EFFICIENT REALTIME IMAGE SCALING AND WARPING IN HARDWARE

by

Mert Büyükmıhçı

Submitted to Graduate School of Natural and Applied Sciences

in Partial Fulfilment of the Requirements

for the Degree of Master of Science in

Computer Engineering

Yeditepe University

2017

ii

EFFICIENT REALTIME IMAGE SCALING AND WARPING IN HARDWARE

ACKNOWLEGEMENTS

This thesis work has been done as part of a project jointly supported by TÜBİTAK ARDEB-EEEAG and European Union's Artemis Joint Undertaking. On TÜBİTAK side, the project is called "M-RIVA: Methodology development for Real-time Implementation of Video Algorithms on FPGAs" and has been designated project no. 114E343. On Artemis side, the project is part of a bigger project called "ALMARVI: Algorithms, Design Methods, and Many Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing" and has been designated project no. 621439. ALMARVI project (http://almarvi.eu) is run by a consortium of 16 universities and companies from Netherlands, Finland, Czech Republic, and Turkey, which include Philips, Nokia, Aselsan, and Özyeğin University. I was supported by TÜBİTAK (under project no. 114E343) between April 2014 and May 2016.

It is with immense gratitude that I acknowledge the support and help of Assoc. Prof. Sezer Gören Uğurdağ and Assof. Prof. H. Fatih Uğurdağ. Pursuing my thesis under their supervision has been an experience which broadens the mind and presents an unlimited source of learning.

Finally, I would like to thank my family for their endless love and support, which makes everything more beautiful.

ABSTRACT

EFFICIENT REALTIME IMAGE SCALING AND WARPING IN HARDWARE

Downscaling and warping are found in many image/video processing applications. This thesis offers an area-efficient downscaler hardware architecture and an implementation of a warping algorithm on hardware. The proposed downscaler is called “Output Domain Downscaler (ODD)”. Both warping and ODD are demonstrated based on the implementation of bilinear interpolation method. Same interpolation method used in a different setting caused the difference between downscaler and warping implementations. Memory read and write methods of both warping and downscaler are also implemented with a single general purpose FIFO. FIFO size calculation tool and a scheduler tool were used when implementing warping unit. Output domain downscaler is also combined with edge detection and sharpening spatial filter. This thesis compares ODD to a straight-forward implementation of the same combination of downscaling methods, which is called “Input Domain Downscaler (IDD)”. IDD tries to output a new pixel of the downscaled video frame every time a new pixel of the original video frame is received. However, every once in a while, there is no downscaled pixel to produce. IDD sometimes also skips a complete row of input pixels. ODD, on the other hand, spreads out the job of producing downscaled pixels almost uniformly over a frame. As a result of that, output domain downscaler is able to do more resource sharing, i.e., can do the same job with fewer arithmetic units, thus offers a more area-efficient solution than input domain downscaler. In this thesis, output domain downscaler architecture is implemented with a downscale ratio between 1 and 2 with no loss of generality. That is because it is best to achieve larger downscale ratios of bilinear interpolation by applying a downscale ratio between 1 and 2 multiple times.

