DESIGN AND IMPLEMENTATION OF AN ON-LINE CFA DEMOSAICKING CORE

by

Gökhan Kabukcu

BSc, in Computer Engineering, Boğaziçi University, 2006

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Systems and Control Engineering
Boğaziçi University

2008

DESIGN AND IMPLEMENTATION OF AN ON-LINE CFA DEMOSAICKING CORE

APPROVED BY:

Assoc. Prof. Arda Yurdakul (Thesis Supervisor)	.....
Prof. Lale Akarun	.....
Assist. Prof. Şenol Mutlu	.....

DATE OF APPROVAL: 14.05.2008

ACKNOWLEDGEMENTS

First of all, I thank my advisor Assoc. Prof. Arda Yurdakul for her support and guidance during my studies. She supported me from the beginning and gave me a huge amount of her time for this thesis.

I would like to thank Prof. Lale Akarun and Assist. Prof. Şenol Mutlu for being a part of the committee. I also thank Dr. Suzan Üsküdarlı who motivated me to start graduate study. They devoted their time and energy to this research. I have special thanks to Prof. Mehmet C. Çamurdan and Dr. Suzan Alptekin for their support and motivation during my student life.

I thank the members of CASLAB and SOSLAB, Salih Bayar, Ahmet Yıldırım, Dağhan Dinç for their help and support during my studies. I also have special thanks to my classmates İsmail Arı, Ahmet Bombacı, Ali Hakan Altınsoy and Murat Eroğlu.

I thank my homemates throughout my student life, Hüseyin Armağan, Tuncay Atar, Ferhat Bahar, Kaan Karabulut and Furkan Armağan for their great support and help. I am grateful to them for having always good time. I also thank my friends Utku Karacaoğlu, Mustafa Yılmaz, Ali Ateş and Hakan Söğüt and Eyyup Şahin who visit us at home without being bored. We will always be waiting for them to show hospitality.

I want to express my gratefulness to my family, especially to my nephews, for their endless love, support and encouragement.

Finally, I thank my other friends Ozan Akar, Selami Çiftçi, Banu İnce, Birim Duru, Kemal Aktay, Cevat Tüzün, Ahmet Baki, Alper Gül and Yasin Yıldırım.

This thesis has been partially supported by The Scientific and Technological Research Council of Turkey (Pr. No: 104E139).

ABSTRACT

DESIGN AND IMPLEMENTATION OF AN ON-LINE CFA DEMOSAICKING CORE

The thesis introduces a low-cost algorithm for improving the demosaicking process in the texture areas such as one-pixel patterns. The algorithm first detects difficult texture regions. After the detection process is completed, the algorithm demosaicks the texture areas using special demosaicking operations whereas non-texture regions are restored using some of the existing demosaicking approaches. In this way, the quality of the texture areas in demosaicked images can be improved up to 70% while the computational complexity of the original demosaicking solution is increased only slightly.

The new algorithm is implemented as a core by using VHDL (Very High Speed Integrated Circuit Hardware Description Language) language. The operational verification of the VHDL implementation is performed on FPGA (Field Programmable Gate Array). The Virtex-II XC2V500 device is selected in the implementation. The core is capable of processing 1000×1000 pixels real-time digital video and $1000 \times n$ pixels digital still images. The system operates at 25 MHz frequency and can process 25 images per second which is a sufficient speed for video processing.

ÖZET

RENK FİLTRESİ DİZİLERİNİN GERÇEK ZAMANLI YONGA ÜSTÜ MOZAİKLEMEME İŞLEMİ İÇİN ÇEKİRDEK TASARIMI

Bu tezde, Bayer Filtresinden yüksek kalitede görüntü elde etmek ve kayıp renklerin hesaplanması işleminin performansını geliştirmek için filtrede bir piksel genişliğe kadar düşen desenlerde eksik renk değerlerini hesaplayan bir algoritma sunulmaktadır. Algoritma öncelikle bir piksele kadar düşebilen ve eksik renk değerlerinin doğru hesaplanması zor olan bölgeleri tespit eder. Problemli ve eksik değerlerin hesaplanmasının zor olduğu bölgelerde, özel bir ara değer hesaplama yöntemi seçilmiştir. Problemsiz ve kenarların az olduğu bölgelerde ise mevcut olan algoritmalar kullanılmaktadır. Bu metotla, eksik değerlerin hesaplanmasının zor olduğu, bölgelerde mevcut algoritmaların hesaplama karmaşıklığı kabul edilebilir bir miktarda aşılarak %70'e varan bir görüntü kalitesi artışı yakalanmıştır.

Geliştirilen algoritma çekirdek olarak VHDL dili kullanılarak gerçeklenmiştir. Algoritmanın Alan Programlamalı Kapı Dizileri (APKD) üzerinde işlevsel doğrulaması yapılmıştır. Gerçekleme esnasında APKD olarak Virtex-II XC2V500 yongası kullanılmıştır. Geliştirilen çekirdek 1000×1000 boyutlarındaki gerçek zamanlı video işleme ve $1000 \times n$ boyutlarındaki durağan görüntü işleme işlemlerini gerçekleştirebilir. Gerçeklenen sistem 25 MHz frekansta çalışmaktadır ve saniyede 25 tane resim işleyebilir. Ulaşılan bu hız ise video işleme için yeterli imkanı sağlamaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

D	Difference of Interpolators
I_H	Horizontal Interpolator
I_V	Vertical Interpolator
K_B	The Difference of Green and Blue Values
K_R	The Difference of Green and Red Values
O_B	Operations Used in Calculation of Blue Values
O_G	Operations Used in Calculation of Green Values
O_R	Operations Used in Calculation of Red Values
δ_H	Horizontal Edge Classifier
δ_V	Vertical Edge Classifier
B	Blue
BW	Big Window
C	Chrominance (Red and Blue)
CFA	Color Filter Array
CIELAB	Commission Internationale d'Eclairage L^*, a^*, b^* Color Space
DCM	Digital Clock Manager
ECI	Effective Color Interpolation
EOL	End of Line
FPGA	Field Programmable Gate Array
G	Green
MAE	Mean Average Error
MSE	Mean Square Error
NG	Neighbor Green
R	Red
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1. INTRODUCTION

Single chip image sensors are widely used in digital imaging devices such as digital still cameras, video camcorders, PC cameras and even cellular phones. In a cost-effective device, a monochrome image sensor is covered with a color filter array (CFA). The CFA consists of filters, each of which allows only one color to be measured at each pixel. The image obtained at the output of a CFA is mosaic-like. Therefore, a process called "demosaicking" is required in order to have a full color image.

The choice of color for each filter and the arrangement of these filters in a CFA differ depending on the camera manufacturers. Mostly, CFAs with three primary colors, namely red (R), green (G) and blue (B) are used since images are stored in RGB color format in computers and the complexity of demosaicking process is comparably low [1]. Several patterns exist for the arrangement of color filters due to different objectives of the manufacturers [2]. The most common CFA is the Bayer array [3], as shown in Figure 1.1. The green filters appear twice more than red and blue ones because the peak sensitivity of human visual system for luminance occurs at around the frequency of green light.

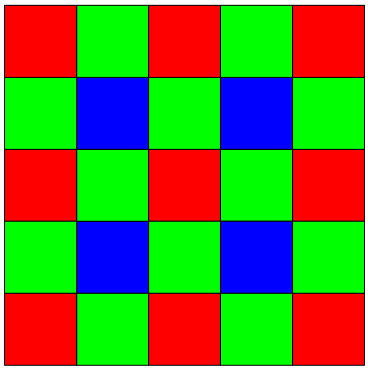

Figure 1.1. A Bayer array pattern

In demosaicking, missing color values of each pixel are estimated. If a mosaicked-image is decomposed into three single-color images, then the missing colors can be estimated by using interpolation. However, typical interpolation algorithms do not

produce good results in demosaicking [4, 5]. Observed artifacts are zipper effect, aliasing and false colors. These artifacts are due to the fact that color channels are not independent from each other. For RGB images, cross correlation between channels has already been found to vary between 0.25 and 0.99 [6]. Based on this fact, two basic color models for handling color correlation have been proposed in the literature. In color ratio model, it is assumed that the ratio of color channels is constant in the interior part of an object [7]. Division is a nonlinear operation. Hence, it is usually the most time, area and power consuming operation among the four basic arithmetic operations. In color difference model, it is assumed that the difference between color channels is constant in the interior part of an object [8, 9]. As its name implies, this model does not suffer from division operation. Another advantage of this model is that it perfectly fits linear interpolation model which is the simplest interpolation method [10]. Hence, computation cost of an algorithm based on color difference and linear interpolation models is lower than that of an algorithm based on color division and appropriate interpolation models.

In the literature both color difference [8],[10]-[19] and color ratio [7],[20]-[25] have been extensively used. Based on these models, interpolation methods for demosaicking have been developed. In non-adaptive interpolation, bilinear interpolation is the most preferred approach [8, 10]. In edge-adaptive interpolation, the interpolation is done along the edges detected by edge classifiers [7, 9], [14]-[20], [26]-[28]. After inital interpolation, some methods try to refine the high frequency components of interpolated color values so as to reduce aliasing effect [22, 31]. In post processing methods, the demosaicked image is subject to false color-correction and sharpening processes [19, 29, 30]. Some of these methods use the original uncorrupted Bayer CFA data.There are also some methods like pattern matching and template fitting so as to obtain sharper contours and better textured regions [6, 11]. Demosaicked image quality can be improved by denoising [32] or enhancing luminance [33] and contrast [34]. There are also preprocessing methods: Since mosaicking is a linear shift-variant process, CFA samples are subject to scaling and linear shifting operations in a recent work so that they will be normalized at the input of the interpolator [21].

All of the methods proposed in the literature can be used for "off-line" demosaicking, i.e. a host computer is used to demosaick the image data acquired by the camera. However, it cannot be claimed that all of them are suitable for "on-line" demosaicking where the camera directly produces the demosaicked image, because the computational resources and storage units must be small enough to fit in a digital imaging device. Another important issue is the time that is spent during demosaicking. This is an important issue during real-time demosaicking in devices like video camcorders. Power consumption is another challenging problem that should be handled carefully in portable imaging devices since all of them rely on batteries for operation. In order to meet all of these constraints, one can consider choosing an algorithm that requires a few operations per pixel and small storage. Therefore iterative methods like successive approximation [10] are not favorable for on-line demosaicking, because the number of operations per pixel is dependent on the number of iterations. Post processing improves the image quality but also increases computational cost. Some post processing algorithms like median filtering [29] can be included to the demosaicking system with very little overhead. On the other hand, some sophisticated algorithms [20, 22, 23] use complex computational steps. Algorithms of this sort are good for off-line demosaicking.

Similarly, memory requirements for on-line demosaicking has to be low. In demosaicking, each missing color of a pixel is estimated with estimated or measured color values of the current and neighboring pixels. Therefore, neighborhood of a pixel has to be available in the window of computation prior to demosaicking. The dimension of this window varies depending on different algorithms. It can be as small as a 3×3 pixels [7] or as big as the picture itself [1] [10] or even bigger [2] [22, 23]. It can be fixed like most algorithms or variable as it is in [20]. Neighborhood size can be treated as a measure for estimating storage cost. As a result, the algorithms that require big neighborhood should be avoided.

[1] In iterative algorithms like [10], the neighborhood can be as big as the picture itself because changes in a pixel causes neighboring pixels to change at each iteration.

[2] In these algorithms, full-color images and full-color sub-images are produced and used in demosaicking

This thesis focuses on finding a cost-effective solution to the on-line demosaicking problem. The proposed method, which is called ECI+OP, finds one-pixel patterns in a mosaicked image. In this way, textured patterns are restored better than other low-cost solutions in the literature. Moreover, its computational cost and storage requirements are lower than algorithms that produce similar results. It is a gradient-based method and uses both measured and estimated color values for detecting the direction of an edge. The proposed method differs from the others in computing the gradient and processing the edge information: Edge-detector tries to decide whether a pixel is in a difficult texture region or in a smooth region. If it is in a difficult texture region, then the directional interpolation of ECI+OP (ECI plus one pixel pattern) is done. If it is in a smooth region, effective color interpolation algorithm (ECI [8]) is used. ECI is a low-cost and memory-efficient algorithm that produces very good results in the smooth interior parts of the objects. Therefore, it is a good candidate for on-chip demosaicking. On the other hand, it has problems in demosaicking the highly-textured regions [8]. Some methods have already been proposed in the literature so as to improve drawbacks of ECI [10, 23, 34]. However, they are either expensive or inefficient.

It is also expected that ECI+OP edge-detection and edge-adaptive interpolation methods to be safely used with other methods in the literature even though it has not been tried yet. If desired, post processing methods can be applied to improve the quality of the image produced by the proposed method. Experimentally, ECI+OP method outperforms other low-cost algorithms [8, 9, 34].

A core for the ECI+OP algorithm has been implemented by using the VHDL language. The verification of the core is done on a medium-scale general purpose FPGA, Virtex-II XC2V500. The memory requirement of the algorithm has been low as expected. Four line memories are used inside the FPGA device. The core can process the size of $1000 \times n$ still images. There is a constraint on the image width in the design. Since the line memories used in the design can hold 1024 pixel values, there is an upper bound on the width of the image. This constraint can be relaxed by combining more than one line memories together. On the other hand, there is no limit on the height of the images. The line memories are used in rotation in order to hold

the pixel values in an image row. For video processing, the core can process at most 1000×1000 pixels of real-time video input. The core processes at a frequency of 25 MHz. At this frequency, 25 images can be processed per second which is enough for video processing purposes.

The thesis is organized as follows: In Chapter 2, ECI algorithm will be summarized. In Chapter 3, the proposed solution is presented with its abilities and limitations. Its computational cost and comparisons with other existing methods are discussed with experiments. In Chapter 4, the details of the core implementation are described. The FPGA implementation of the core is also presented. Chapter 5 concludes the work.

2. EFFECTIVE COLOR INTERPOLATION (ECI)

ECI is based on color difference model. It assumes that R and B values are correlated with the G values in the neighborhood of a pixel which is a 5×5 window (Figure 2.1(a)). Based on this model, K_R and K_B are defined as:

$$K_R = G - R \tag{2.1}$$

$$K_B = G - B \tag{2.2}$$

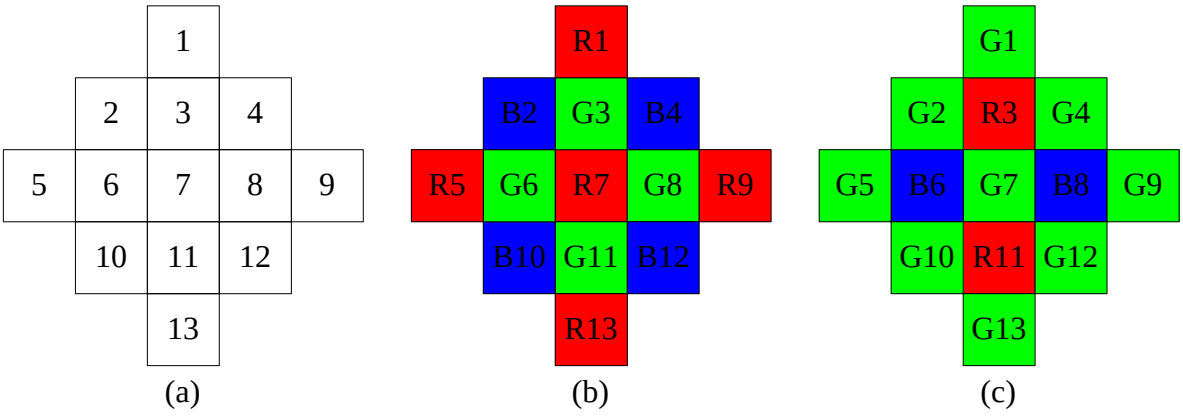

Figure 2.1. (a) 5×5 interpolation window for ECI. Reference CFA samples in the window when central sample is (b) red, and (c) green.

ECI is a two pass algorithm. Firstly, all missing green values are calculated. Referencing Figure 2.1(b), estimation of the G value at R_7 pixel is done by averaging K_{R3}, K_{R6}, K_{R8} and K_{R11}:

$$G_7 = R_7 + 0.25\,(K_{R3} + K_{R6} + K_{R8} + K_{R11}) \tag{2.3}$$

Since R_7 is not surrounded by R values, K_R values are obtained by estimating the R values from the two adjacent R samples. For example, calculation of K_{R3} needs G_3 and R_3 color values. However, that pixel has only the green value, G_3. Estimation of

R_3 can be done by averaging R values surrounding G_3:

$$K_{R3} = G_3 - R_3 = G_3 - 0.5\,(R_1 + R_7) \tag{2.4}$$

After green plane interpolation, red and blue channel interpolations are done. In Figure 2.1(b), blue is the other missing color. Its value is calculated by using K_B's of the blue values at the neighbourhood.:

$$B_7 = G_7 - 0.25\,(K_{B2} + K_{B4} + K_{B10} + K_{B12}) \tag{2.5}$$

In this way, all color values are completed for a red or blue CFA sample.

Now let us consider another case shown in Figure 2.1(c). Here, CFA sample of the central pixel provides us green color value. Since there are only two neighbors for each missing color, they can be estimated as follows:

$$R_7 = G_7 - 0.5\,(K_{R3} + K_{R11}) \tag{2.6}$$

$$B_7 = G_7 - 0.5\,(K_{B6} + K_{B8}) \tag{2.7}$$

3. INTERPOLATION IN TEXTURED REGIONS

Performance of the ECI algorithm is quite low at extremely high frequencies [8]. This drawback can be reduced by using the edge-directed interpolation method that is described in the following section. It can also be used with other methods in the literature. However, this method cannot be regarded as a post-processing algorithm because edge-directed interpolation of one pixel affects the interpolation of neighboring pixels.

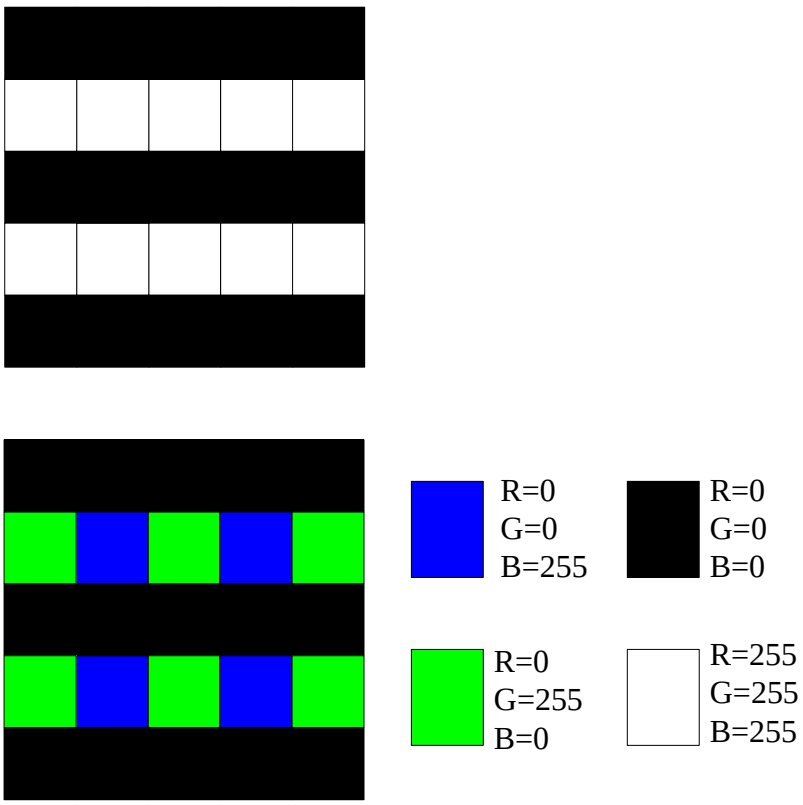

Figure 3.1. A one-pixel pattern and its corresponding mosaicked image. R, G and B values are given for colors.

The proposed algorithm successfully detects edges which are apart from each other by just one pixel, as observed in Figure 3.1. Green plane is used for both detection and interpolation. The pixel neighborhood for edge detection is a 3×3 window which is located at the center part of the 5×5 window of the ECI (Figure 3.2).

In this chapter, we introduce the improved algorithm called ECI+OP and it is explained in detail in Section 3.1. The discussion of the algorithm is made in Section 3.2. The computational cost of the algorithm and the comparisons between the ECI+OP and the previous algorithms in terms of computational cost are presented in Section 3.3. Finally, experimental results are given in Section 3.4.

