplication of JPEG. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 27:69–79, 2001.

[OS95] T. P. O'Rourke and R. L. Stevenson. Improved image decompression for reduced transform coding artifacts. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(8):298–304, 1995.

[PH06] Mustafa Parlak and Ilker Hamzaoglu. An efficient hardware architecture for H.264 adaptive deblocking filter. In *First NASA/ESA Conference on Adaptive Hardware and Systems*, pages 381–385, 2006.

[SCL06] Shen-Yu Shih, Cheng-Ru Chang, and Youn-Long Lin. A near optimal deblocking filter for h.264 advanced video coding. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 170–175, New York, NY, USA, 2006. ACM Press.

[SVMJ95] Brian Schoner, John Villasenor, Steve Molloy, and Rajeev Jain. Techniques for FPGA implementation of video compression systems. In *Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays FPGA '95*, pages 154–159, February 1995.

[SZZ04] Miao Sima, Yuanhua Zhou, and Wei Zhang. An efficient architecture for adaptive deblocking filter of H.264/AVC video coding. *IEEE Transactions on Consumer Electronics*, 50(1):292–296, February 2004.

[THAE00] Cesar Torres-Huitzil and Miguel Arias-Estrada. An FPGA architecture for high speed edge and corner detection. *Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, pages 112–116, 2000.

[VIA07] VIA Technologies, Inc. World Wide Web Document, May 2007. `http://www.via.com.tw/en/products/chipsets/p4-series/pn800/`.

[Wal91] Gregory K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–34, 1991.

[WB06] Alexander Wong and William Bishop. Efficient deblocking of block-transform compressed images and video using shifted thresholding. In *Proceedings of the Eighth IASTED International Conference on Signal and Image Processing*, August 2006.

[WLY02] Chaminda Weerasinghe, Alan Wee-Chung Liew, and Hong Yan. Artifact reduction in compressed images based on region homogeneity

constraints using the project onto convex sets algorithm. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(10):891–897, 2002.

[XOZ97] Zixiang Xiong, M. T. Orchard, and Ya-Qin Zhang. A deblocking algorithm for JPEG compressed images using overcomplete wavelet representations. *IEEE Transactions on Circuits and Systems for Video Technology*, 7:433–437, 1997.

[ZRG+02] Paul S. Zuchowski, Christopher B. Reynolds, Richard J. Grupp, Shelly G. Davis, Brendan Cremen, and Bill Troxel. A hybrid ASIC and FPGA architecture. In *IEEE/ACM International Conference on Computer Aided Design*, pages 187–194, November 2002.