ÖZET

DONANIM ÜZERİNDE ETKİLİ GERÇEK ZAMANLI GÖRÜNTÜ ÖLÇEKLEME VE BÜKME

Birçok görüntü/video işleme uygulamasında boyut küçültme ve bükme bulunur. Bu tez, alan etkili bir boyut küçültücü donanım mimarisi ve donanıma bükme algoritması uygulaması sunacak. Sunulan algoritmanın adı “ODD (Output Domain Dowscaler)” dır. Hem bükme hem de ODD çift doğrusal ara değerleme yöntemi uygulamasına dayanarak gösterilmiştir. Aynı ara değerleme metodunun farklı şekillerde gerçeklenmesi boyut küçültme ve bükme üniteleri arasındaki farkı oluşturur. bükme ve küçültme ünitelerinin hafızadan okuma ve yazma mantıkları benzer ve genel bir FIFO ile sağlanımştır. Bükme ünitesi gerçeklenirken FIFO boyutu hesaplayıcı ve planlama oluşturucu araçlar kullanılmıştır. ODD, aynı zamanda ayrıt sezimi ve keskinleştirilmiş uzamsal süzgeç ile birleştirilmiştir. Bu tez, ODD ı aynı birleşme yöntemlerinin direk uygulayan ve adına “Input Domain Downscaler” dediğim, yöntem ile karşılaştıracak. İnput domain downscaler, orijinal video karesinin yeni bir pikseli her alındığında küçültülmüş video karesinin yeni bir pikselini çıkarmaya çalışır. Bununla birlikte, arada sırada, üretecek küçültülmüş piksel olmaz. İnput domain downscaler, ayrıca, bazen girdi piksellerinin bir satırının tamamını atlar. Öte yandan, ODD, küçüktülmüş pikselleri bulma işini bütün frame e yayar. Bunun bir sonucu olarak, ODD daha çok kaynak paylaşımı yapabilir, örneğin; aynı işi daha az aritmetik birim ile yapar, böylece İnput domain downscaler dan daha iyi alan etkili bir çözüm sunar. Bu tez, ODD mimarisini, genel özellik kaybı olmaksızın 1 ile 2 arasında bir oran ile uygulayacak. Bunun nedeni çift doğrusal ara değerlemenin daha büyük boyut küçültme oranları elde etmek için birçok kez 1 ve 2 arasında küçültme oranı uygulamak en iyi yöntemdir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Image scaling and warping algorithms are widely used in various areas like computer vision [1], medical devices [2], online videos [3], and image zooming [4]. Due to extensive usage, efficient and low-cost implementations of image scaling and warping algorithms are crucial. Image scaling can be in the form of either downscale or upscale. This thesis will focus on downscaling and warping algorithms and their implementations.

1.1. BILINEAR INTERPOLATION

Bilinear interpolation is a low cost algorithm which makes interpolation calculations on functions of two variables on a rectilinear 2D grid [11]. The idea is to perform linear interpolation in *x* dimension, and then performing it in *y* dimension. Each step is linear but the overall interpolation is quadratic rather than linear.

A simple 2D grid example is shown in Figure 1 there are four pixels shown in Figure 1.1 $P(m,n)$, $P(m+1,n)$, $P(m+1,n)$, and $P(m+1,n+1)$, two interpolated pixels on the sides $P(m,i)$ and $P(m+1,i)$, and one pixel between the two interpolated pixels $P(j,i)$, which is the downscaled pixel.

Note *m* denotes the column, *n* denotes the row index in 2D grid. The interpolated pixels are calculated from Equation 1.1 and 1.2 (simplified to Equation 1.4 and 1.5), whereas the downscaled pixel is calculated from Equation 2.3 (simplified to Equation 1.6). In Figure 1.1, the distance between i and n (*y* dimension components of pixels) is shown as *dy* and the distance between *m* and *j* (*x* dimension components of pixels) is shown as *dx*.

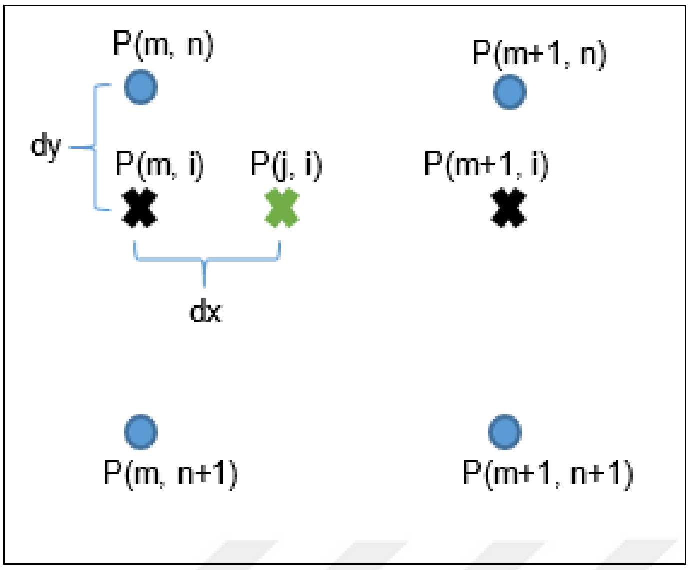


Figure 1.1. Simple four pixel grid for bilinear interpolation

$$P(m,i) = \frac{(n+1)-i}{(n+1)-n} \times P(m,n) + \frac{i-n}{(n+1)-n} \times P(m,n+1) \tag{1.1}$$