3.1. The ECI+OP Algorithm

Let I_H and I_V be the horizontal and vertical interpolators for the pixel at the center. Regardless of the CFA color sample, the interpolators for central pixel are defined as:

$$I_H = 0.5\,(G_6 + G_8) \tag{3.1}$$

$$I_V = 0.5\,(G_3 + G_{11}) \tag{3.2}$$

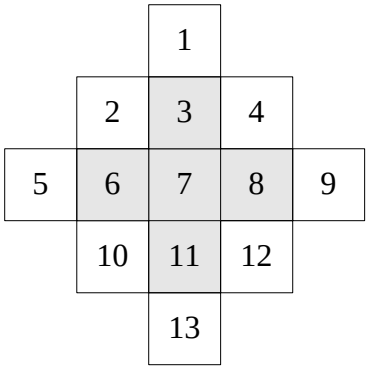

Figure 3.2. Edge-detector and edge-directed interpolator window on ECI window

Decision on the existence of an edge is given by interpreting D which is the difference between two interpolators.

$$D = |I_H - I_V| \tag{3.3}$$

False edge detection can be avoided by checking whether D is greater than a threshold value α. This threshold can be set by the camera manufacturer. If D is smaller than α, then it is assumed that the pixel is in the smooth region and all of its missing colors can be estimated with ECI. Otherwise, it is assumed that it is on an edge and an edge directed interpolation has to be carried out.

The direction of the edge is estimated by using the G value of the pixel at the center. In this operation, it is assumed that if G value is closer to I_H, then it is more likely that the edge is horizontal. Hence, two classifiers δ_H and δ_V are defined for vertical and horizontal edge classifiers:

$$\delta_H = |G_7 - I_H| \tag{3.4}$$

$$\delta_V = |G_7 - I_V| \tag{3.5}$$

In other words, if $\delta_H < \delta_V$, then the edge is horizontal. Obviously, only half of the CFA samples are green. Therefore G value for each red or blue CFA sample has to be estimated by the ECI algorithm prior to edge direction detection. Depending on the color of the CFA sample for the central pixel, there are two cases in the proposed method:

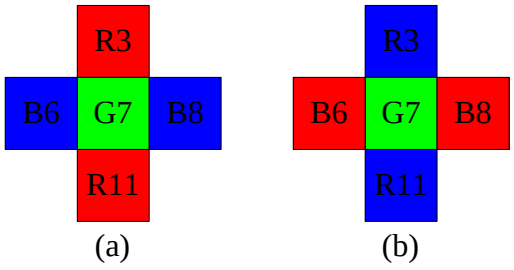

Figure 3.3. Edge-directed interpolation for the green CFA sample at the centre: (a)X=Blue and Y=Red, (b) X=Red and Y=Blue

3.1.1. CFA Sample for the Central Pixel is Green

In this case, blue and red values are missing in the central pixel. However if one missing color X exists in the horizontal neighboring CFA samples, the other missing color Y appears in the vertical neighboring CFA samples as shown in Figure 3.3. Edge directed interpolation allows interpolation in one direction. For example, if the central pixel is on a horizontal edge, then the missing X color value is calculated with the linear interpolation of X samples in the horizontal direction:

$$X_7 = 0.5\,(X_6 + X_8) \tag{3.6}$$

However, there is not enough information for making a good edge-directed estimation for Y color value in a horizontal edge. Therefore, this value is computed with the ECI algorithm. Interpolation along a vertical edge is carried out in a similar manner.

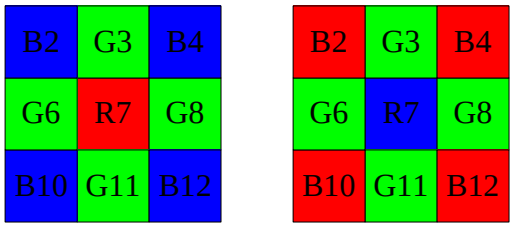

Figure 3.4. Edge-directed interpolation for a blue or red CFA sample at the center: (a)Central CFA sample is Red and C=Blue, (b) Central CFA sample is Blue and C=Red

3.1.2. CFA Sample for the Central Pixel is Red or Blue

In this case, one missing color is green. Let the other missing color be denoted by C. Green CFA samples surround the central pixel in the horizontal and vertical directions. The CFA samples of C appear in the diagonal directions. This placement is shown in Figure 3.4. Whenever there is an edge, the green color value of the central pixel is the same with the directional interpolator. For example, if there is an horizontal

edge, then

$$G_7 = I_H \tag{3.7}$$

Calculation of other missing color is not as straightforward as green value estimation. Diagonal CFA samples cannot be used directly because the distance between each C pixel is two pixels. Therefore estimated C values for green CFA samples on the edge are used. For example, if the edge is horizontal, then

$$C_7 = 0.5\,(C_6 + C_8) \tag{3.8}$$

Interpolation along a vertical edge is done similarly.

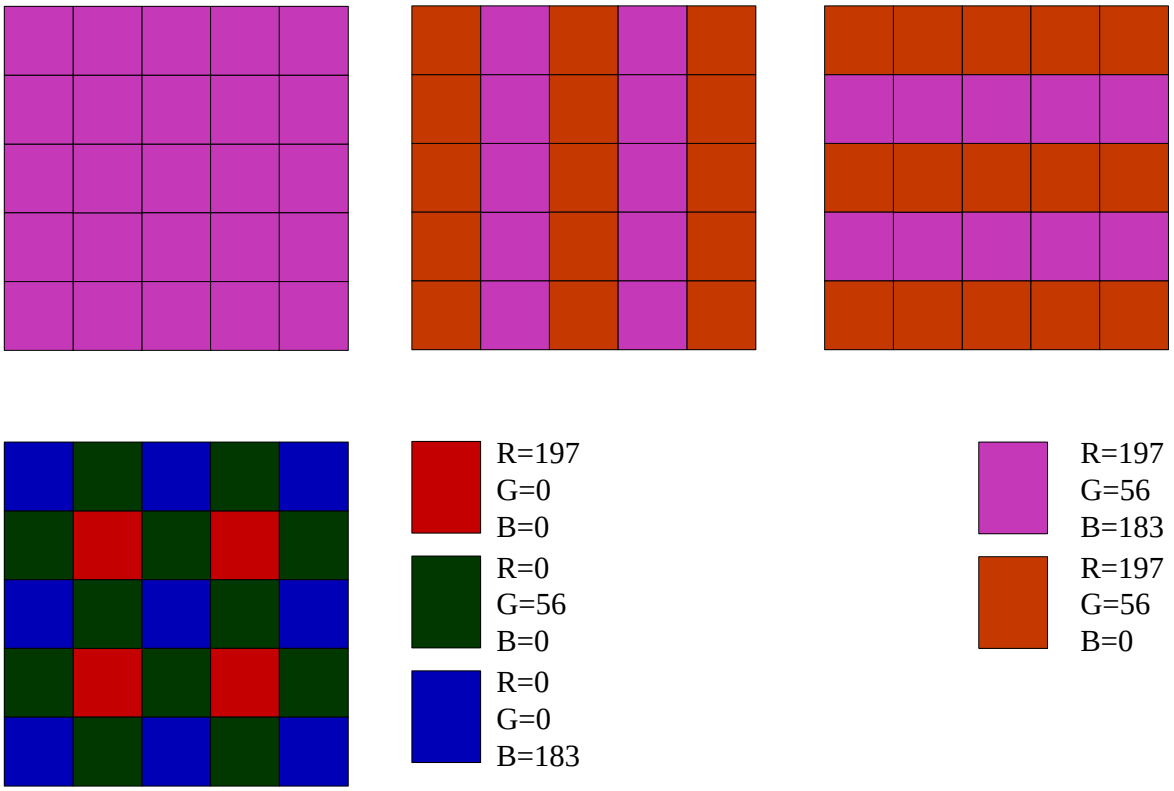

Figure 3.5. The Bayer array may produce the same mosaicked image in the images where either R or B color values are allowed to vary.

3.2. Discussion

The proposed algorithm is effective in detection of one-pixel patterns when the color information varies in G color values, but it does not give correct results when the information varies only in R and B values in one-pixel patterns whereas G values are uniform. However this limitation is valid for every algorithm developed for a Bayer array because the problem lies in the placement of the color cells. This can be illustrated with an example: Consider the synthetic images shown at the upper row of Figure 3.5. In these images, G and R values are constant. Blue values are also held constant in the first one, but allowed to vary in the others. After the Bayer array, all of these images have the same mosaicked image.

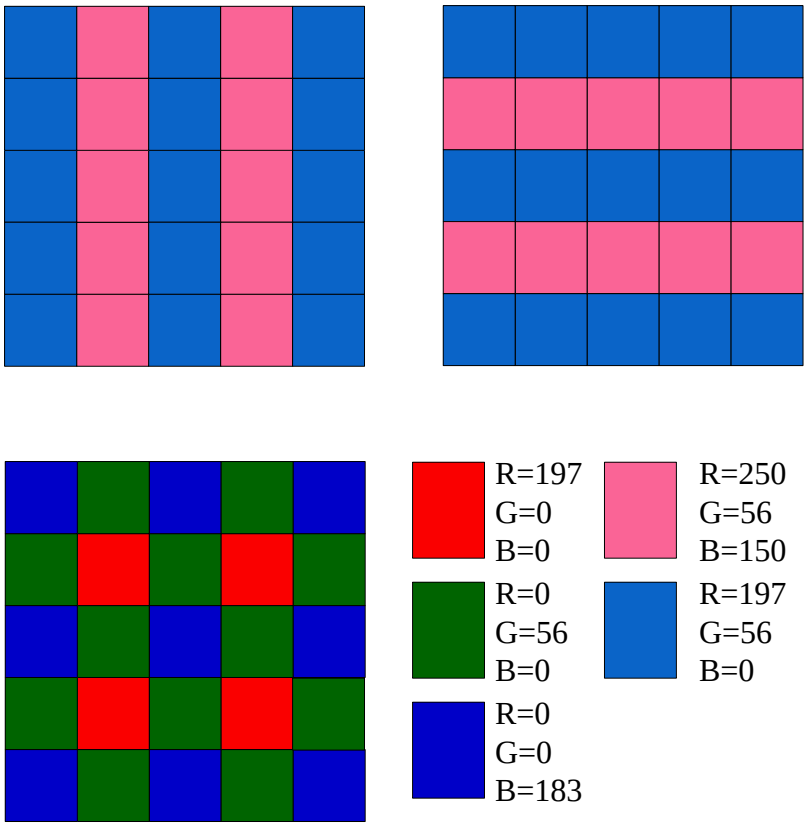

Figure 3.6. Texture may not be estimated correctly in the images where G color values are held constant while R and B color values vary simultaneously.

Another example is shown in Figure 3.6 where R and B values are allowed to change simultaneously, but G is held constant. It is impossible to extract the correct

patterns in these examples and similar images when green channel is not effected with one-pixel patterns. The solution to this problem can be found by either changing the type of the CFA array.

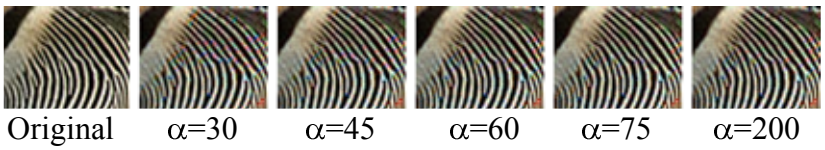

Figure 3.7. Effect of α in demosaicking. The best threshold value for this image is $\alpha = 75$

The edge-detector of ECI+OP detects each variation in G values. However, identification of a variation as an edge is completely dependent on the value of the threshold α. Similarly, detection of one-pixel patterns with gradual edges depends on α which is a parameter to be determined by the camera manufacturer. Undesirable results can be obtained if α is not chosen appropriately (Figure 3.7).

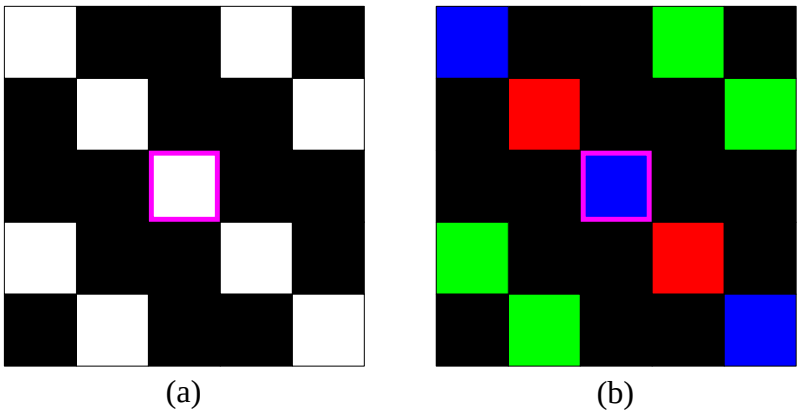

Figure 3.8. A one-pixel wide diagonal pattern (left), and its corresponding mosaicked image (right).

The algorithm can also extract many diagonal patterns. Several examples for diagonal patterns have been presented in Section 3.4. However, when the pixel separation between two diagonal lines is one pixel, then it might not generate correct results due to lack of green samples. In Figure 3.8, the CFA sample at the center (identified with a pink border) has no green CFA samples at the diagonal neighbors. Hence if it is

tried to make an interpolation in the diagonal direction, the estimated green values at the diagonal pixels obtained from ECI can be used. However, ECI produces the same green values for the pixel at the center and its northwest and southeast neighbors. So the interpolation effort in the diagonal direction will be redundant. Actually, the edge-directed algorithm does not even operate at such patterns. It leaves interpolation to ECI.

Even though the proposed algorithm has been combined with ECI, it can also be used with other spatial algorithms that are based on color difference and color ratio model. It can be used for replacement of an existing edge-detection and edge-adaptive interpolation mechanism. If the output of the proposed solution is post processed in the way that it is proposed in [19], then the results will be improved.

3.3. Computational Cost

Computational costs of the ECI+OP algorithm and other selected algorithms from the literature are presented in Table 3.1. These costs are computed by counting the operations that needed to estimate each missing color value for every CFA sample. O_G, O_B and O_R are defined as the number of operations that are used in the calculations of R and B values for G samples, R and G values for B samples and B and G values for R samples respectively. Since green CFA samples are two times more than B or R samples, then the average demosaicking cost for a pixel is

$$O_{AVE} = 0.5 * O_G + 0.25\,(O_B + O_R) \tag{3.9}$$

The varible i is used in successive approximation method because number of operations per pixel depends on the number of iterations. Computational cost of optimal recovery cannot be calculated by using the approach in other algorithms, because the green plane interpolation requires a training set and an optimization problem is solved. Green plane interpolation is done in two passes and the authors estimate each pass to take 1200 operations per pixel [20].

Table 3.1 shows that the proposed algorithm (ECI+OP) is quite inexpensive compared to other methods in the literature. Obviously it is more expensive than ECI, because it uses ECI as a subroutine. However, this overhead in computational resources occupies at most 15% of the smallest Virtex-II FPGA.

The ECI algorithm realizes the color plane interpolations in a 5×5 window. The proposed algorithm makes edge detection in a 3×3 window whose center coincides with the center of 5×5 window as shown in Figure 3.2. It uses a subset of pixels that are used by the ECI. Therefore it does not cause an additional on-chip memory cost.

Table 3.1. Computational cost per pixel (i: number of iterations)

	ADDs	MULTs	SHIFTs	ABSs	COMPs	DIVs	Total
ECI [8]	9	0	4.5	0	0	0	13.5
ECI+Luminance [34]	101	43	15	0	5	11	175
Color Filtering and a posteriori Decision [19]	33	0	5.5	6	0.5	0	45
Successive Approximation [10]	$34i$	0	$16.5i$	0	0	0	$50.5i$
Variance of Color Differences [15]	263.5	18	56.5	40	2	22	402
(ACPI) [9]	24	0	19.5	2	1	0	46.5
Primary Consistent Soft Decision [14]	258	4	6	0	3	0	271
Optimal Recovery (except green plane interpolation) [20]	149	0	3	0	0	14	166
Proposed algorithm (ECI+OP)	15.5	0	8	2	2	0	27.5

3.4. Experimental Results

The color images shown in Figure 3.9 have been used to evaluate the demosaicking performance of the proposed solution. A problematic region in each image is also identified and the proposed solution is compared with other computationally inexpensive solutions (Figures 3.10-3.19). Visual comparison reveals that the proposed algorithm performs better than other low-cost solutions even though it has lower computational cost than all of them but the ECI. It is interesting to see that the algorithm presented in [34] might enhance contrast in images but their performance in problematic regions is worse than regular ECI.

For objective comparisons, three well-known metrics are used: CIELAB (ΔE^*_{ab}), mean square error (MSE) and mean absolute error (MAE). However, CIELAB is considered as a more important metric than others because it is designed to approximate human vision [35]. The performance of the proposed algorithm in complete images and in problematic parts is presented in Tables 3.2-3.7. If CIELAB results are examined for evaluating the performance of the proposed algorithm at problematic regions, it can be easily observed that the proposed algorithm outperfoms the others. MAE and MSE results for cropped parts are also much better than the others except one case which is Figure 3.17. In complete pictures, the performance of ECI+OP is better than the others in most cases. The average-case, worst-case and best-case performance of the ECI+OP algorithm compared to other ones are also presented. The average case performance of ECI+OP algorithm is the best in all three metrics. However, it can also be seen that there are some cases where the proposed algorithm is not very successful. In these cases, the worst-case error is at most -5%. However, there is one single case where the error is about -15% in MSE metric. This is the complete Eiffel1 image. When a detailed analysis is made, it can be seen that the problem is at almost-black regions where texture is quite dense. ECI gives the best MSE result for this image because it treats this region as if it is the interior side of an object and demosaicks it into a smooth almost-black region.

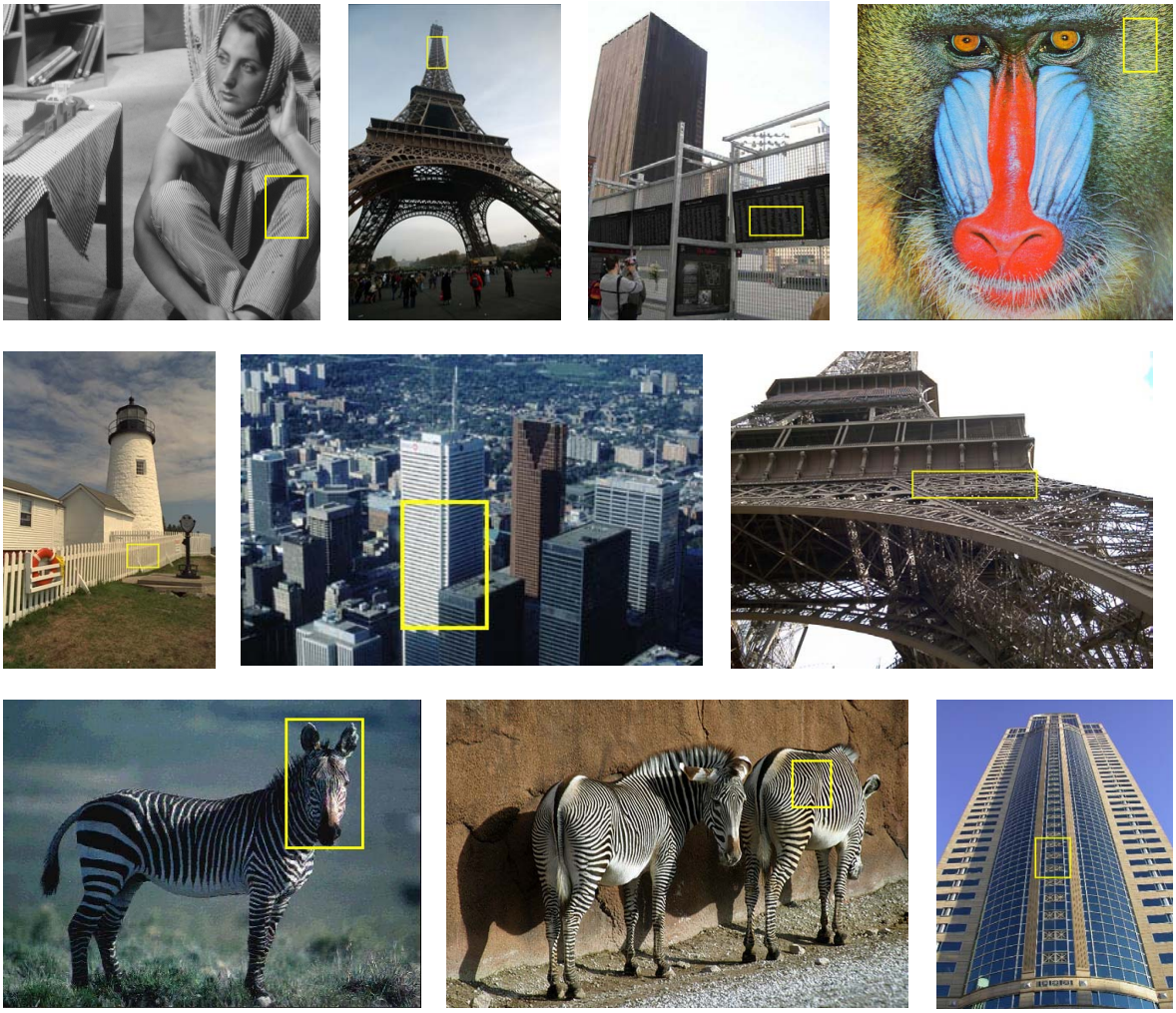

Figure 3.9. Test Images: Barbara, Eiffel1, Tower1, Monkey, Lighthouse, Tower3, Eiffel2, Zebra1, Zebra2, Tower2 (from left to right and top to bottom).