$$\mathrm{P}(\mathrm{m}+1,\mathrm{i}) = \frac{(\mathrm{n}+1)-\mathrm{i}}{(\mathrm{n}+1)-\mathrm{n}} \times \mathrm{P}(\mathrm{m}+1,\mathrm{n}) + \frac{\mathrm{i}-\mathrm{n}}{(\mathrm{n}+1)-\mathrm{n}} \times \mathrm{P}(\mathrm{m}+1,\mathrm{n}+1) \tag{1.2}$$

$$P(j,i) = \frac{(m+1)-j}{(m+1)-m} \times P(m,i) + \frac{j-m}{(m+1)-m} \times P(m+1,i) \tag{1.3}$$

$$P(m,i) = \big((n+1)-i\big) \times P(m,n) + (i-n) \times P(m,n+1) \tag{1.4}$$

$$P(m+1,i) = \big((n+1)-i\big) \times P(m+1,n) + (i-n) \times P(m+1,n+1) \tag{1.5}$$

$$P(j,i) = \big((m+1)-j\big) \times P(m,i) + (j-m) \times P(m+1,i) \tag{1.6}$$

Eq. 1.6 also rearranged in order to minimize the arithmetic operations. The final formula was shown in Eq. 1.7. This rearrangement in addition to the simplification reduced the total arithmetic calculations from 18 down to 9.

$$\begin{aligned} P(j,i) = \Bigg(\bigg(\Big(\big(\left(P(m+1,n+1) - P(m+1,n)\right) \times i\big) + P(m+1,n)\Big) \\ - \Big(\big(\left(P(m,n+1) - P(m,n)\right) \times i\big) + P(m,n)\Big)\bigg) \times j\Bigg) \\ + A \end{aligned} \tag{1.7}$$

where $A = \big(\left(P(m,n+1) - P(m,n)\right) \times i\big) + P(m,n)$.

1.2. RELATED WORK

Downscaling is to find the downscaled pixels in an image or a frame according to the respected ratio. The simplest downscaler in the literature is the Nearest Neighbor method (NN) [5,6]. Nearest neighbor method is more area-efficient and easier to implement as compared to bicubic Interpolation (BcubI) [7] and Adaptable K-Nearest (AKN) [8] methods. However, the drawback of nearest neighbor method is that the resulting image/frame contains blocking and aliasing artifacts. Bilinear Interpolation method (BlinI) [9,10] is, on the other hand, has a lower image quality than bicubic interpolation, but can handle the blocking and aliasing artifacts. Bilinear interpolation is also simpler and easier to implement than bicubic interpolation. Although, bicubic interpolation can produce high quality images and handle the aliasing issues very well, because of its complexity and memory requirements, its implementation is difficult and costly.

Chen [11] proposed the use of both an edge detection algorithm and Sharpening Spatial Filter (SSF) to prevent information loss caused by bilinear interpolation in order to realize a downscaler. Incorporating edge detection to bilinear interpolation enables the analysis of local characteristics of pixels such that it can be determined whether there is a non-homogenous color distribution or not. If there is, with the help of sharpening spatial filter, the color characteristics of pixels is enhanced by considering three closest neighbor pixels, acting like a high-pass filter [12]. This filter requires a three-line buffer to implement which is technically costly in terms of memory. This issue was circumvented by using a simple version of sharpening spatial filter [11].

This thesis also includes an implementation of an image warping algorithm. Both downscaling and warping were used in an algorithm called optical flow in order to show the real world applications of both warping and downscaling.

1.3. CONTRIBUTIONS OF THE THESIS

In this Thesis, a low cost, low memory downscale method called ODD and an implementation of a warping algorithm was presented. Downscale method includes edge detection system and a sharpening spatial filter with it. ODD method is superior to traditional downscale methods in terms of register and LUT type areas. Presented method gains in the order of 48% register type area and 21% LUT type area. Warping unit was implemented by using a high level synthesis tool in order to reduce the design time and complexity. Usage of this tool reduced the complexity and increased the design speed. Reducing design compexity allowed the usage of floating point units instead of fixed point units, thus increasing precision. A general purpose FIFO was used for both downscaling and warping implementations. Another tool was used to calculate the FIFO size.

2. DOWNSCALING

This thesis proposes a novel downscaler which also combines bilinear interpolation with an edge detection algorithm and sharpening spatial filter, but in a more area-efficient way. The proposed downscaler in this thesis is called as “ODD” (Output Domain Dowscaler) and the classical downscaler [11] called as “Input Domain Downscaler” (IDD). Proposed method reduces the arithmetic units by rearranging bilinear interpolation pixel equations. Furthermore, this thesis introduces a register bank which reduces the number of reads from the line buffers.

For the evaluation purposes, a development kit which included a Virtex-7 FPGA was used to implement both ODD and input domain downscaler with a frame rate of 90 frame per second (FPS) and a resolution of 1920x1080.

2.1. EDGE DETECTION

Bilinear interpolation can cause information loss since it only takes the average weight of the four closest pixels. In order to prevent the loss, this thesis used a linear sigmoidal edge detecting technique [13,14]. The choice was made due to its relatively low cost. Using an edge detection technique enabled the evaluation of the local characteristic of any pixel just by looking at its four neighbor pixels. Edge detection technique permitted spotting edges in the interpolated pixels. Sharpening spatial filter allowed an enhancement so that there is a minimal information loss after the bilinear interpolation operation [15,16].

In order to find edges around a target interpolated pixel, $P(j,i)$, its four neighbor pixels $P(m-1,i)$, $P(m,i)$, $P(m+1,i)$, and $P(m+2,i)$ should be taken into account. The asymmetry parameter, E, for linear sigmoidal edge detection technique is given in Equation 2.8.

$$E = |P(m+1,i) - P(m-1,i)| - |P(m+2,i) - P(m,i)| \tag{2.8}$$

If E is greater than 0, it means that the variation between pixels $P(m+1,i)$ and $P(m-1,i)$ is greater than the one between pixels $P(m+2,i)$ and $P(m,i)$. On the other hand, if E is less than 0, it can be concluded that the variation between $P(m+2,i)$ and $P(m,i)$ is greater than $P(m+1,i)$ and $P(m-1,i)$. Finally, if E is equal to zero, this means that edges are symmetric at both sides. After finding edges, the related pixels are sent to sharpening spatial filter in order to enhance the edges and eliminate low-frequency noises. Pixels that have to be used in order to calculate the edges location was shown in Figure 2.2.

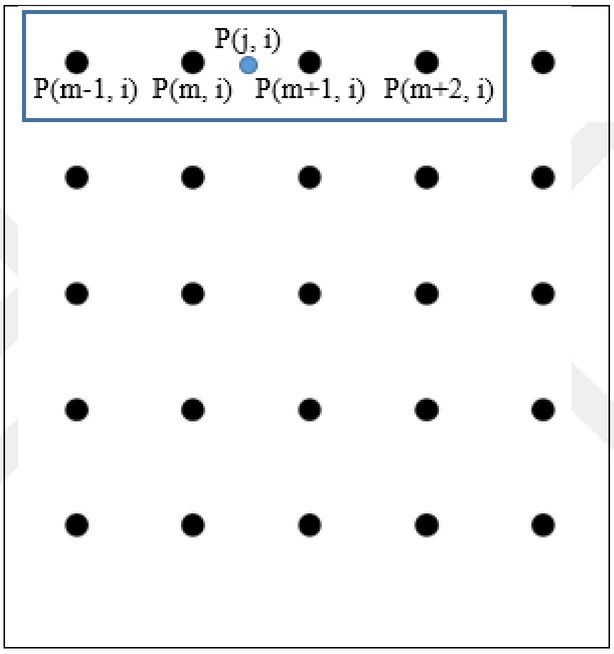


Figure 2.2. Pixel window for edge detection

2.2. SHARPENING SPATIAL FILTER

As previously asserted, sharpening spatial filter [12] acts like a high-pass filter. It can be both used to enhance the edges and to eliminate low-frequency noises. It increases the intensity of the center pixel by using its four neighbor pixels shown in Figure 2.3.