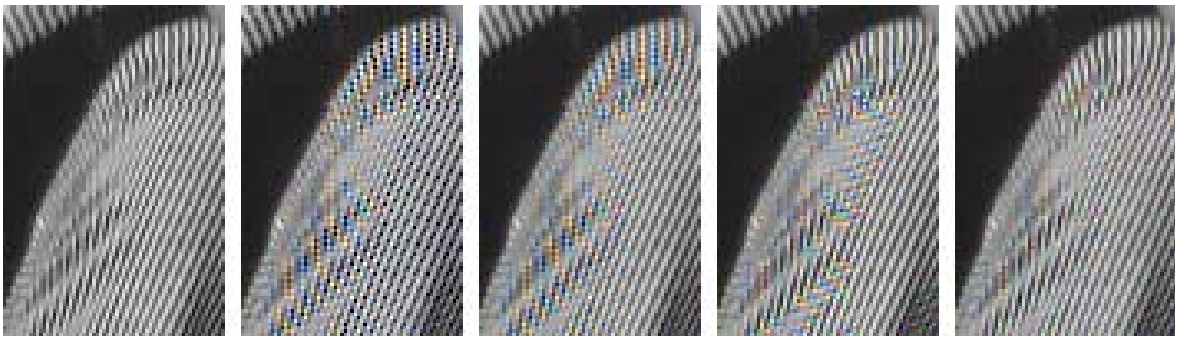

Figure 3.10. Cropped region of Barbara image: Original, ECI+Luminance [34], ECI [8], ACPI [9], and ECI+OP (From left to right).

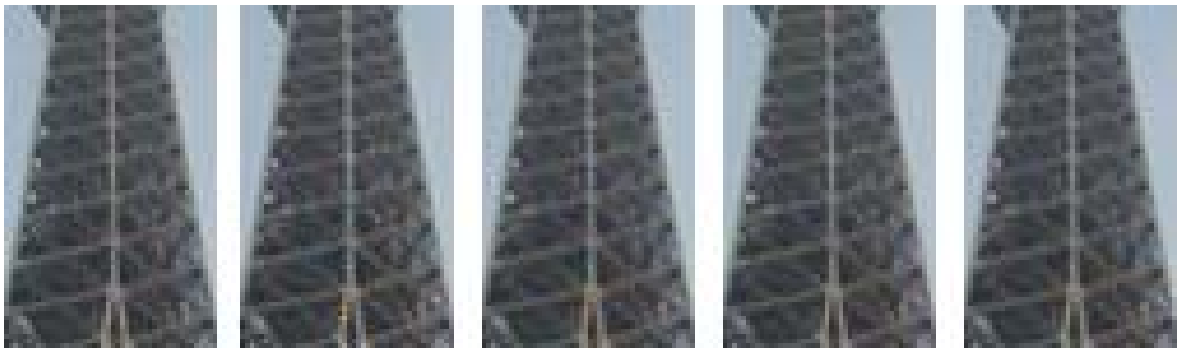

Figure 3.11. Cropped region of Eiffel1 image: Original, ECI+Luminance [34], ECI [8], ACPI [9], and ECI+OP (From left to right).

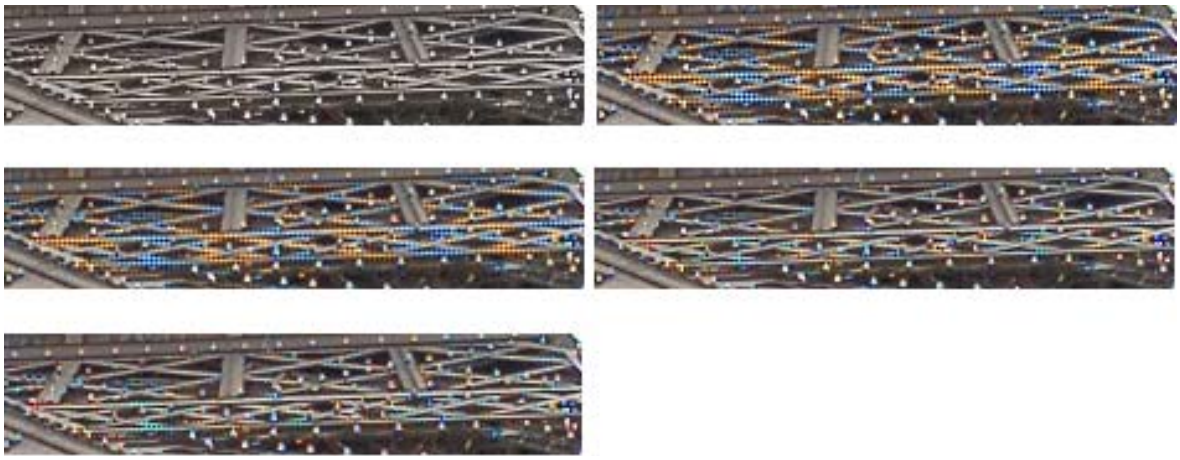

Figure 3.12. Cropped region of Eiffel2 image: Original, ECI+Luminance [34], ECI [8], ACPI [9], and ECI+OP (From left to right).

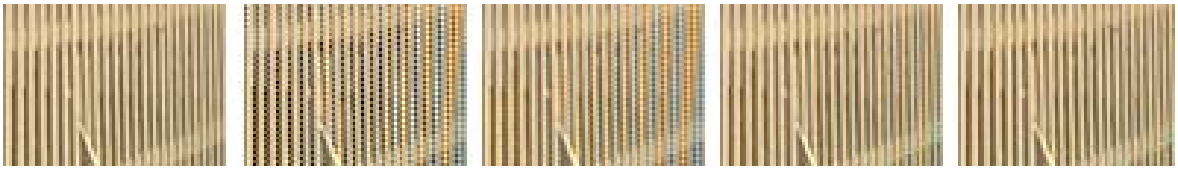

Figure 3.13. Cropped region of Lighthouse image: Original, ECI+Luminance [34], ECI [8], ACPI [9], and ECI+OP (From left to right).

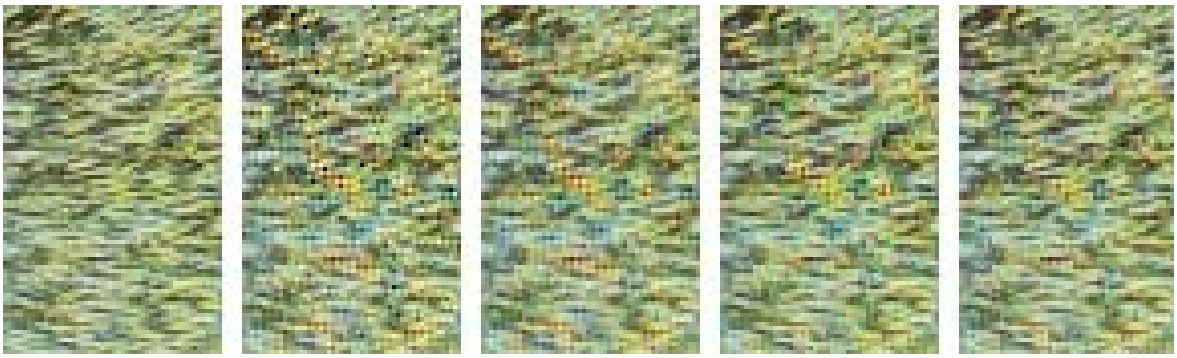

Figure 3.14. Cropped region of Monkey image: Original, ECI+Luminance [34], ECI [8], ACPI [9], and ECI+OP (From left to right).

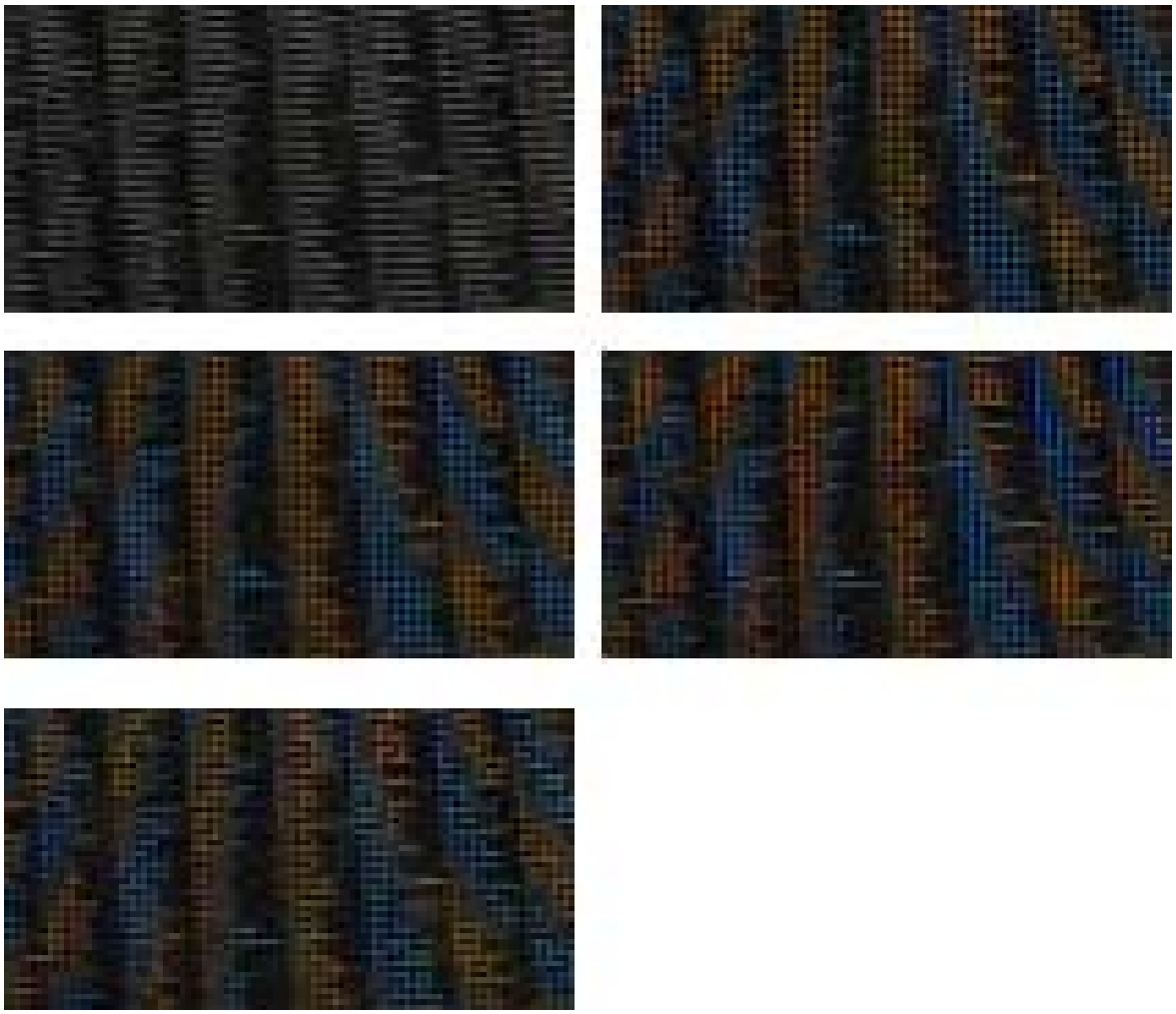

Figure 3.15. Cropped region of Tower1 image: Original, ECI+Luminance [34], ECI [8], ACPI [9], and ECI+OP (From left to right, top to bottom).

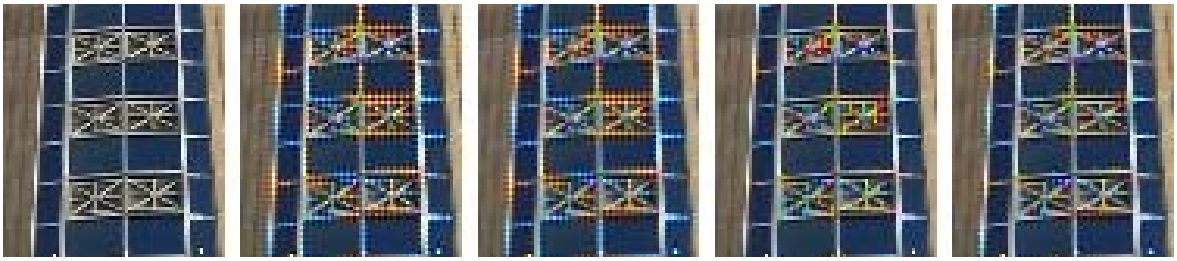

Figure 3.16. Cropped region of Tower2 image: Original, ECI+Luminance [34], ECI [8], ACPI [9], and ECI+OP (From left to right).

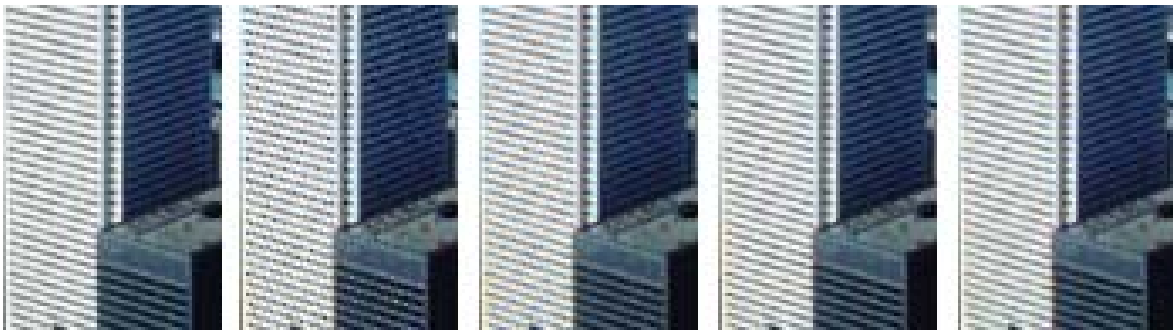

Figure 3.17. Cropped region of Tower3 image: Original, ECI+Luminance [34], ECI [8], ACPI [9], and ECI+OP (From left to right).

Figure 3.18. Cropped region of Zebra1 image: Original, ECI+Luminance [34], ECI [8], ACPI [9], and ECI+OP (From left to right).

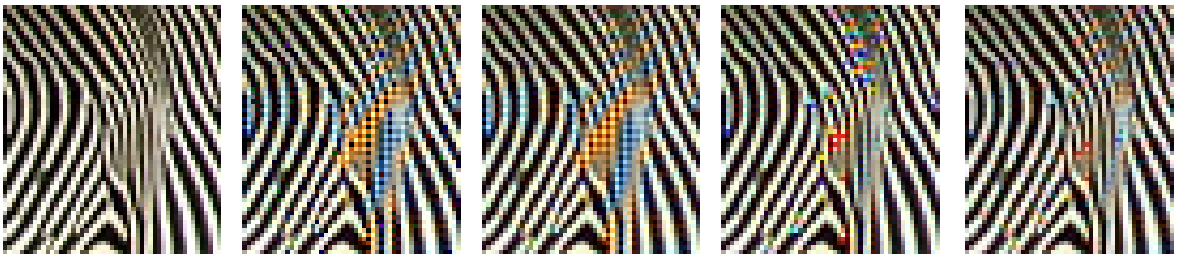

Figure 3.19. Cropped region of Zebra2 image: Original, ECI+Luminance [34], ECI [8], ACPI [9], and ECI+OP (From left to right).

Table 3.2. ΔE^*_{ab} results for cropped images given in Figures 3.10-3.19

					Percent Improvement of ECI+OP over		
	ECI	ECI+L	ACPI	ECI+OP	ECI	ECI+L	ACPI
Barbara	25,87	38,39	32,49	22,16	14,35	42,27	31,80
Eiffel1	10,34	10,82	10,80	10,13	2,04	6,38	6,28
Eiffel2	395,21	421,34	323,20	320,92	18,80	23,83	0,70
Lighthouse	60,50	83,10	40,30	18,25	69,84	78,04	54,72
Monkey	39,51	40,22	42,68	37,05	6,24	7,88	13,20
Tower1	192,87	199,53	244,69	175,66	8,92	11,96	28,21
Tower2	77,61	81,08	71,42	65,74	15,29	18,92	7,96
Tower3	21,69	24,48	21,29	19,33	10,89	21,04	9,23
Zebra1	30,33	33,70	34,71	30,10	0,75	10,67	13,29
Zebra2	127,82	147,00	122,63	122,01	4,55	17,00	0,51
				min	0,75	6,38	0,51
				max	69,84	78,04	54,72
				ave	15,17	23,80	16,59

Table 3.3. ΔE^*_{ab} results for complete test images given in Fig. 3.9

	ECI	ECI+L	ACPI	ECI+OP	Percent Improvement of ECI+OP over ECI	ECI+L	ACPI
Barbara	14,71	19,28	17,67	14,42	2,01	25,23	18,44
Eiffel1	18,29	19,92	18,21	18,45	-0,87	7,40	-1,32
Eiffel2	39,36	42,43	38,50	39,41	-0,15	7,10	-2,38
Lighthouse	10,98	11,97	10,81	9,89	9,93	17,38	8,50
Monkey	55,46	56,92	57,90	55,51	-0,09	2,48	4,13
Tower1	17,10	18,31	17,36	16,04	6,19	12,39	7,63
Tower2	34,96	36,81	33,98	31,31	10,44	14,94	7,84
Tower3	44,34	49,85	46,47	42,97	3,09	13,79	7,53
Zebra1	40,38	44,77	42,82	40,34	0,10	9,91	5,80
Zebra2	70,84	81,16	75,08	69,60	1,75	14,24	7,30
				min	-0,87	2,48	-2,38
				max	10,44	25,23	18,44
				ave	3,24	12,49	6,35

Table 3.4. MSE results for cropped test images given in Figures 3.10-3.19

	ECI	ECI+L	ACPI	ECI+OP	Percent Improvement of ECI+OP over ECI	ECI+L	ACPI
Barbara	323,10	572,55	302,28	139,14	56,94	75,70	53,97
Eiffel1	26,91	35,47	25,76	21,24	21,08	40,12	17,55
Eiffel2	615,55	697,52	345,85	315,35	48,77	54,79	8,82
Lighthouse	423,42	1117,90	177,39	28,56	93,25	97,44	83,90
Monkey	405,84	525,36	415,21	303,07	25,32	42,31	27,01
Tower1	214,83	235,66	334,43	167,22	22,16	29,04	50,00
Tower2	704,96	826,11	507,20	384,81	45,41	53,42	24,13
Tower3	154,68	231,47	69,76	72,83	52,92	68,54	-4,39
Zebra1	214,88	302,05	283,63	206,29	4,00	31,70	27,27
Zebra2	917,46	1170,20	644,88	588,32	35,88	49,72	8,77
				min	4,00	29,04	-4,39
				max	93,25	97,44	83,90
				ave	40,57	54,28	29,70

Table 3.5. MSE results for complete test images given in Fig. 3.9

					Percent Improvement of ECI+OP over		
	ECI	ECI+L	ACPI	ECI+OP	ECI	ECI+L	ACPI
Barbara	31,83	75,42	40,88	22,45	29,47	70,23	45,08
Eiffel1	33,40	50,28	42,00	38,53	-15,36	23,36	8,25
Eiffel2	114,02	141,32	104,75	105,22	7,71	25,55	-0,46
Lighthouse	36,99	61,61	23,87	20,05	45,79	67,46	16,02
Monkey	195,65	270,99	198,28	188,04	3,89	30,61	5,17
Tower1	62,27	81,44	57,42	46,54	25,27	42,85	18,96
Tower2	321,31	351,79	285,14	214,24	33,32	39,10	24,86
Tower3	87,16	119,39	68,18	65,56	24,78	45,08	3,84
Zebra1	46,23	66,32	58,80	46,06	0,38	30,55	21,67
Zebra2	124,05	200,39	125,57	112,20	9,55	44,01	10,64
				min	-15,36	23,36	-0,46
				max	45,79	70,23	45,08
				ave	16,48	41,88	15,40

Table 3.6. MAE results for cropped test images given in Figures 3.10-3.19

	ECI	ECI+L	ACPI	ECI+OP	Percent Improvement of ECI+OP over ECI	ECI+L	ACPI
Barbara	9,60	13,21	8,89	7,06	26,39	46,52	20,52
Eiffel1	2,96	3,34	2,98	2,79	5,89	16,70	6,38
Eiffel2	14,19	15,50	9,61	9,47	33,27	38,88	1,41
Lighthouse	12,23	21,16	6,67	3,00	75,48	85,83	55,06
Monkey	13,28	15,27	13,26	11,42	14,01	25,23	13,86
Tower1	9,47	9,80	10,79	7,79	17,81	20,54	27,85
Tower2	14,61	15,94	10,71	10,18	30,31	36,15	4,96
Tower3	7,01	9,08	4,82	4,91	30,00	45,95	-1,81
Zebra1	8,56	9,92	9,59	8,39	1,95	15,42	12,53
Zebra2	17,33	20,01	13,47	13,31	23,21	33,48	1,15
				min	1,95	15,42	-1,81
				max	75,48	85,83	55,06
				ave	25,83	36,47	14,19

Table 3.7. MAE results for complete test images given in Fig. 3.9

	ECI	ECI+L	ACPI	ECI+OP	Percent Improvement of ECI+OP over ECI	ECI+L	ACPI
Barbara	2,43	3,50	2,58	2,29	5,54	34,39	11,15
Eiffel1	2,16	2,55	2,19	2,22	-2,69	12,88	-1,20
Eiffel2	4,32	4,90	3,90	4,08	5,65	16,78	-4,50
Lighthouse	2,73	3,37	2,37	2,31	15,40	31,41	2,34
Monkey	8,84	10,28	8,76	8,68	1,77	15,58	0,98
Tower1	3,51	4,07	3,08	3,11	11,25	23,59	-1,00
Tower2	8,50	9,00	6,88	6,92	18,56	23,11	-0,60
Tower3	4,98	5,90	4,43	4,46	10,51	24,53	-0,53
Zebra1	3,10	3,59	3,29	3,06	1,09	14,78	6,90
Zebra2	5,57	6,93	5,77	5,37	3,65	22,50	6,90
				min	-2,69	12,88	-4,50
				max	18,56	34,39	11,15
				ave	7,07	21,96	2,04

4. IMPLEMENTATION

The ECI+OP algorithm is implemented as a core by using the VHDL language. The VHDL implementation is also verified by mapping it onto the FPGA, Virtex-II XC2V500. The main implementation concern is the convenience of the algorithm for "on-line" demosaicking. The computational resources and storage units should be small to fit the implementation in digital imaging device. The time spent for demosaicking should be short for real-time demosaicking. Low power consumption is preferred for portable devices.