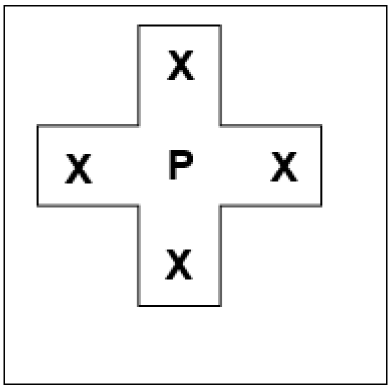


Figure 2.3. Four neighbor pixels

Increasing the intensity of the central pixel by looking at its four neighbor pixels requires at least a three-line buffer memory. In order to reduce the memory requirement of sharpening spatial filter, This thesis uses a method proposed by Chen [11] which requires only a two-line buffer memory (can be reduced to 1 by using register bank). The last form of the neighbor pixel requirement in a 2D pixel grid is shown in Figure 2.4, where only neighbors of the left pixels on the interpolation window are shown. Similarly, the same operation can be applied to all window pixels. Formulas for Chen's method are shown in Equations 2.9, 2.10, 2.11, and 2.12 where *S* denotes the filter sensitivity coefficient.

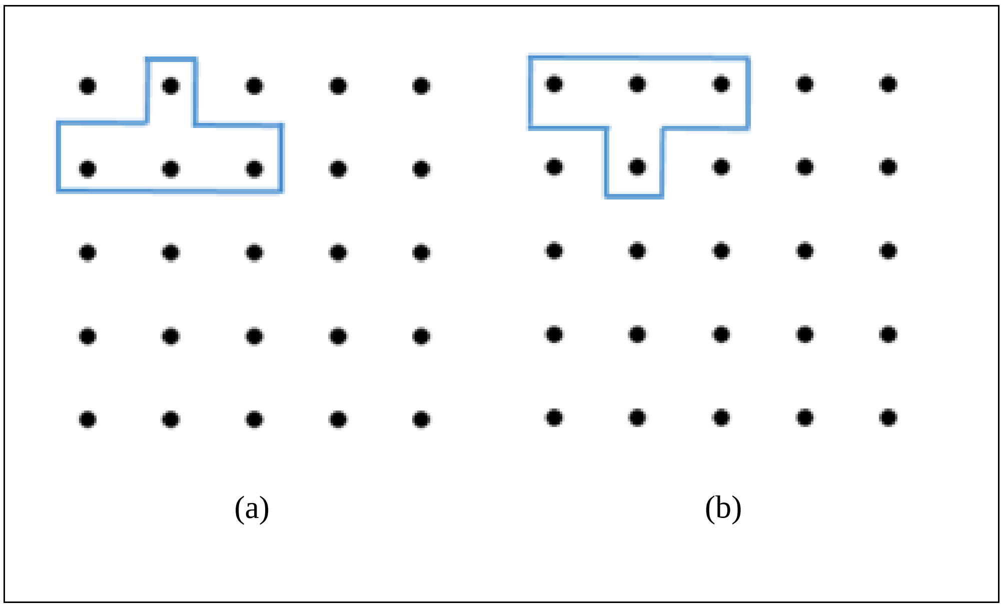


Figure 2.4. Neighbor pixels. (a) Neighbor pixels of the bottom centre pixel, (b) Neighbor pixels of the top center pixel.

$$P'(m,n) = \frac{\big(S \times P(m,n) - P(m+1,n) - P(m,n+1) - P(m-1,n)\big)}{S-3} \tag{2.9}$$

$$P'(m,n+1) = \frac{\big(S \times P(m,n+1) - P(m+1,n+1) - P(m,n) - P(m-1,n+1)\big)}{S-3} \tag{2.10}$$

$$P'(m+1,n) = \frac{\big(S \times P(m+1,n) - P(m+2,n) - P(m+1,n+1) - P(m,n)\big)}{S-3} \tag{2.11}$$

$$P'(m+1,n+1) = \frac{S \times P(m+1,n+1) - P(m+2,n+1) - P(m+1,n) - P(m,n+1}{S-3} \tag{2.12}$$

2.3. REGISTER BANK

In order to implement edge detection, sharpening spatial filter and bilinear interpolation operations, a total number of eight pixels are needed. Four of them are needed for bilinear interpolation, $P(m,\ n)$, $P(m+1,n)$, $P(m,n+1)$, and $P(m+1,n+1)$. Four more are needed for edge detection, $P(m,n)$, $P(m-1,n)$, $P(m+1,n)$, and $P(m+2,n)$, but two of them ($P(m,n)$ and $P(m+1,n)$) are the same ones with bilinear interpolation pixels. Still four more pixels are needed for sharpening spatial filter, which are (in case of top left center pixel) $P(m,n)$, $P(m-1,n)$, $P(m+1,n)$, and $P(m,n+1)$ but again $P(m,n)$ and $P(m+1,n)$ are the same ones with bilinear interpolation pixels.

In total, eight pixels are needed to be available at all times in order to make the necessary calculations. A register bank was used that includes eight registers in order to reduce the total number of pixels to be read from the line buffer. Reading all eight of these pixels from line buffers significantly increases the memory area requirement. To reduce this extra requirement, a register bank was implemented.

Register bank working mechanism is depicted in Figure 2.5. The address calculator module calculates all window pixels addresses for a specific area shown in the top level architecture given in Figure 1.8. After calculating the addresses, in order to obtain necessary pixels, four addresses are sent to the memory module. Note that instead of reading eight pixels from memory, with the help of the register bank, the total number of reads is reduced to four. Four pixels read from line buffers are written to R13, R14, R23, and R24, while the previous values are shifted to R11, R12, R21, and R22 as shown in Figure 2.5. Proceeding like this, all eight necessary pixels can be held in registers and start the module calculations with them by only reading four pixels at a time from line buffers.

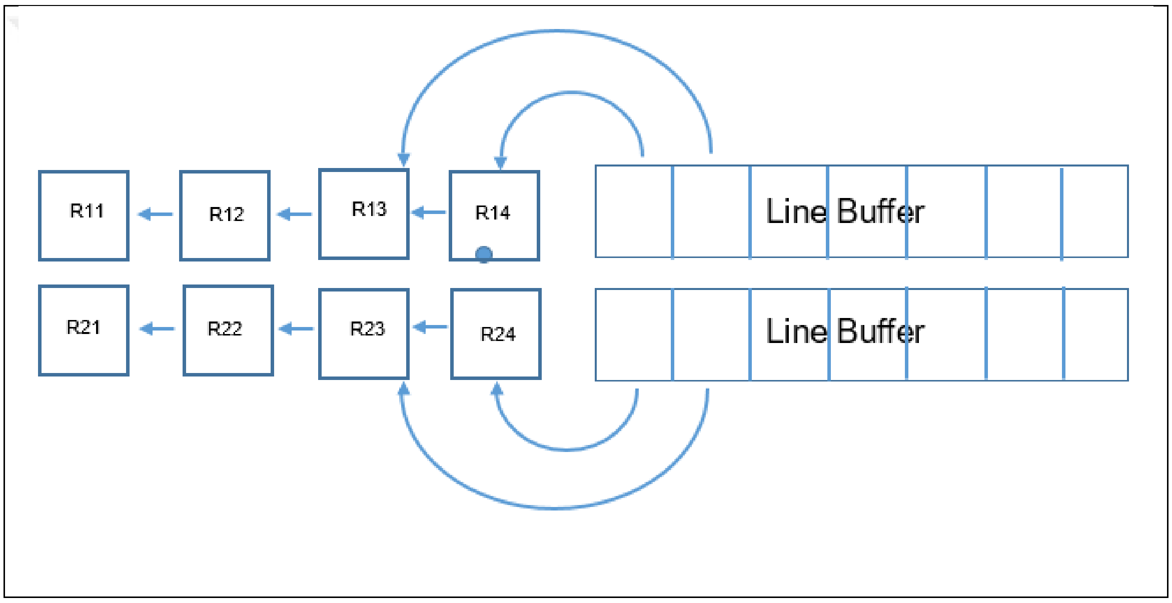


Figure 2.5. Reading from line buffers and register shifting operation

2.4. ODD

The pixel window shift in input domain downscaler [17] is shown in Figure 2.6, whereas the window shift in ODD is shown in Figure 2.7. Note that blue dots denote downscaled pixels in both Figure 2.6 and 2.7. In Figure 2.6, the window is sliding towards to the right with every cycle. This causes empty windows where no calculation can be made because there is no downscaled pixel (no blue dot in the window) in them. In addition, as shown in Figure 2.6, when there is no downscaled pixel within a whole line, input domain downscaler still goes through those lines with empty windows but again, without doing any calculation. On the other hand, ODD does not wait at empty pixel windows as opposed to input domain downscaler because it shifts the window directly to the downscaled pixel location. As it can be seen in Figure 2.7, ODD does not wait at empty windows and does not wait for a whole empty line. In order to achieve such efficiency, more than just one line buffer was used.