The green interpolation is done first in most studies, since the Bayer Array includes twice more green data than red and blue data.Then the red and blue interpolations are performed. This approach is mostly used in "off-line" demosaicking. However, in implementation part it has some disadvantages. Performing the whole green interpolation at first means storing the whole image for the red and blue interpolations. In order to store the green-interpolated image, the size of the memory must be at least as big as the size of image and the memory size increases when the image size increases.

In this thesis, instead of carrying out the green, red and blue interpolations separately, they are combined and performed together. Disrupting the order of the interpolations is not logical, since the abundance of green data results in better demosaicking results when the green interpolation is performed at first. In the implementation, the order of interpolations is kept unchanged for a 7×7 window. The green interpolation is performed first and then the red and blue interpolations are carried out in a 7×7 window. When the whole image is considered, the 7×7 window traces the image and the green, red and blue interpolations seem to be fulfilled at the same time.

Combining the interpolations in a 7×7 window results in small memory requirements when compared to perform interpolations in order for the full image. Since the three interpolations are performed in the same window, the calculated R,G and B values do not need to be stored in memory and can be given from the output ports of

the core. After the image is started to be read to the demosaicking core, the interpolations for R, G and B values starts without waiting for the completion of the image. A demosaicked image pixel is produced from the outputs of the core as soon as the operations on that pixel is finished. Therefore, total time required to process an image is decreased for successive image processing operations.

4.1. Combining Interpolations

As shown in Figure 4.1 and Figure 4.2, the combination of interpolations are performed in order to find the missing values R and B at location (3,3). In this figure, C_l and C_r represent the R (or B) values on the left and right neighbors of the central pixel. C_u and C_d represent the B (or R) values on the up and down neighbors of the central pixel. As discussed in Chapter 2, the G values for C_l, C_r, C_u and C_d have to be calculated for the calculation of R and B values for location (3,3).

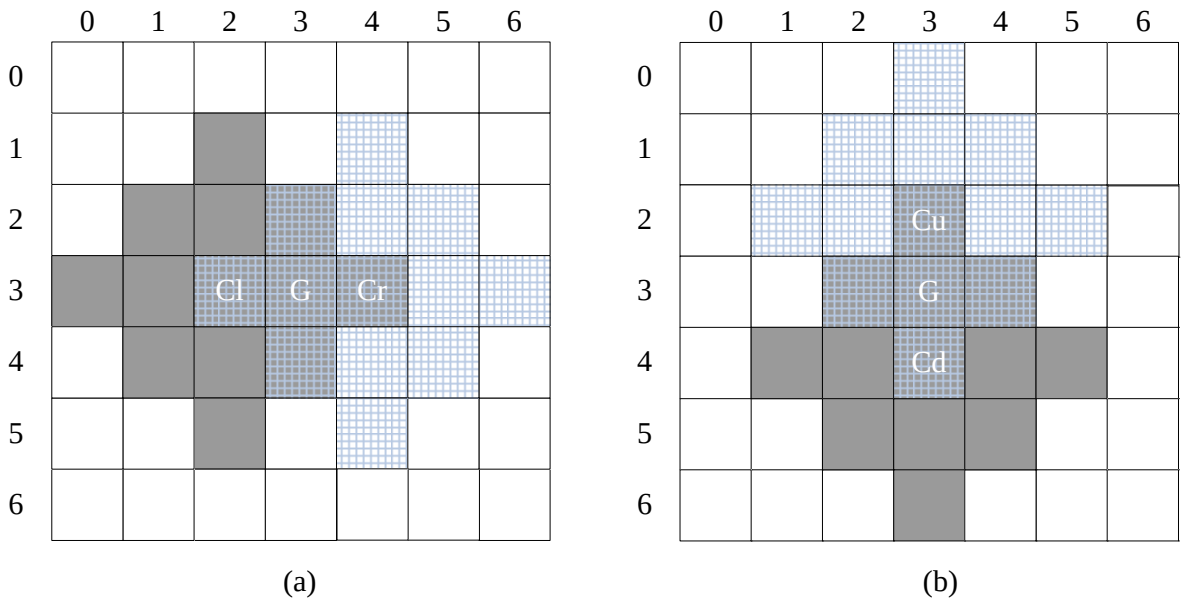

Figure 4.1. The 5×5 blocks to calculate neighboring G values

As shown in Figure 2.1(b), in a 5×5 window, where the central pixel is a C pixel (red or blue), the green value of the center pixel can be calculated. This 5×5 window is used to calculate the G values for C_l, C_r, C_u and C_d. The reason for using a 7×7 window is the fact that combination of four 5×5 windows constitutes a 7×7

window. The 5×5 blocks to calculate the G values are shown in Figure 4.1. In Figure 4.1(a), the values needed to calculate the G values for C_l and C_r are shown by grey and striped pixels, respectively. The values needed to calculate the G values for C_u and C_d are shown in Figure 4.1(b). The combination of the Figure 4.1(a) and Figure 4.1(b) is shown in Figure 4.2 which corresponds to a 7×7 window. The grey pixels are needed to calculate G values for C_l and C_r and the striped pixels are needed to calculate G values for C_u and C_d.

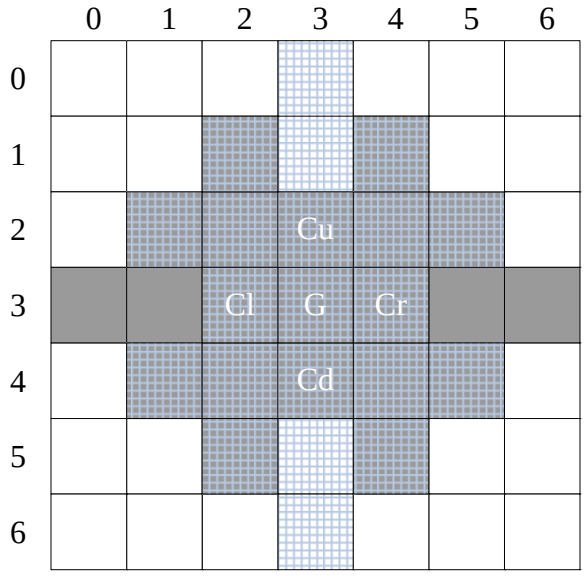

Figure 4.2. The 7×7 block construction

The 7×7 window which is called *Big Window* is shown in Figure 4.3. In Big Window BW, the three interpolations are done separately, and the BW traces the whole image file. In diamond shape BW, the main aim is to calculate the B and R values of the central pixel which is located at (3,3) (row and column index, respectively). In BW, while calculating the R and B values of the central pixel, the G values of the (2,3), (3,2), (3,4) and (4,3) are also calculated.

In Figure 4.3, there is also a *Small Window* which is a 3×3 window and used to calculate the R values on B pixels and the B values on R pixels. BW traces the rows of the image and calculates all missing G values and R and B values on G pixels. In Figure 4.3, it can be observed that the calculated values and the original values

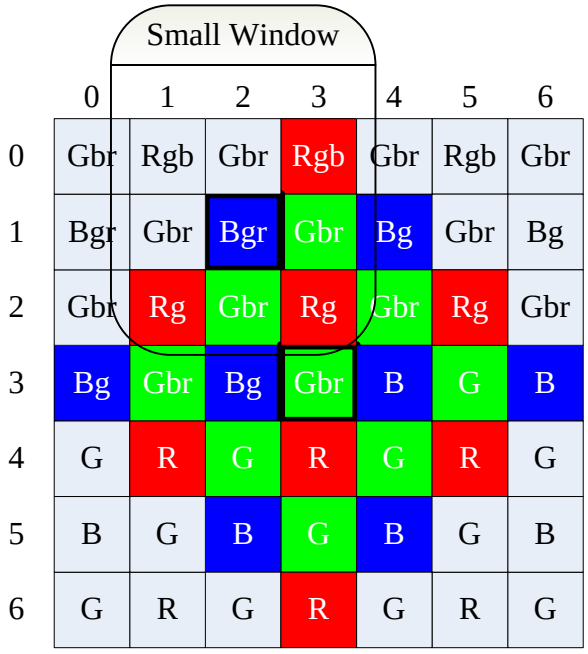

Figure 4.3. Big Window BW (7×7), the operation window after combining the interpolations together

of the Bayer Array. The uppercase letters are original Bayer Array values and the lowercase letters are the calculated values by the BW and SW (i.e. Gbr means G from original Bayer Array and the R and B values are calculated). By referencing the Figure 4.3, after the BW block performs its own calculations, the SW block follows it and completes the missing interpolations so as to obtain the full-color image. The reason for the operating distance of 2 rows between the SW and BW blocks is that the values calculated by BW is used by the SW and it waits the needed values to be ready.

In BW, the four G values of the neighboring R and B pixels are calculated first. In each calculation, the update operation for the calculated G values is also performed. When the four neighboring G values are ready, the R and B value calculation of the central pixel is performed by referencing the Figure 2.1(c). In order to calculate the central R and B values, the equation 2.6 and the equation 2.7 are used, respectively.

In SW, R values on B pixels or B values on R pixels are computed. In order to perform the mentioned operations, a 3×3 window is needed for both the calculation and the update operations. By referencing the Figure 2.1(b) and the equation 2.5, R

and B values can be calculated on B and R pixels, respectively. Now, it is more easy to see that the SW has to follow the BW at least 2 rows before since the SW block needs the central G value in 3×3 window and the C values at the corner points of the 3×3 window to calculate the K_R or K_B values.

When the BW and the SW blocks are considered, the R, G and B interpolations are combined together. As mentioned above, by using the given 7×7 window, the interpolations are made in order (not together) but the interpolations seem to be performed together after the BW and SW pass a particular part of the image.

4.2. The Design Overview

The input and output configuration of the core is shown in Figure 4.4.

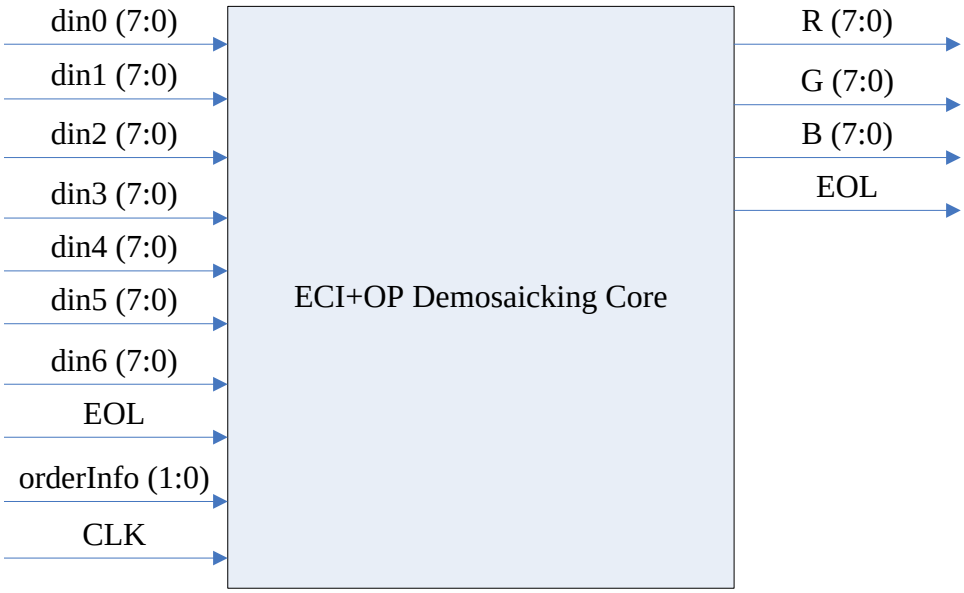

Figure 4.4. The input and output configuration of the full design

In Figure 4.4, the lines of mosaicked image are read by the wire through din_i ports. The *EOL* input informs the system that an end of line is met. These inputs are explained in detail in Section 4.3. The *orderInfo* signal is a 2-bit input and shows the type of the first two pixels of the first line of the Bayer Array. If the *orderInfo* signal is *00*, the first line of the Bayer Array starts with RG configuration. The first line of the

Bayer Array starts with BG, GR and GB configuration when the *orderInfo* signal is *01*, *10* and *11*, respectively. The Bayer Array samples and the corresponding *orderInfo* inputs are shown in Figure 4.5. The R, G and B outputs of the core represent the red, green and blue values of a pixel.

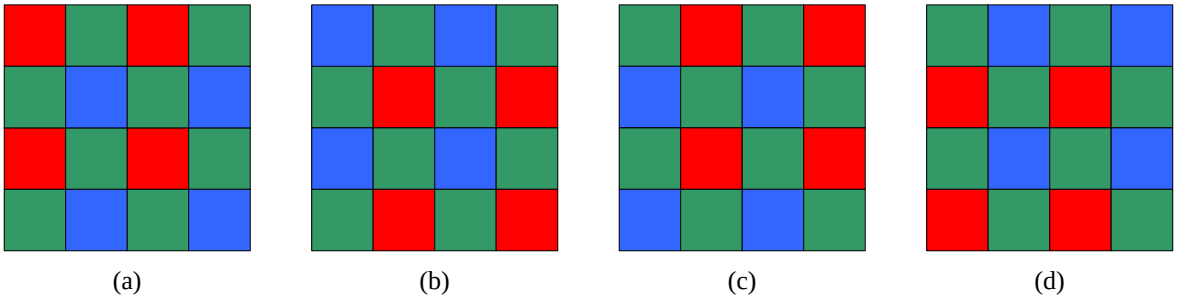

Figure 4.5. The Bayer Array patterns and corresponding *orderInfo* signals. (a) 00 (b) 01 (c) 10 (d) 11

The modules of the core are shown in Figure 4.6. After the image input is taken, it is put into a buffer which consists of register blocks. The buffer size is 7×7 (and an extension of 3×3 window) and the data needed for the BW block are kept ready in the buffer. The details about the buffer are explained in Section 4.3.

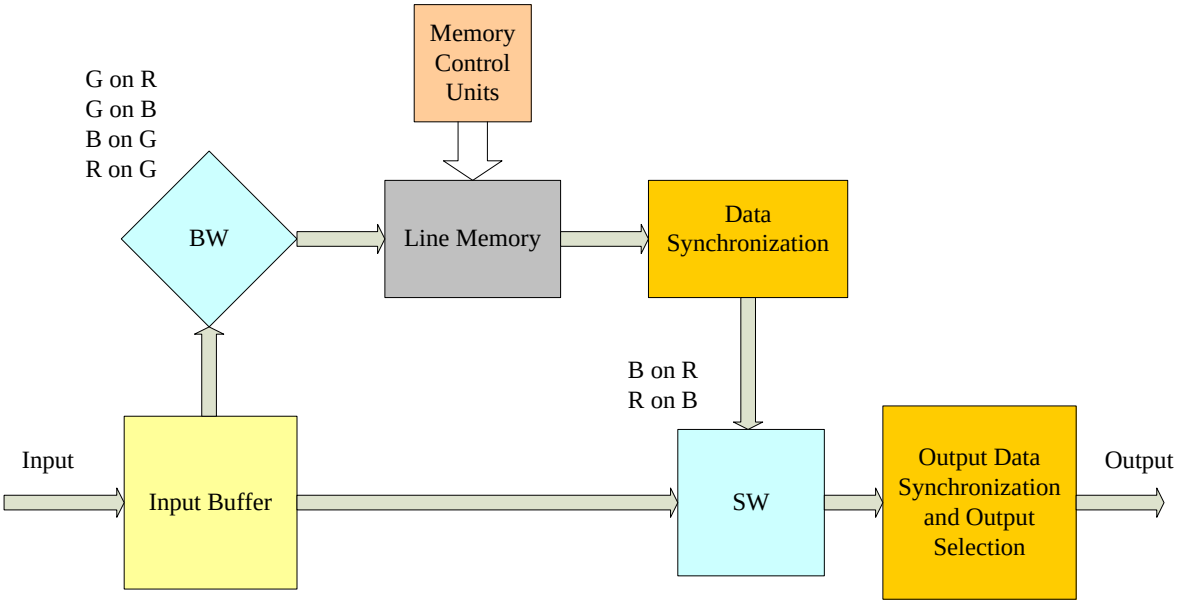

Figure 4.6. The design overview implemented by VHDL

BW block in the design is responsible for finding G in all R and B pixels and finding R and B values on G pixels as discussed in Section 4.1. The input for the BW block is provided from the input buffer. Since the data in the buffer is always moving, BW calculates the missing colors and sends the data to the memory. The detailed explanation for the BW will be given in Section 4.4.

In memory module of the Figure 4.6, four lines of the whole image is stored. These four rows are the operating row of the BW, which is the row that the central pixel belongs to in BW, and the first three rows above the operating row of the BW block. Keeping those four lines in the memory is important in order not to lose the data calculated by BW and transmit them to the SW. Since SW follows BW after two rows, the data calculated by BW have to be collected in memory. In Section 4.5, the memory organization and the read-write procedures will be described in detail.

In data synchronization module after the memory unit, the data needed for the SW block are taken from the memory and combined with the data coming from the input buffer. The data coming from the memory are the values calculated by the BW unit. On the other hand, the SW block also needs the original mosaicked image values which are taken from the input buffer. Data Synchronization part is discussed in Section 4.6 and the details of the SW block is explained in Section 4.7.

At the end of the design in Figure 4.6, another module is present for output data synchronization and output selection. They combine and synchronize the outputs of the SW block, BW block and the data in the input buffer so as to produce demosaicked image data. The outputs of the design are the R, G and B values of an image pixel. These modules are explained in Section 4.8.

4.3. Buffer

The buffer part of the core in Figure 4.6 is designed to feed the BW block. The mosaicked image data travels along the input buffer and they are passed to BW from the buffer. The internal structure of the buffer is shown in detail in Figure 4.7.

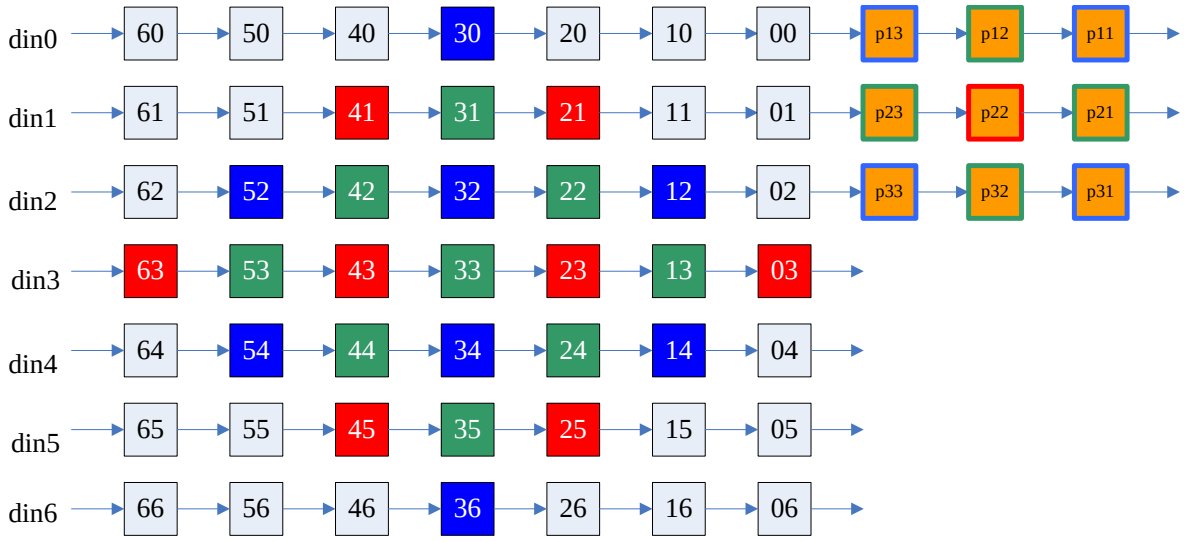

Figure 4.7. The internal structure of the 7×7 input buffer

Input buffer takes seven inputs from din0 to din6. They are 8-bit inputs (since R, G and B values ranges from 0 to 255) and each of them is either a green or a blue or a red due to the pixel composition of the Bayer Array. At the beginning of the process, first seven rows of the mosaicked image is fed to the buffer. In other words, first seven color values in the first column of the image is fed to the buffer in the first clock signal, then the first seven color values in the second column of the image is fed and so on. At each clock cycle, first seven color values (corresponds to the first seven rows in one column, 1×7 matrix) of one column of the image are fed to the buffer, from inputs din0 to din6 in order. This process takes n clock cycles if the width of the image is n. After feeding the first seven rows of the Bayer Array to the buffer in n cycles, the rows between 2-8 are fed to the buffer between the cycles n and 2n. When switching from 1^{st}-7^{th} rows to the 2^{nd}-8^{th} rows, an end-of-line (EOL) signal is given to the system in order to understand feeding the 1st-7th rows is over and switching to another 7 rows is performed.

Each node in the buffer is a 8-bit register and at each clock cycle it transmits the data from left to right (Figure 4.7). By using this strategy, 7×7 mosaicked data can be provided at the outputs of the registers. As shown in Figure 4.3, BW block needs a diamond shape data group as input and the buffer can provide this group of data once in two cycles. In Figure 4.7, there exists a G data at the center node (3,3)

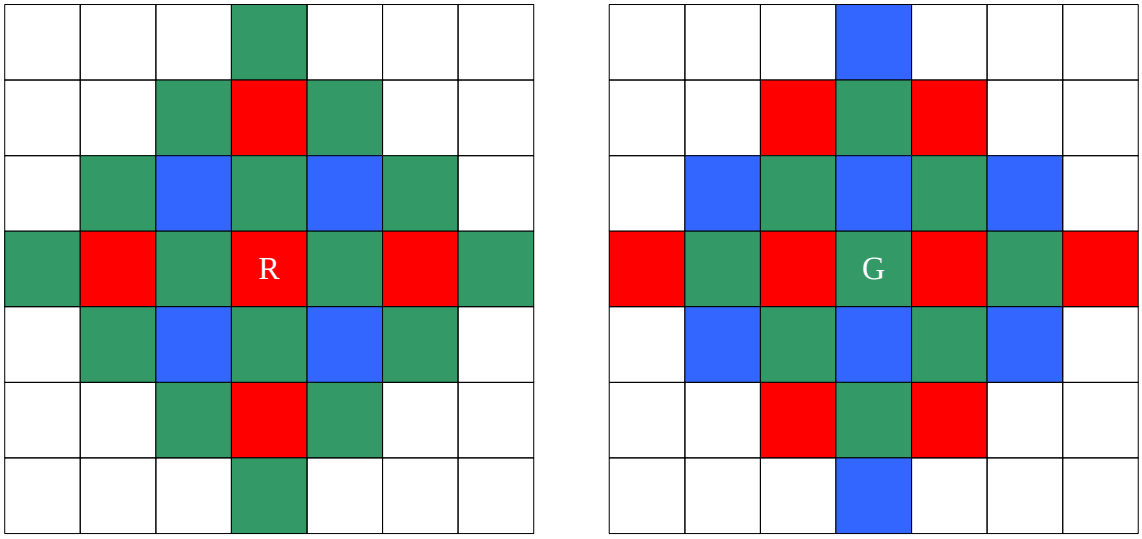

Figure 4.8. (a) The invalid input configuration for BW (b) The valid input configuration for BW.

at the current cycle. Now the data at the outputs of the flip-flops can be given to the BW block. In the next cycle, a R data will be shifted to the node (3,3) and this configuration (Figure 4.8(a)) will be invalid for the BW block since it operates when a G pixel is located on (3,3). After one cycle, the configuration again will be convenient for the BW block (Figure 4.8(b)).

While feeding the buffer with the 1^{st}-7^{th} rows, the BW takes the Bayer Array data from the buffer and starts to find the R and B values on G pixels and G values on R (or B) pixels in the 4^{th} row. BW calculates the missing color values at the 4^{th} row, since it operates on the central node (3,3) of 7×7 group of data in Figure 4.7. When an EOL signal is introduced to the system, BW block starts to calculate the missing color values at 5th row since the 2^{nd}-8^{th} rows of the image is fed to the buffer.

Finally, there exists a 3×3 window of orange p_{ij} values at the right-top of the buffer in Figure 4.7. It is extended from the first three rows of the buffer and it provides inputs to the SW block in Figure 4.6. The data in line memory and corresponding 3×3 window of p_{ij} data are used together by SW block to generate valid R values on B pixels and B values on R pixels.

4.4. Big Window

As discussed in Section 4.1, Big Window (BW) is used to calculate the R and B values of the central pixel G_{33} (Green at (3,3) location in buffer) of the buffer in Figure 4.7, and used to calculate the G values of the neighboring pixels B_{32}, B_{34}, R_{23} and R_{43} of the central pixel.

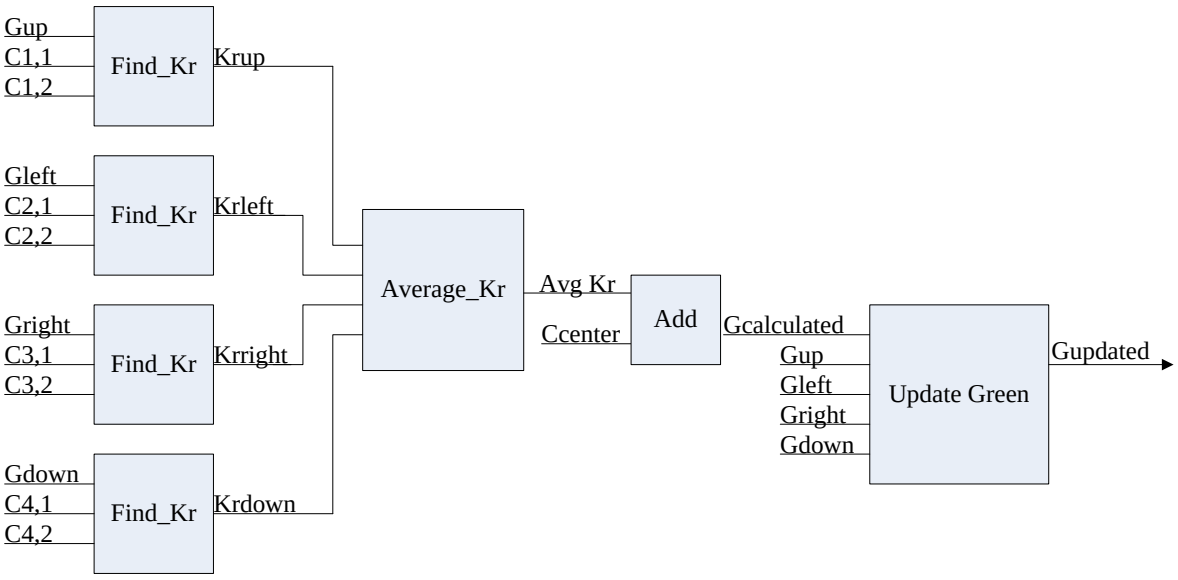

Figure 4.9. The Neighbor Green (NG) part of the BW block

In Figure 4.9, the Neighbor Green (NG) block is introduced as a part of the BW block. NG block is used to calculate the G value of one of four neighbor cells of the central pixel (3,3) in Figure 4.7 (B_{32}, B_{34}, R_{23} or R_{43}). Operation of NG can be illustrated with an example: The input connections for the NG unit to calculate the G value of the pixel (4,3) in Figure 4.7 is as follows:

- $Gup \rightarrow G_{42}$, $C_{1,1} \rightarrow R_{41}$, $C_{1,2} \rightarrow R_{43}$
- $Gleft \rightarrow G_{53}$, $C_{2,1} \rightarrow R_{63}$, $C_{2,2} \rightarrow R_{43}$
- $Gright \rightarrow G_{33}$, $C_{3,1} \rightarrow R_{43}$, $C_{3,2} \rightarrow R_{23}$
- $Gdown \rightarrow G_{44}$, $C_{4,1} \rightarrow R_{43}$, $C_{4,2} \rightarrow R_{45}$

By using this input configuration for the NG unit, the K_R values for the (4,2),

(5,3), (3,3) and (4,4) pixels are produced as Kr_{up}, Kr_{left}, Kr_{right} and Kr_{down} from *Find_Kr* units, respectively. Then an average taker unit finds the average of four K_R values and adds to the R value at (4,3) location. By performing these operations, the G value at (4,3) is calculated without an update operation. By using the G values at (4,2), (5,3), (3,3) and (4,4) pixels, the updated G_{43} value is calculated according to Section 3.1.

By using a similar strategy, the G values on B_{32}, B_{34}, R_{23} and R_{43} is calculated. In the BW block, there are four NG units, each for calculating one neighbor G value of the central pixel G_{33} in Figure 4.7.

In BW block, *calculateC* block is formed by combining two of four NG units. Two NG blocks which are used to calculate the G values at B_{32} and B_{34} locations are combined and called horizontal *calculateC* block. The other two NG blocks which used to calculate the G values at R_{23} and R_{43} locations are also combined and called vertical *calculateC* block. The design of a *calculateC* block is shown in Figure 4.10. The left, right, up and down phrases are used to explain the relative locations with respect to the central pixel G_{33} in Figure 4.7.

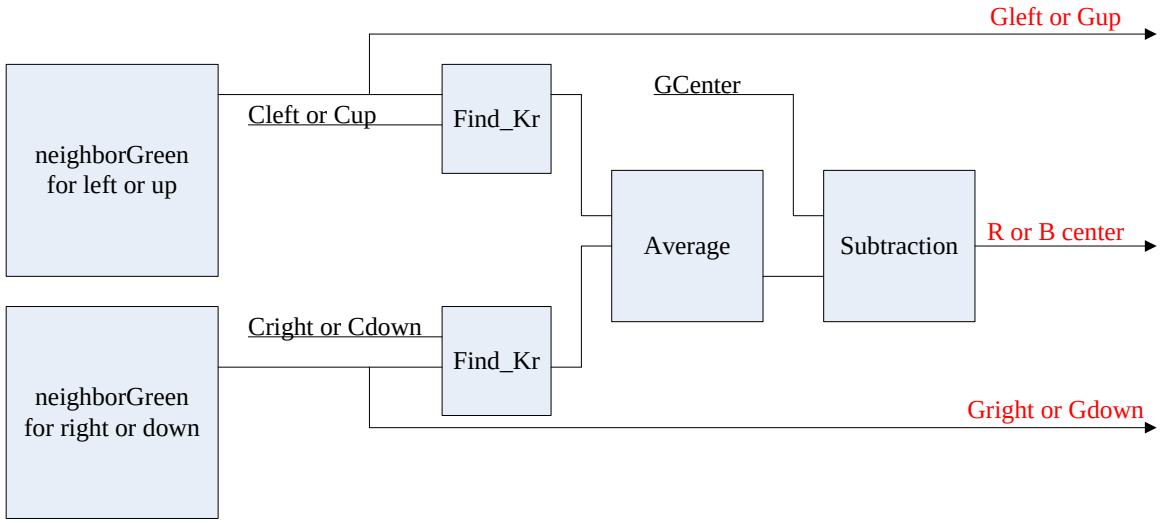

Figure 4.10. The calculateC part of the BW block

There are two *calculateC* units in the BW block, one of which is responsible for R value calculation of the central pixel and the other one for B value calculation. In

Figure 4.7, horizontal *calculateC* calculates the R value on the central G_{33} pixel, since R values are located on the left and right of the G_{33} pixel. In addition, the G values of the pixels on the left and right of the G_{33} pixel are calculated. The vertical *calculateC* block calculates the B value and the G values of the pixels on the upside and downside of the G_{33} pixel on the other hand. The outputs of the calculateC block is shown in red color in the Figure 4.10.

The whole BW block design can be given in Figure 4.11. BW block is composed of two *calculateC* blocks and one of them used to calculate the G values on horizontal neighbors and one C value at the center, the other one is used to calculate the G values on vertical neighbors and the other C value at the center.

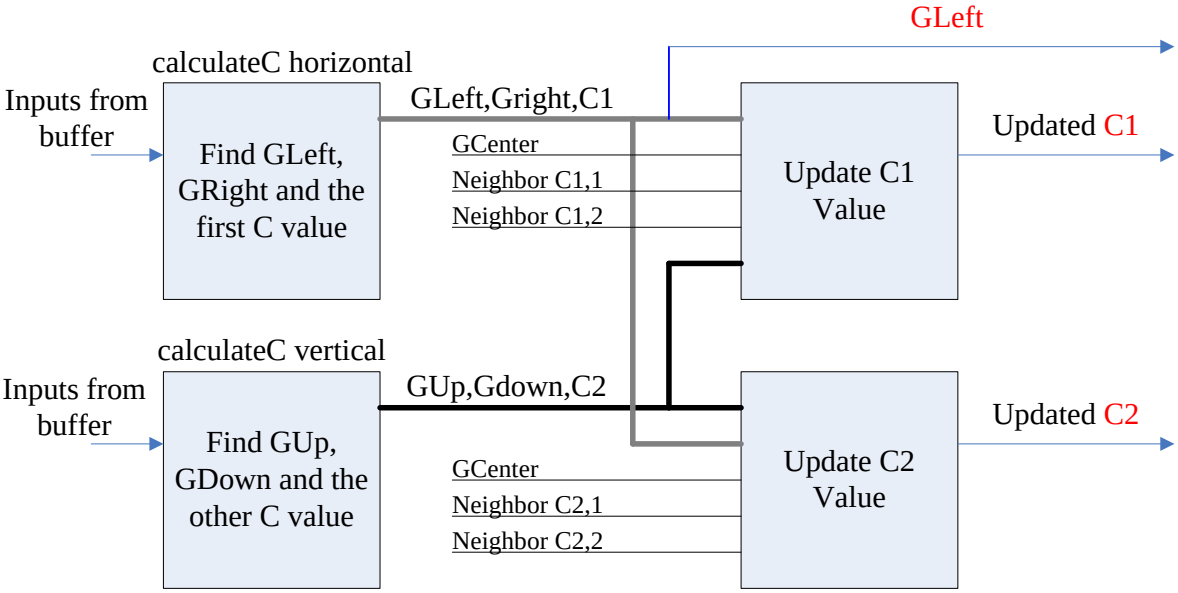

Figure 4.11. The Big Window BW block design

If C1 in Figure 4.11 is assumed to be the R value on the central pixel G_{33} in Figure 4.7, $C_{1,1}$ and $C_{1,2}$ corresponds to the R values neighboring to central pixel. Similarly, if C2 is the B value of the central pixel, $C_{2,1}$ and $C_{2,2}$ are the B values of the neighboring B pixels.

As can be seen in Figure 4.11, the C1 and C2 values that corresponds to R and B values of the central pixel are calculated and updated in BW block. These values are

the final values and will be written to the memory. The reason to write the values to the memory is that they will be needed in SW block calculations. The signals written in red in Figure 4.11 (G_{left}, C1 and C2) are the outputs of the BW block. Since BW block generates valid data once in 2 cycles as discussed in Section 4.3, the outputs of the BW block have 2 cycles to be written into memory. In the first cycle G_{left} is written to the memory, and in the second cycle C1 and C2 values are written to the memory. For example, the data generated for the left neighbor R_{43} of the central pixel G_{33} in Figure 4.7 are written in the first cycle, and the C1 and C2 values of the G_{33} pixel are written in the second cycle.

4.5. Memory Organization

The memory unit takes two types of inputs as discussed in the previous section. One of them is the G value at (4,3) and the other one is the B and R values of the central pixel of the BW block in Figure 4.7. In one cycle, the memory takes G_{43} input, and in the next cycle memory takes the R_{33} and B_{33} inputs. The data coming to the memory is written sequentially in order not to lose the order of data in an image row. The detailed hardware design of the memory block is given is Figure 4.12.

Four block RAMs are used in memory and all of them are dual-port RAMB16_S18 _S18 which can store 1024 16-bit words of data. Therefore, in order to store 16-bit data to the RAMs, the G value of the left neighbor of the central pixel is concatenated with eight zeros and expanded to 16-bit, represented as 0G in the Figure 4.12. Similarly, the B and R value of the central pixel is also concatenated and represented as BR in the figure. In memory, BR data is taken by one cycle delay, in order to get 0G first to the memory.

The *Select Data* block is a simple multiplexer and selects between the 0G and delayed BR inputs. The select signal of the *Select Data* unit is generated from the *Sequence Generator* block. *Select Data* selects one of the two data inputs in turn (i.e. *order* signal like 010101). *Sequence Generator* takes two inputs as EOL and *orderInfo*. *orderInfo* is the signal that shows the first color in each Bayer Array row. If a row

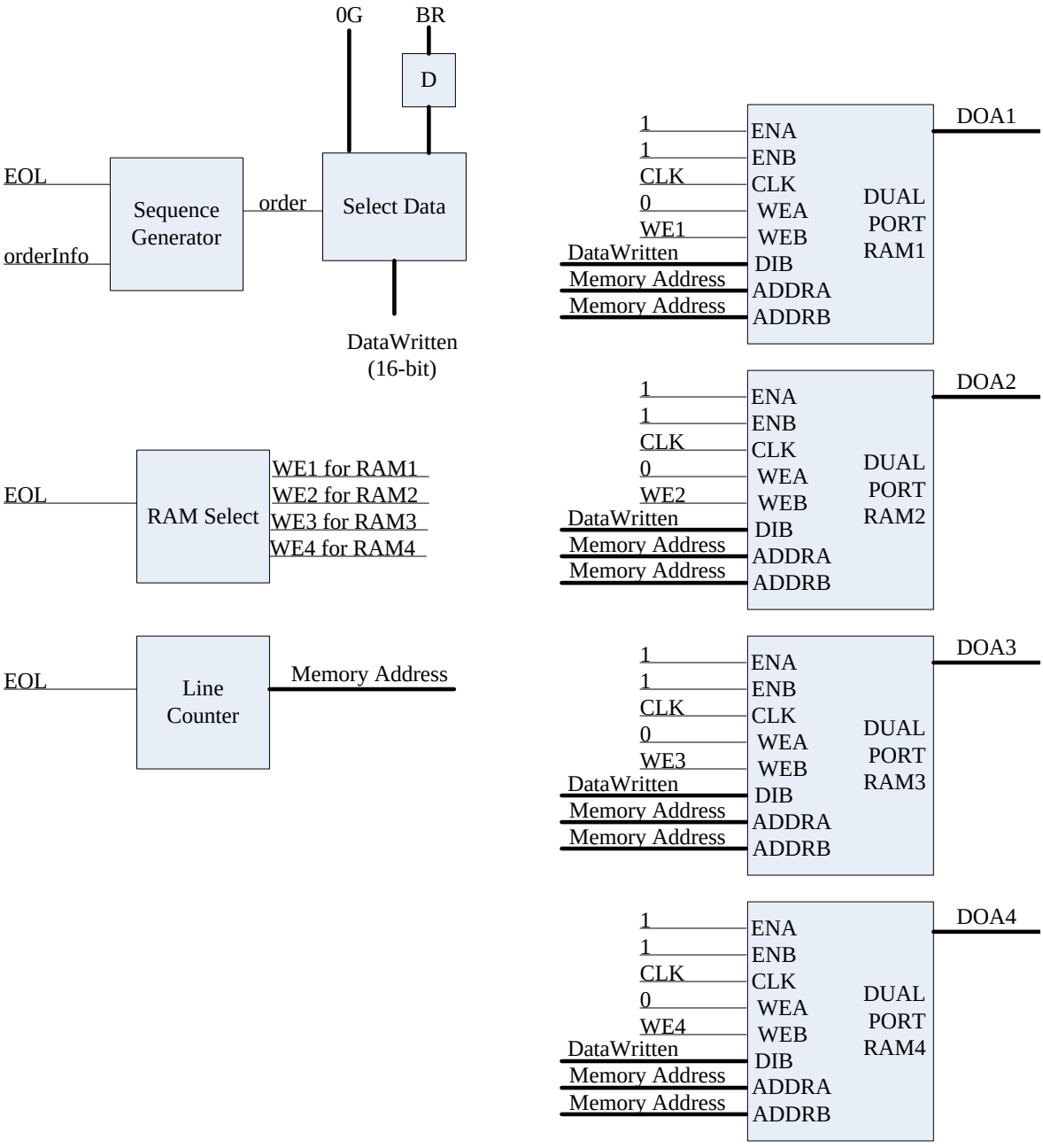

Figure 4.12. The memory block of the design

is starting with a G value the *orderInfo* signal is 1, otherwise it is 0. This is used to determine when to select the 0G or BR inputs. If *orderInfo* is 1, the *Sequence Generator* starts with selecting the 0G input (generates a sequence like 010101), otherwise it starts the sequence by selecting the BR input first (generates a sequence like 101010). The *orderInfo* signal is a chip input signal. EOL signal is used to reset the sequence. When the EOL signal is introduced to the *Sequence Generator* unit, it checks the *orderInfo* signal and restarts the sequence. Due to the characteristic of the Bayer Array, if *orderInfo* signal is 0 in a row, it has to be 1 in the next row. In other words, it toggles when an EOL signal is introduced. As a result, the output of the *Select Data* multiplexer is the data that should be written to the memory and it is connected to the data input ports of the block RAMs.

As discussed in Section 4.2, only four rows of the input image is written to the memory. The four rows of the image are the first four rows of the Figure 4.7. In other words, the operating row of the BW block and the three rows above the operating row are present in the memory. In Figure 4.12, there exists four block RAMs and each of them is responsible for one of the rows that must be written into memory. While BW block is operating on a row, the data is written to one memory and the other three RAMs are emitting the data. The RAM read-write procedure is shown in Figure 4.13.

As shown in Figure 4.13(a), the contents of the 1st, 2nd and 3rd rows of the image is read from memory and sent to SW block. At the same time, the BW block is calculating the missing values for the 4th row and writes to 4th RAM. When the BW block finishes the calculations of missing values for the 4th row, the 1st row will be obsolete and the data for 5th row is written to RAM1 in which the data for the 1st row is stored in Figure 4.13(b). In Figure 4.13(c), the 6th row is introduced and the data for 6th row is written to RAM2 where row2 data is stored.

In Figure 4.12, the *RAM Select* unit is used to generate the write enable signals for the RAMs. *RAM Select* unit is a 2 bit counter and counts up when an EOL signal is introduced. The 2-bit output of the counter is connected to a decoder and the write enable signals are created. Every time an EOL signal is activated, the next RAM is

used for write operations.

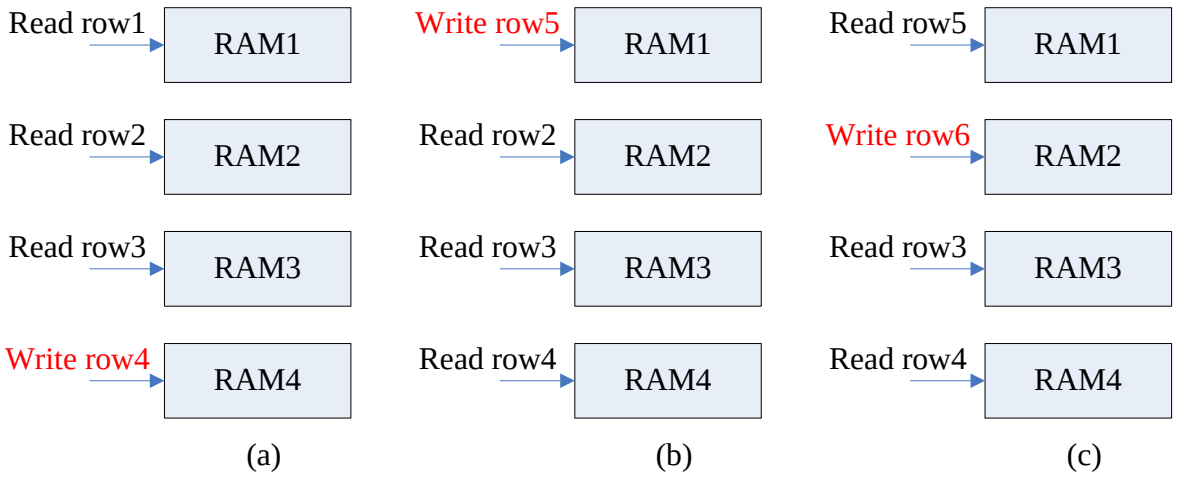

Figure 4.13. (a) Initial RAM contents. RAM4 is used for write and the others are used for read operations. (b) The RAM contents after the first EOL signal. (c) The RAM contents after second EOL signal

The *Line Counter* unit is used to generate the 10-bit memory address. At each clock cycle the address is incremented by one since the data of one pixel are written to memory at each cycle. When an EOL signal is introduced, the counter is reset to zero in order to perform the write operation from the beginning of the RAM.

As mentioned before, all block RAMs in the memory are dual-port RAMs. Both ports of all RAMs always enabled and A ports are always used for read operations (all WEA inputs are connected to 0). B port of the RAM in which the operating row data is written is used for write operations and the B ports of the remaining block RAMs are used again for read operations.

In order to explain how the read operations from the memory is performed, the two states of the memory has to be understood carefully. These two states can be named as 0G-write state in which the 0G input is written and BR-write state in which the BR data are written.

4.5.1. 0G-write State

When the 0G input of the memory block is written to block RAMs, the contents of the block RAMs are shown in Figure 4.14, where it is assumed that the output data of BW are written into RAM4.

In RAM4, the 0G input for the R pixel is being written. At the same time, the 3×3 operation window of SW block is shown as *SW Input*. The only missing colors are R values on B regions and the B values on R regions. This interpolation will be performed by SW block. In 3×3 *SW Input*, the data on corner points (0G data on B region) will not be used in SW block. The center of the 3×3 block and the neighbors on left,right,up and down will be needed in SW block calculations (corresponds to the locations b,x,y,a and c in Figure 4.14, respectively).

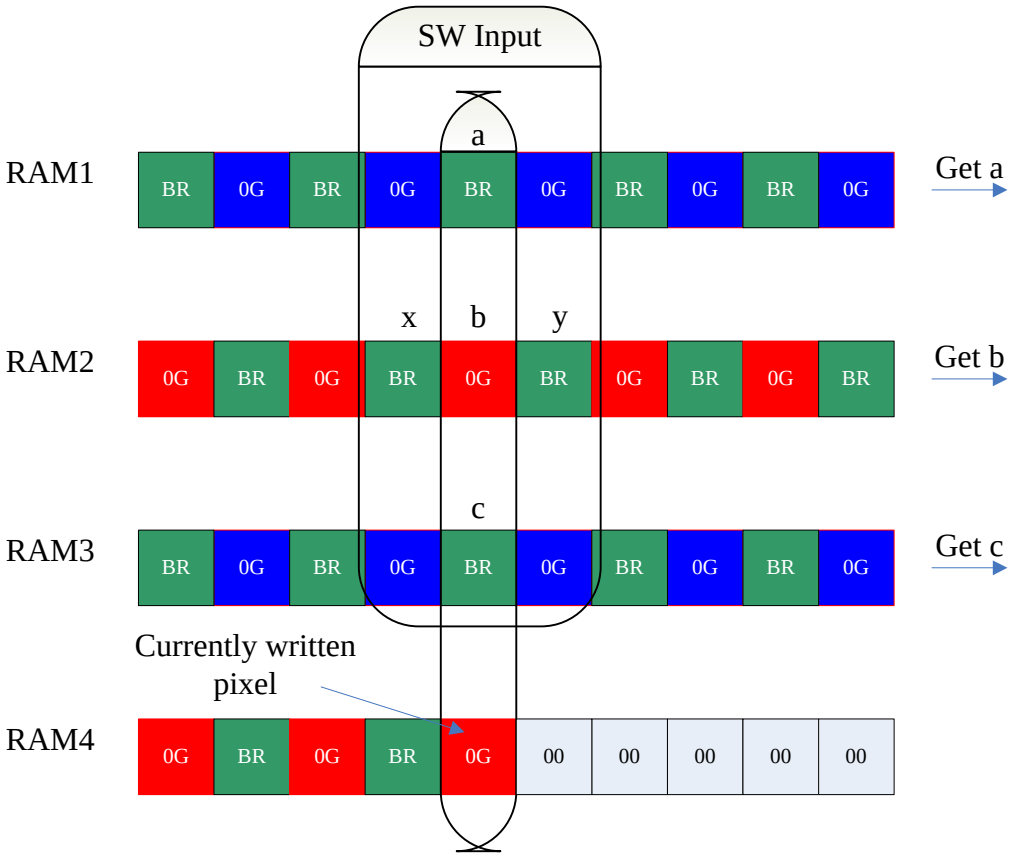

Figure 4.14. Inner sight of the RAMs at 0G-write State

In 0G-write State, the a,b and c values in Figure 4.14 are given to the output ports of the block RAMs. Since the system toggles between 0G-write State and BR-write State, the values at a,b and c locations are emitted from RAM1, RAM2 and RAM3 once per two cycles.

4.5.2. BR-write State

The second state of the memory block is the BR-write State. The contents of the RAMs are shown in BR-write State in Figure 4.15.

In BR-write State, the values at x and y locations are emitted. By using this strategy, the inputs for the SW block are ready after a 0G-write State and a BR-write State. In BR-write State, the B and R values of a G pixel is written into RAM4 as shown in Figure 4.15.

In BR-write State, both x and y values are given from the port A of the same RAM. Since there are two pixels distance between the x and y pixels, the x pixel value can be gathered by adding a two unit delay at port A. When the y value is gathered from port A, the x value can be taken from the signal which is connected to port A by two unit delays in the same clock cycle. The unit delays to take x value are shown in Figure 4.16. The DOA inputs of the figure are the outputs of the RAMs in memory. For each RAM, the DOA output corresponds to y value and the output with two unit delays corresponds to x value.

4.6. Data Synchronization

The data synchronization part of the design in Figure 4.6 is responsible for providing the correct data transmission to the SW block. Since the block RAMs emit data at each cycle, the invalid data should not be sent to the SW. The 3×3 *SW Input* in Figure 4.14 has a C value at the center of the window. If the 3×3 *SW Input* is shifted right by one pixel, the central pixel will be G and the data sent to SW block will be invalid. In the next cycle, the data sent will be again valid. The main purpose of the

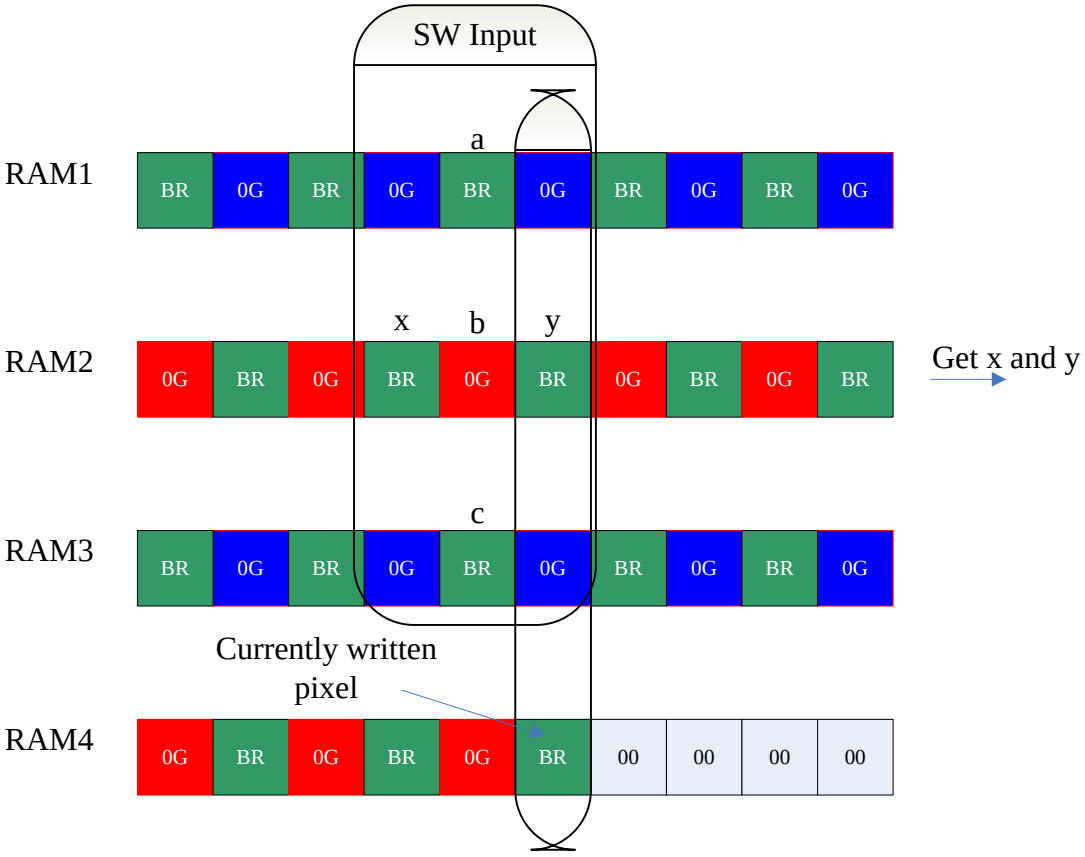

Figure 4.15. Inner sight of the RAMs at BR-write State

data synchronization part is sending data at correct cycles to SW block and separating the 16-bit RAM output data into 8-bit color values (i.e. separate 16-bit BR value into B and R values).

The first part of the data synchronization part is shown in Figure 4.16. The input shown by DOA_i is the output of the A port of the RAM_i where i shows the RAM number from 1 to 4. As discussed in Section 4.5.2, the output from the A port of a RAM is used both without delay and with 2 cycle delay. The value of the signal y_i in Figure 4.16 corresponds to the y value in Figure 4.15. Similarly, the value of the signal x_i corresponds to x value.

In Figure 4.16, there exist z_i signals in addition to x_i and y_i signals. The z_i signals are formed by concatenating the x_i and y_i signals in order to be selected in the

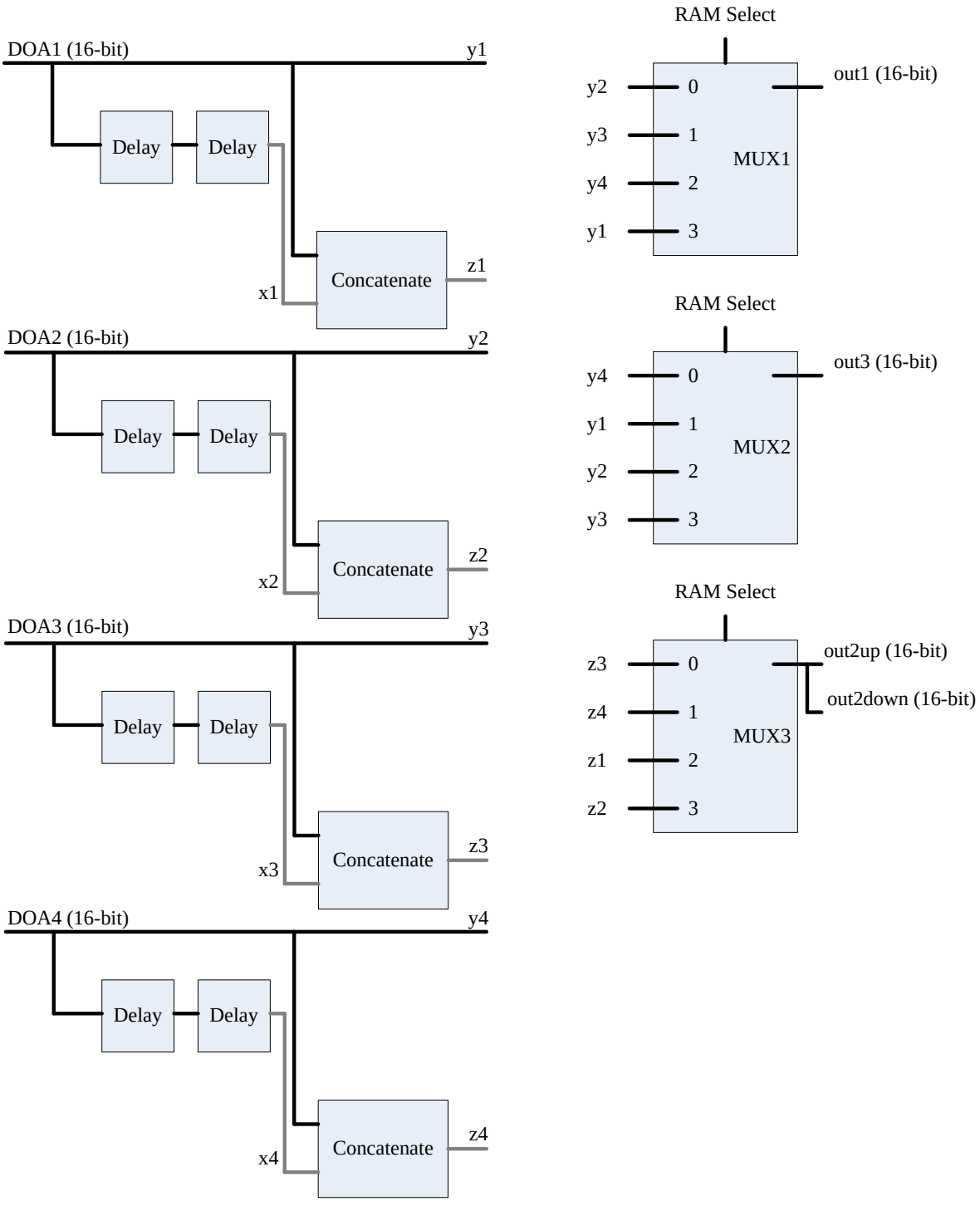

Figure 4.16. Synchronization of block RAM outputs

same multiplexer MUX3. Instead of using one 32-bit MUX3, two 16-bit MUXs could be used. The inputs come from block RAMs for the SW block as shown in Figure 4.17(a). The MUX_i blocks in Figure 4.16 are used to select the five pixel values in Figure 4.17(a) from the memory. *MUX1* block selects a, *MUX2* block selects c values and finally the *MUX3* block selects x, b and y values. In one cycle, the *MUX1*,*MUX3* and *MUX2* select a,b and c inputs, respectively. In the next cycle, *MUX3* selects z signal to generate x and y values. In Figure 4.17(b), the 8-bit inputs for the SW is shown. The a,b,c,x and y values is converted to B_u,G_c,B_d,B_l and B_r respectively.

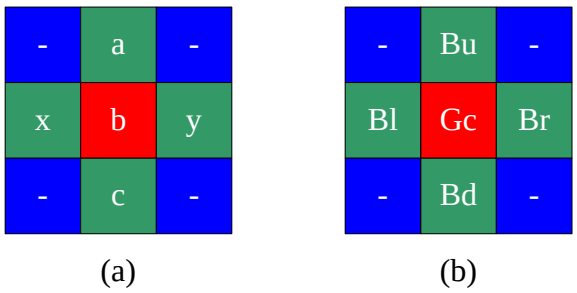

Figure 4.17. (a) Outputs of memory for SW block (b) The inputs of the SW block

The inputs of the multiplexers in Figure 4.16 are organized to select the SW block inputs from the appropriate RAMs. To explain the operation of multiplexers with an example, assume that the operating row of the BW block is written to RAM4 at a current state of the system. Therefore, the inputs for the SW block have to be taken from RAM1, RAM2 and RAM3. The *RAM Select* signal shows the number of the block RAM that is used for write operation. Since RAM4 is used for write operation at the current state, the *RAM Select* signal has the value 3. MUX1 will use the y1 value from RAM1 to generate a value in Figure 4.17(a) and MUX2 will use the y3 value from RAM3 to generate c value. In a similar way, MUX3 will use the z2 value from RAM2 to generate x, b and y values.

At the output of MUX3 in Figure 4.16, the 32-bit signal is decomposed back as *out2up* and *out2down*. Actually, these values corresponds to y and x values in Figure 4.15 and will be used to calculate B_r and B_l values in Figure 4.17, respectively. *out2up* output will also be used for G_c value calculation. The *out2up*, *out2down*, *out1* and

out3 outputs of multiplexers in Figure 4.16 are connected to latches in Figure 4.18.

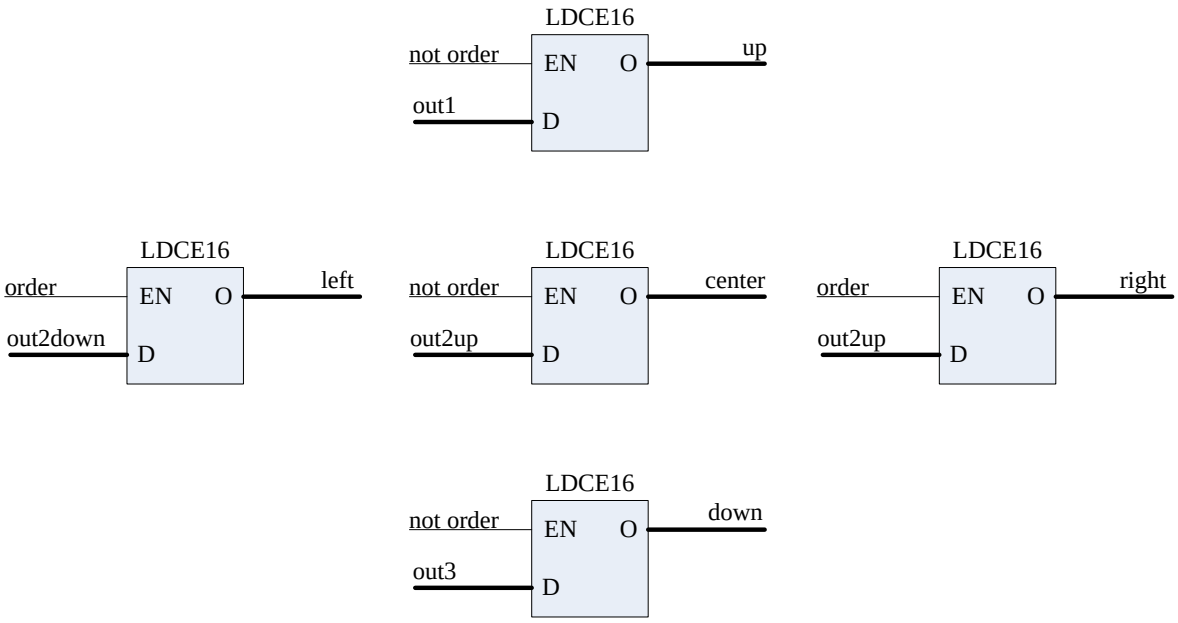

Figure 4.18. Input Generation for Small Window SW block

In Figure 4.18, there are five latches and these are used to transmit the values generated from 0G-write and BR-write states to SW block at correct cycles. The *up*, *center* and *down* outputs are the data produced at 0G-write state and the *left* and *right* outputs are the data given to synchronization unit at BR-write state. The *order* signal connected to enable input of the latches shows the state in which the data, connected to data input of latches, are generated in block RAMs. If the *order* signal is 1, the data are generated at BR-write state. Otherwise, the data are generated at 0G-write state. Therefore the latches sends the correct values to the SW block. After latches send the *up*, *center* and *down* values in one cycle, the *left* and *right* values are sent in the next cycle. However, the *up*, *center* and *down* are still valid in the next cycle and the SW block can get five color values together. At each two cycles, the SW unit takes 5 values from synchronization unit and combine them with the values in buffer and continues its operation.

4.7. Small Window

As discussed in Section 4.2, the Small Window SW block is used the interpolate the missing R values on B pixels and B values on R pixels. The hardware design for the SW block is described in Figure 4.19.

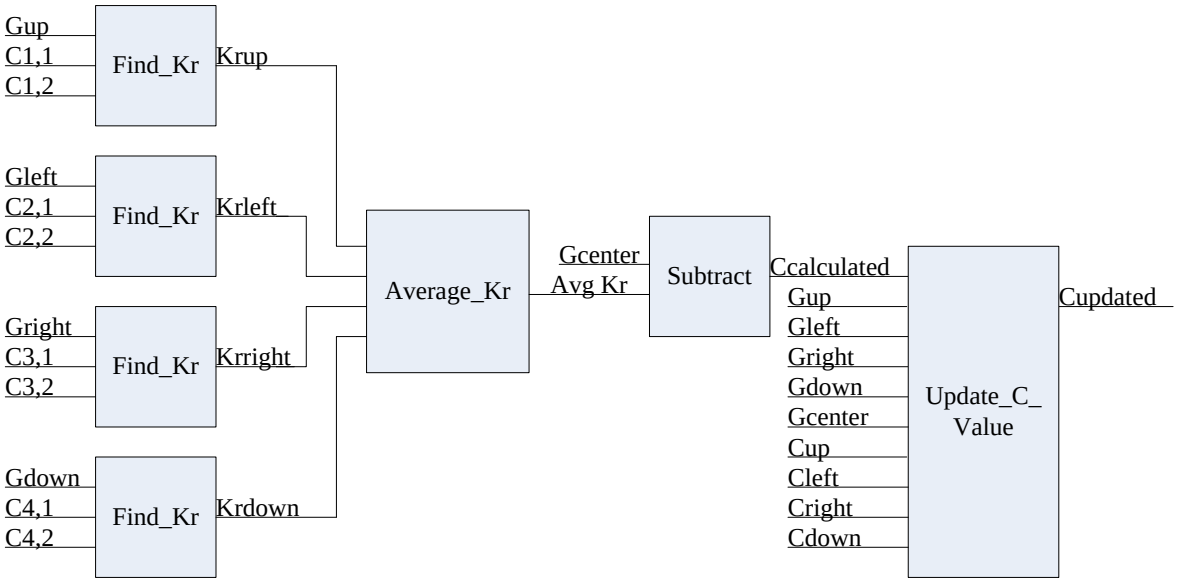

Figure 4.19. The hardware design of Small Window

The input signals for the SW block is gathered from two different blocks. The first one is the buffer in Figure 4.7. The outputs of the registers represented by p_{ij} are connected to the SW block inputs. In addition to the buffer, the data synchronization part of the system sends B_u, B_l, B_r, B_d and G_c values in Figure 4.17. Assume that SW block calculates B value on R pixel (R value on B pixel is performed in similar way). The whole input connection configuration can be summarized as:

- $Gup \rightarrow p_{12}$, $C_{1,1} \rightarrow p_{13}$, $C_{1,2} \rightarrow p_{11}$
- $Gleft \rightarrow p_{23}$, $C_{2,1} \rightarrow p_{13}$, $C_{2,2} \rightarrow p_{33}$
- $Gright \rightarrow p_{21}$, $C_{3,1} \rightarrow p_{11}$, $C_{3,2} \rightarrow p_{31}$
- $Gdown \rightarrow p_{32}$, $C_{4,1} \rightarrow p_{33}$, $C_{4,2} \rightarrow p_{31}$
- $Gcenter \rightarrow G_c$, $C_{up} \rightarrow B_u$, $C_{left} \rightarrow B_l$
- $C_{right} \rightarrow B_r$, $C_{down} \rightarrow B_d$

The B value of the center of 3×3 is calculated in SW block. However, this B value is not updated. In case of an edge, the B value has to be updated. The details of the *Update_C_Value* block in Figure 4.19 is shown in Figure 4.20. The inputs G_{left}, G_{right}, G_{up} and G_{down} are connected to the G values at p_{23}, p_{21}, p_{12} and p_{32} locations in Figure 4.7.

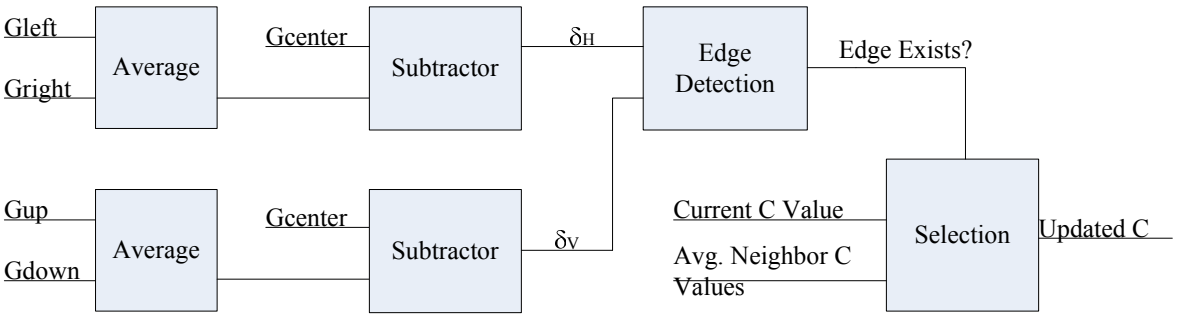

Figure 4.20. The hardware design of updating C value

The update of C value calculated in SW block is performed by referencing the update operations in Section 3.1.

4.8. Output Selection

In Output Selection part of the system, the calculated R, G and B color values or values from input buffer are produced from the output ports of the core. The core has only three 8-bit outputs and each one of them represents the R, G and B colors of a pixel. The output is given by starting from the top left pixel of the image and continues left-to-right and top-to-bottom. The details of the Output Selection unit is given Figure 4.21.

The *RFinal*, *GFinal* and *BFinal* values are the R, G and B values of one pixel that is calculated by the algorithm discussed in Section 3.1. In order to get *RFinal*, *GFinal* and *BFinal* values, three multiplexers are used. The select signal for the multiplexers are connected to a signal called *order* that shows which color of the pixel originally exists in the Bayer Array. If the pixel whose values are being emitted is a G pixel, the *order* signal has the value 0. On the other hand, if the pixel whose values are being

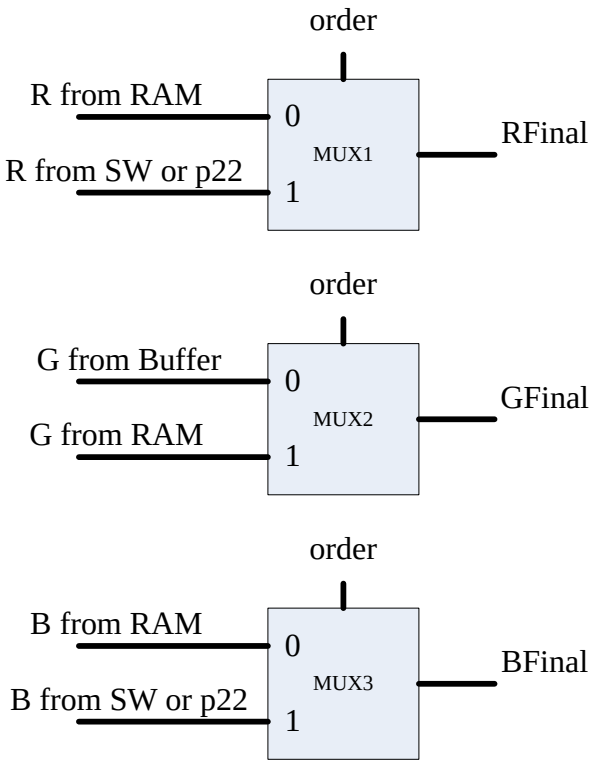

Figure 4.21. The hardware design of Output Selection unit

emitted is a R or B pixel, the *order* signal has the value 1. In case of the *order* signal value is 0, the multiplexers will select the first inputs to emit the R, G and B values of a G pixel in the Bayer Array. As expected, the second multiplexer will select the G value from the buffer since the original G value is given to the core. Recall that the R and B values of the G pixel are calculated by BW block and stored in the RAMs. So, the first and third multiplexers emits the R and B values from the RAMs, respectively. If the *order* signal value is 1, the multiplexers select the second inputs. When the R, G and B values of the R or B pixel is emitted, the G value of the R or B pixel is taken from the RAMs since it has already been calculated by BW block. If the original value of the pixel in the Bayer Array is R, the R value is the p_{22} value in the buffer and the B value is calculated by the SW block. On the other hand, if the original value of the pixel is B, the p_{22} location value corresponds to B value and the R value is calculated by SW block similarly. It is useful to remind that the *order* signal in the Output Selection block is generated from the *Sequence Generator* unit in Figure 4.12, like the *orderInfo* signal in Section 4.5 and the *order* signal in Section 4.6.

4.9. Architectural Modifications for Power Consumption Reduction

The BW block in Section 4.4 generates valid data once at each two clock cycles. When the data at (3,3) location in Figure 4.7 is G, the BW block generates valid data, otherwise the generated data for R or B value at location (3,3) is invalid and garbage. Therefore, feeding the BW block with only valid data makes sense in order to reduce the power consumption. After feeding the valid data, the new invalid data will not be given to the BW block in the next cycle and the previous data will remain the same for 2 cycles.

In order to achieve power reduction in BW block, a new unit is added between the buffer and the BW block which is the *FF Block1* in Figure 4.22. The new block consists of 8-bit registers and each connection from the buffer arrives to BW block with a unit delay. These flip-flops are connected at a half-speed clock in order to transmit the data once in two cycles.

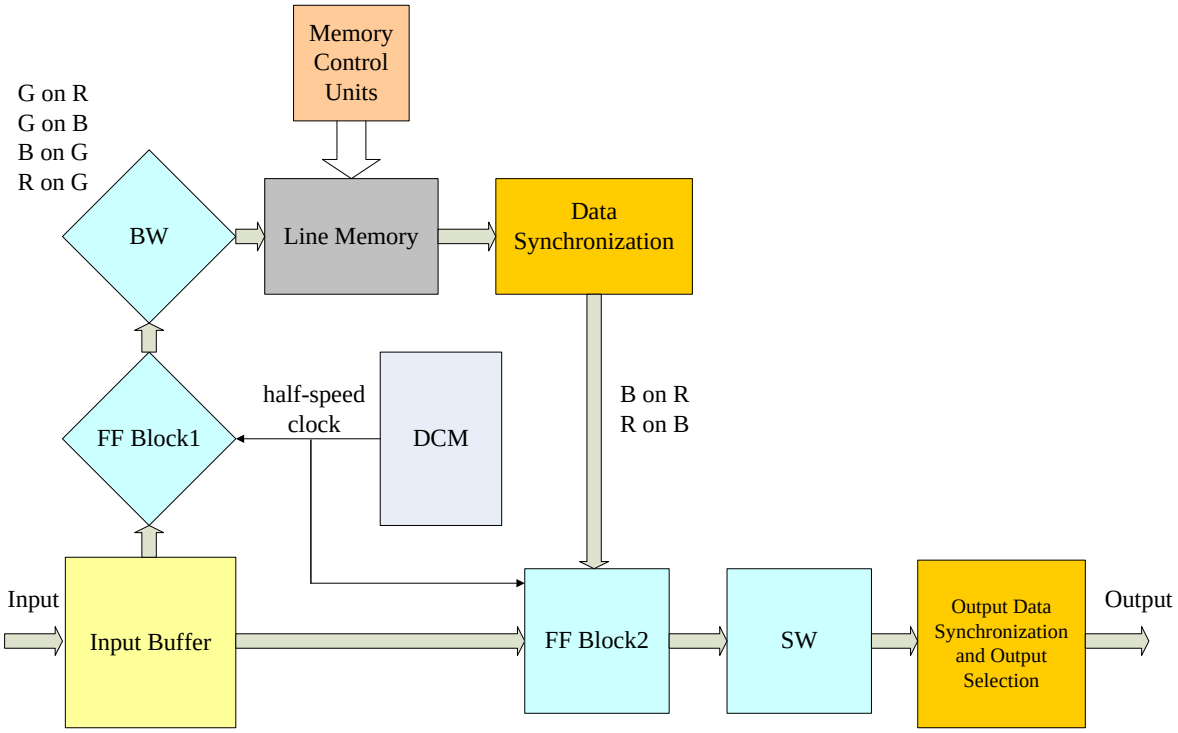

Figure 4.22. The design overview after architectural modification

The half-speed clock is generated by a Digital Clock Manager (DCM) in the

FPGA. The DCM generates two half-speed clocks. The first one is the half speed of the system clock and the other one is the 180 degree shifted version of the half-speed clock. The reason for using a shifted clock is that the G values shift 1 position in the Bayer Array when the operating row is changed to the next row. When an EOL signal is introduced to the system, the operating row is changed, the half-speed clock used by the unit between BW and buffer is changed to the other half-speed clock. In other words, the half-speed clock and the shifted half-speed clock are toggled when an EOL signal is introduced.

The similar strategy is used for the SW block for power reduction. Since SW block generates valid data once in two cycles like BW block, the valid inputs for the SW block remains the same for 2 cycles by using an interconnection unit. The interconnection unit is shown as *FF Block2* in Figure 4.22. Therefore, the SW block generates the same inputs for two cycles and the outputs changes at every two cycles.

4.10. FPGA Implementation

The core is implemented on a Virtex-II 500K gate FPGA. The number of occupied slices in the design is 1,145 and the total number of 4 input LUTs used is 1,321. There are four BlockRAMs and 1 DCM unit in the design. The number of resources in Virtex-II 500K gate device and the utilization of the resources are shown in Table 4.1 in detail. The block RAMs are used in memory block as discussed in Section 4.5 and the DCM is used in order to generate half-speed clock for power optimization purposes. The placement of the implementation on FPGA is shown in Figure 4.23.

The synthesized core works at a clock frequency 25 MHz. Since R, G and B values of one pixel are calculated and produced at each clock cycle, the FPGA can process 1,000,000 pixels at 40ms which corresponds to a 1000×1000 image. The reason for calculating the image size that can be processed in 40ms is to determine the image size for real-time video processing. Since there are 25 images in a 1sec video, one image has to be processed in at most 40ms. So, the core has been successfully verified that it can process 1000×1000 pixels of real-time video sequence.

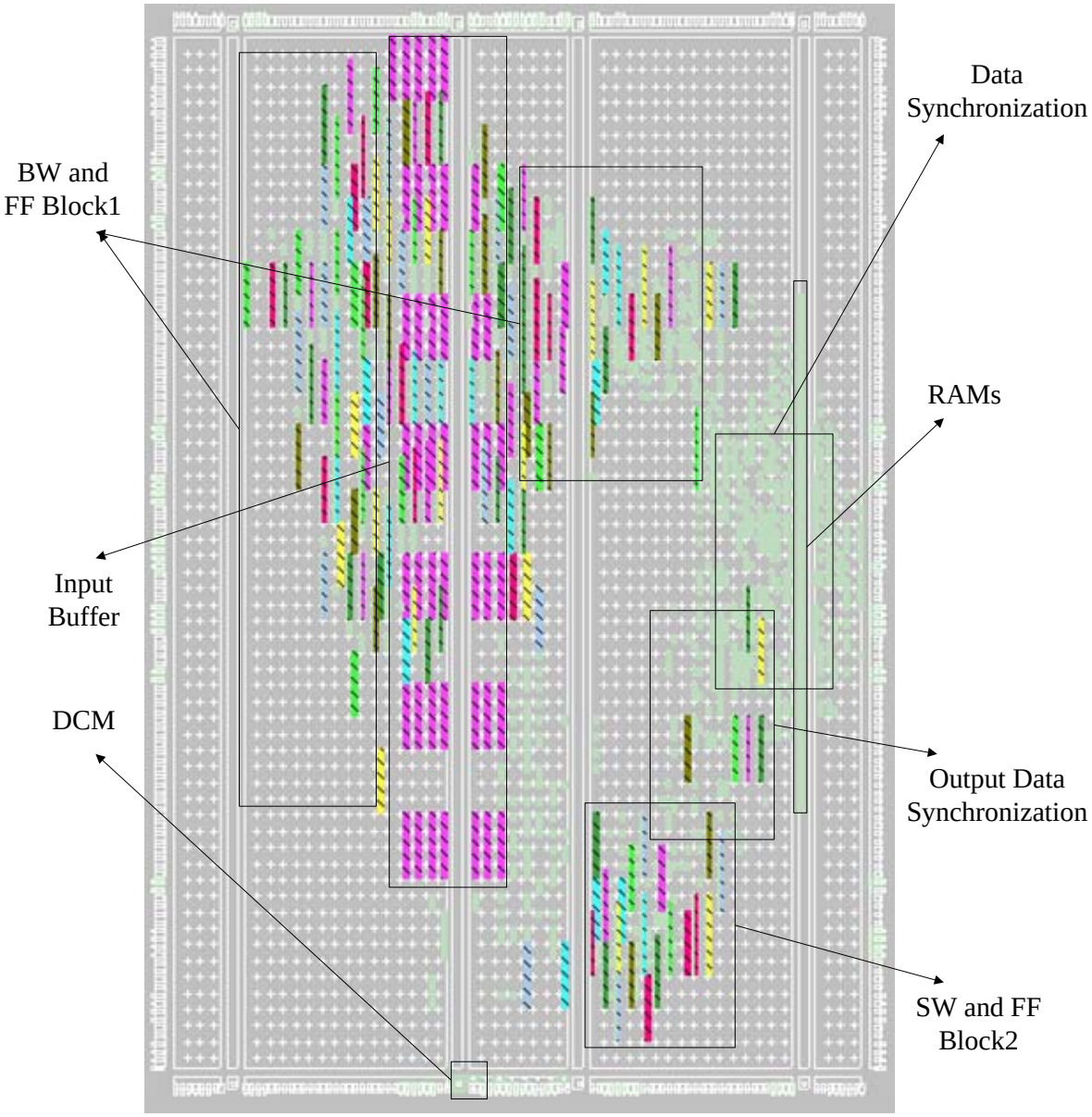

Figure 4.23. The placement of the FPGA implementation

Table 4.1. The utilization of resources in 500K gate Virtex-II device.

	Used	Available	Utilization
Total Number of Slice Registers	1072	6144	17%
Total Number of 4-input LUTs	1321	6144	21%
Number of Occupied Slices	1145	3072	37%
Number of Bonded IOBs	84	172	48%
Number of Block RAMs	4	32	12%
Number of DCMs	1	8	12%

There is a width constraint on the image size in the design which can be at most 1024 pixels. Since one row of the image is stored in a block RAM and the block RAM can store 1024 16-bit data, the width of the image cannot exceed 1024. In order to process wider images, the RAM size has to be increased. This can be done by combining more than one RAM and expanding the address size. If two block RAMs are combined and used, the width of the image can be increased up to 2048 pixels. In this configuration, eight block RAMs are needed in the design.

Another issue is the image construction by using the data coming from the outputs of the FPGA. After starting to feed the FPGA with the pixel values of the Bayer Array, the first R, G and B values of the image are produced after 15 cycles. To construct the full image, the R, G and B values have to be collected after 15 cycles and should be placed in a two dimensional image array. If the image width is known, the values should be placed into next row after the color values are taken in the number of image width while placing the color values. Otherwise, the values should be placed into next row 15 cycles after the EOL signal. Therefore, the full image can be constructed successfully.

Three software tools have been used for the implementation of the core. Xilinx ISE 8.02.03i is used for development, Modelsim SE 6.0a is used for performing simulations and the XPower tool of Xilinx ISE is used for power estimation. All simulations are performed by using simulation resolution of 1ns and 1ps. After the architectural modifications for power reduction, the simulations are performed by using 1ps simula-

tion resolution since the DCM unit can be simulated at most 1ps simulation resolution.

The total power consumption of the initial design is 36899 mW and the dynamic power consumption is 36455 mW. These are the time-based power simulation results. After the architectural modifications, the total and dynamic power consumption have reduced to 11844 mW and 11423 mW, respectively. In other words, 68% reduction in power consumption has been achieved after architectural modification. Even though the power consumption values are still high, it must be kept in mind that the simulation resolution has a direct impact on the power consumption.

Table 4.2. Power consumptions for simulation resolution of 1ps and 1ns.

	Total Power	Dynamic Power
Design in Figure 4.6 with 1ns	1454 mW	996 mW
Design in Figure 4.6 with 1ps	36899 mW	36455 mW
Design in Figure 4.22 with 1ns	not available	not available
Design in Figure 4.22 with 1ps	11844 mW	11423 mW

In Table 4.3, the total power and dynamic power consumptions for the designs before and after the architectural modifications are shown for different simulation resolutions. The power consumption of the design after architectural modification with 1ns simulation resolution is not available since DCM cannot be simulated at this simulation resolution. The difference between the power consumption of the same design for different simulation resolutions is the glitches on the signals. Glitches are unexpected wrong values on signals. Unequal delays in the combinational circuits may cause glitches. It is shown in [36] that high power dissipation is one of the major disadvantages of FPGAs and the main part of the power consumed is caused by glitches. When 1ns simulation resolution is used, the glitches cannot be noticed and the power consumption is lower than the simulation with 1ps simulation resolution.

In the core, the BW and SW blocks are combinational circuits and glitches occur inside these blocks. There are also glitches in Data Synchronization and Output Data Synchronization modules. As [36] suggests, the power consumption surplus that results

from the glitches can be eliminated by using the introduction of staging registers and pipelining. The design after glitch elimination means that the power consumption of the architecture after modification is at most as the design before modification which is 1454 mW. Since the architectural modification reduces the power consumption, the modified design with glitch elimination is expected to have less power consumption than 1454 mW.

The size of the test image used for power calculations is 10×10. The reason for using a small image is the long simulation time and insufficient RAM size of the computer (The computer used in simulations has Intel Core2Duo 2.13 GHz processor and 3.00 GB of RAM). In order to simulate a 10×10 mosaicked image, 400 MB of storage is required and the simulation lasts 35 minutes. In addition, simulations at 1ps simulation resolution for 36×36 and 96×96 images generate 4.712 GB and 34 GB **.vcd* file, respectively. These **.vcd* files cannot be processed by Xilinx Xpower because of the lack of RAM. In Figure 4.24, the total power consumption results are shown for different image sizes. The figure shows that the total power consumption slightly changes after 10×10 image. Therefore, it is safe to use the results in Table 4.3 in which 10×10 image size is used.

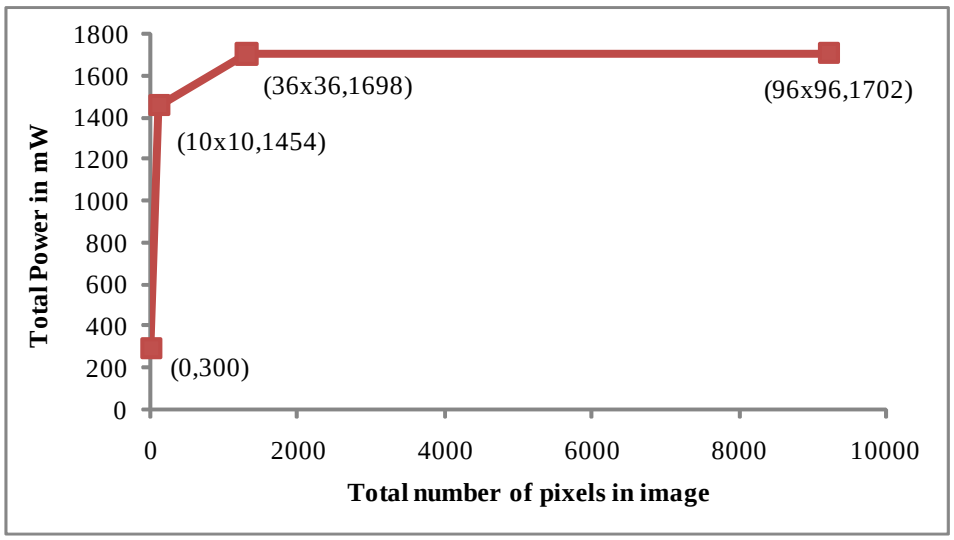

Figure 4.24. The total power consumption for images with different sizes. Data labels are organized as (image size, power consumption in mW)

As mentioned in the beginning of the chapter, the core is implemented by using VHDL and the FPGA mapping is just used for verification of the core. In order to use the implementation with lower power consumption than FPGA mapping, the Application Specific Integrated Circuits (ASIC) can be used. In [37], the power consumption of the same implementation can be reduced by 7.1-14 times by using ASIC when compared to the power consumption on FPGA. Table 4.3 is taken from the study in [37] and shows the power consumption ratios of the same designs (FPGA over ASIC).

As [37] shows, the power consumption of the design that is composed of only memory and logic units can be reduced by 14 times. Therefore, the total power consumption of the proposed core can be reduced approximately to 846 mW by using ASIC without eliminating the glitches. If both the glitches eliminated and the ASIC is used, the total power consumption is expected to be under 100 mW.

Table 4.3. Dynamic power consumption ratio (FPGA/ASIC) taken from [37].

	Logic Only	Logic &DSP	Logic &Memory	Logic, Memory &DSP
booth	26			
rs_encoder	52			
cordic18	6.3			
cordic8	5.7			
des_area	27			
des_perf	9.3			
fir_restruct	9.6			
mac1	19			
aes192	12			
fir3	12	7.5		
diffeq	15	12		
diffeq2	16	12		
molecular	15	16		
rs_decoder1	13	11		
rs_decoder2	11	11		
atm			15	
aes			13	
aes_inv			12	
ethernet			16	
serialproc			16	
fir24				5.3
pipe5proc				8.2
raytracer				8.3
GeoMean	14	12	14	7.1

5. CONCLUSION

In this study, a low-cost edge-detection and edge-directed interpolation mechanism has been proposed. This is used in conjunction with ECI during demosaicking. It detects difficult texture regions that are overlooked by ECI. As a result, the visual quality of the demosaicked image is improved.

The ECI algorithm is selected as a partner algorithm in this thesis, because it is one of the simplest demosaicking solutions and it can produce good demosaicking results while being relatively easy to implement on hardware. However, the proposed mechanism can also be used with other methods in the literature. In difficult textured regions, ECI+OP can achieve 70% improvement over ECI. In general, this is around 10%. However, if there are no or very few difficult regions, then achievement of ECI+OP is very close to that of the regular ECI algorithm, as expected.

The overall cost of the proposed method is lower than the cost of the other low-cost edge-adaptive algorithms. The proposed solution produces better results in the difficult textured regions. The performance of the proposed algorithm is not compared with sophisticated and complex algorithms because it is accepted that those algorithms can produce better results at the expense of very high computational and storage costs.

The proposed algorithm is implemented in VHDL. Verification of the core has been done on Virtex-II 500K gate FPGA. The implementation can process 1000×1000 real-time video and $1000 \times n$ digital still images. Since the interpolations for the R, G and B values are combined in a small window, the memory requirement to store the image data is significantly reduced. By using the power consumption reduction strategies, the power consumption is reduced up to 68% compared to the design before architectural modification.

The ECI algorithm is used for demosaicking process for the smooth regions and the proposed algorithm is used for detection of one-pixel patterns. The optimum thresh-

old value α is used as a decision parameter in edge detection and can be determined by a new mechanism. The ECI algorithm and the proposed algorithm used together in this thesis and new algorithms can be tried for smooth regions instead of ECI algorithm in the future studies. The power consumption of the core can be reduced by pipelining or staging registers in order to map on the FPGA as a future work.

APPENDIX A: L^*, a^*, b^* Color Space

L^*, a^*, b^* color space is a color-opponent space based on nonlinearly-compressed CIE X, Y, Z color space coordinates. L^* represents lightness (L^*=0 means black and L^*=100 gives diffuse white) and a^* (negative a^* values indicate green and positive a^* values correspond magenta) and b^* (negative b^* values indicate blue and positive b^* values correspond yellow) represent the color-opponent dimensions [38]. CIE L^*, a^*, b^* (CIELAB) is specified by the International Commission on Illumination (*Commission Internationale d'Eclairage*).

In order to calculate L^*, a^*, b^* values for a pixel, the X, Y and Z values in CIE X, Y, Z color space should be calculated by using the R, G and B values firstly [35]. The linear transformation relating R, G, B to X, Y, Z is given below:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.490 & 0.310 & 0.200 \\ 0.177 & 0.813 & 0.011 \\ 0.000 & 0.010 & 0.990 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{A.1}$$

The L^*, a^*, b^* values are given below:

$$L^* = 25(100Y/Y_0)^{1/3} - 16 \tag{A.2}$$

$$a^* = 500[(X/X_0)^{1/3} - (Y/Y_0)^{1/3}] \tag{A.3}$$

$$b^* = 200[(Y/Y_0)^{1/3} - (Z/Z_0)^{1/3}] \tag{A.4}$$

where X_0, Y_0 and Z_0 are tristimulus values of the reference white. In Equation A.5, CIE L^*, a^*, b^* color-difference formula is shown. ΔL^*, Δa^*, Δb^* are the difference of L^*, a^*, b^* values between the original images and the demosaicked images. In order to calculate ΔE^*_{ab} value, the sum of Δs values for all pixels is divided into the number of

pixels in the image.

$$(\Delta s)^2 = (\Delta L^*)^2 + (\Delta a^*)^2 + (\Delta b^*)^2 \tag{A.5}$$

REFERENCES

1. Lukac, R. and K. N. Plataniotis, "Color filter arrays: Design and performance analysis", *IEEE Transactions on Consumer Electronics*, vol. 51, no. 4, pp. 1260-1267, November 2005.

2. Savard, J., "Color Filter Array Designs", `http://www.quadibloc.com/other/cfaint.htm`, June 2007.

3. Bayer, B.E., "Color imaging array", *U.S. Patent*, no. 3,971,065, 1976.

4. Sakamoto, T., C. Nakanishi and T. Hase, "Software pixel interpolation for digital still cameras suitable for a 32-bit MCU", *IEEE Transactions on Consumer Electronics*, vol. 44, no. 4, pp. 1342-1352, November 1998.

5. Güntürk, B.K., J. Glotzbach, Y. Altunbaşak, R.W. Schaffer and R.M. Murserau, "Demosaicking: color filter array interpolation", *IEEE Signal Processing Magazine*, vol. 22, no. 1, pp. 44-54, January 2005.

6. Ramanath, R. and W.E. Snyder, " Adaptive demosaicking", *Journal of Electronic Imaging*, vol. 12, no. 4, pp. 633-642, October 2003.

7. Kimmel, R., "Demosaicking: Image reconstruction from color CCD samples", *IEEE Transactions in Image Processing*, vol. 8, no. 9, pp. 1221-1228, September 1999.

8. Pei, S.C. and I.K. Tam, "Effective color interpolation in CCD color filter array using signal correlation", *IEEE Transactions on Circuits and System for Video Technology*, vol. 13, no. 6, pp. 503-513, June 2003.

9. Hamilton, J. and J. Adams, "Adaptive color plane interpolation in single sensor color electronic camera", *U.S. Patent*, no. 5,652,621, 1997.

10. Li, X., "Demosaicing by successive approximation", *IEEE Transactions on Image Processing*, vol. 14, no. 3, pp. 370-379, March 2005.

11. Cok, D.R., "Signal processing method and apparatus for producing interpolated chrominance values in a sampled color image signal", *U.S. Patent*, no. 4,642,678, 1986.

12. Adams, J., "Interactions between color plane interpolation and other image processing functions in electronic photography", *Proceedings of SPIE*, vol. 2416, pp. 144-151, 1995.

13. Zhang, L. and X. Wu, "Color demosaicking via directional linear minimum mean square-error interpolation", *IEEE Transactions on Image Processing*, vol. 14, no. 12, pp. 2167-2178, December 2005.

14. Wu, X. and N. Zhang, "Primary-consistent soft-decision color demosaicking for digital cameras", *IEEE Transactions on Image Processing*, vol. 13, no. 9, pp. 1263-1274, September 2004.

15. Chung, K.H. and Y.E. Chan, "Color Demosaicing Using Variance of Color Differences", *IEEE Transactions on Image Processing*, vol. 15, no. 10, pp. 2944-2955, October 2006.

16. Hibbard, R.H., "Apparatus and method for adaptively interpolating a full color image utilizing luminance gradients", *U.S. Patent*, no. 5,382,976, 1995.

17. Laroche, C.A. and M.A. Prescott, "Apparatus and method for adaptively interpolating a full color image utilizing chrominance gradients", *U.S. Patent*, no. 5,373,322, December 1994.

18. Nilsoon, A.M.I. and P.V.W. Nordblom, "Weighted gradient based color interpolation for color filter array", *EP 1,439,715 A1*, 2004.

19. Menon, D., S. Andriani and G. Calvagno, "Demosaicking with directional filtering

and a posteriori decision", *IEEE Transactions on Image Processing*, vol. 16, no. 1, pp. 132-141, January 2007.

20. Muresan, D.D. and T.W. Parks, "Demosaicing using optimal recovery", *IEEE Transactions on Image Processing*, vol. 14, no. 2, pp. 267-278, February 2005.

21. Lukac, R. and K.N. Plataniotis, "Normalized color-ratio modeling for CFA interpolation", *IEEE Transactions on Consumer Electronics*, vol. 50, no. 2, pp. 737-745, May 2004.

22. Güntürk, B.K., Y. Altunbaşak and R.M. Mersereau, "Color plane interpolation using alternating projections", *IEEE Transactions on Image Processing*, vol. 11, no. 9, pp. 9971013, September 2002.

23. Chang, L. and Y.P. Tan, "Efective use of spatial and spectral correlations for color filter array demosaicking", *IEEE Transactions on Consumer Electronics*, vol. 50, no. 1, pp. 355-365, February 2004.

24. Lu, W. and Y.P. Tan, "Color filter array demosaicking: New method and performance measures", *IEEE Transactions on Image Processing*, vol. 12, no. 10, pp. 1194-1205, October 2003.

25. Lukac, R., K. Martin and K.N. Plataniotis, "Demosaicked image post-processing using local color ratios", *IEEE Transactions on Circuit and Systems for Video Technology*, vol. 14, no. 6, pp. 914-920, June 2004.

26. Adams, J., "Design of practical color filter array interpolation algorithms for digital cameras", *Proceedings of SPIE*, vol. 3028, pp. 117-125, February 1997.

27. Lukac, R. and K.N. Plataniotis, "Data-Adaptive Filters for Demosaicking: A Framework", *IEEE Transactions on Consumer Electronics*, vol. 51, no. 2, pp. 560-570, May 2005.

28. Lukac, R., K.N. Plataniotis, D. Hatzinakos and M. Aleksic, "A Novel Cost Effective

Demosaicing Approach", *IEEE Transactions on Consumer Electronics*, vol. 50, no. 1, pp. 256-261, February 2004.

29. Freeman, T.W., "Median filter for reconstructing missing color samples", *U.S. Patent*, no. 4,724,395, 1988.

30. Fischer, M., J.L. Paredes and G.R. Arce, "Weighted median image sharpeners for the world wide web", *IEEE Transactions on Image Processing*, vol. 11, pp. 717-727, 2002.

31. Glotzbach, J.W., R.W. Schafer and K. Illgner, "A method for color filter array interpolation with alias cancellation properties", *Proceedings of IEEE International Conference on Image Processing*, vol. 1, pp. 141-144, 2001.

32. Hirakawa, K. and T.W. Parks, "Joint Demosaicing and Denoising", *IEEE Transactions on Image Processing*, vol. 15, no. 8, pp. 2146-2157, August 2006.

33. Lian, N., L. Chang and Y.P. Tan, "Improved CFA demosaicking by accurate luminance estimation", *Proceedings of ICIP'95*, vol. 1, pp. 41-44, September 2005.

34. Lee, W. and J. Kim, "A cost-effective demosaicked image enhancement for a single chip CMOS image sensor", *Proceedings of SIPS05*, pp. 148-153, 2005.

35. Jain, A.K., *Fundamentals of Digital Image Processing*, *Prentice-Hall Inc.*, New Jersey, USA, 1989.

36. Fischer, R., K. Buchenrieder and U. Nageldinger, "Reducing the Power Consumption of FPGAs through Retiming", *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS05)*, 2005.

37. Kuon, I. and J. Rose, "Measuring the Gap Between FPGAs and ASICs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, February 2007.

38. Wikipedia, "Lab Color Space", http://en.wikipedia.org/wiki/CIELAB, May 2008.