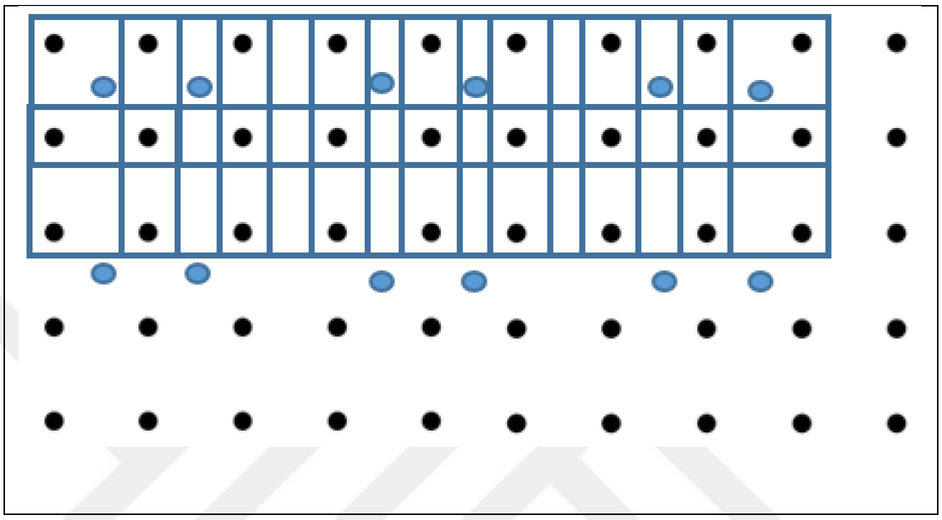


Figure 2.6. Window shift in input domain downscaler

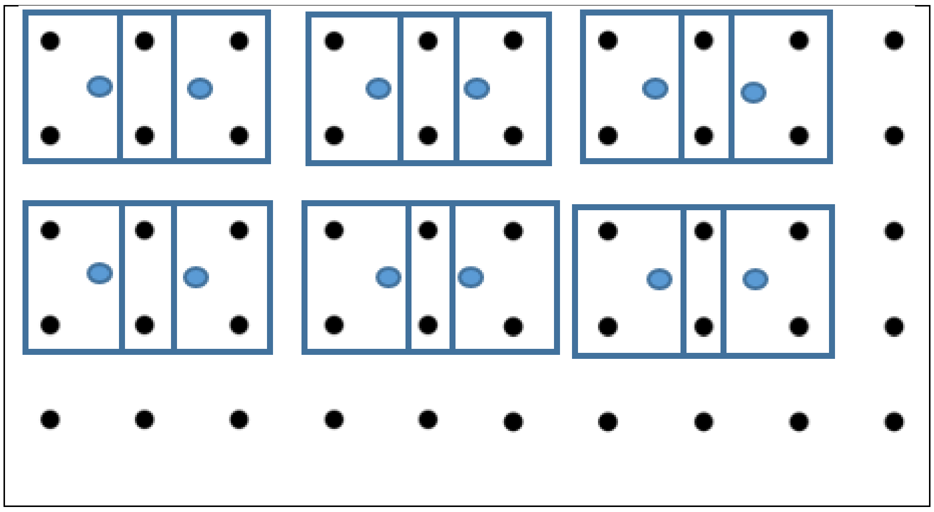


Figure 2.7. Window shift in ODD

The square of downscale ratios has been taken to determine the output rate, because during empty lines, our window just skips the whole line and jumps directly to the next downscaled pixel location. However, this also means that there has to be pre-located pixels ready to be used when window makes the big line jump. In order to satisfy this new prerequisite, line buffer sizes are adjusted and stored enough input pixels before starting to do the window calculations.

Finding the optimal size of the Block RAM (BRAM) required for the line buffers in ODD is very important for area-efficiency and for overcoming under and over flow issues. If the selected BRAM size is less than it is required, that will cause an overflow and the design will not operate properly. If the operation is stalled when there are not enough number of pixels available, the overflow issue is resolved but this causes unnecessary waiting, thus slows down the downscaling process and induces inefficiency. If the selected BRAM size is greater than it is required, the functionality will not be affected, but it will cause unnecessary memory usage and thus more area will be consumed. A mathematical formula should be deployed in order to calculate optimum line buffer size, however making this calculation is very complex and costly.

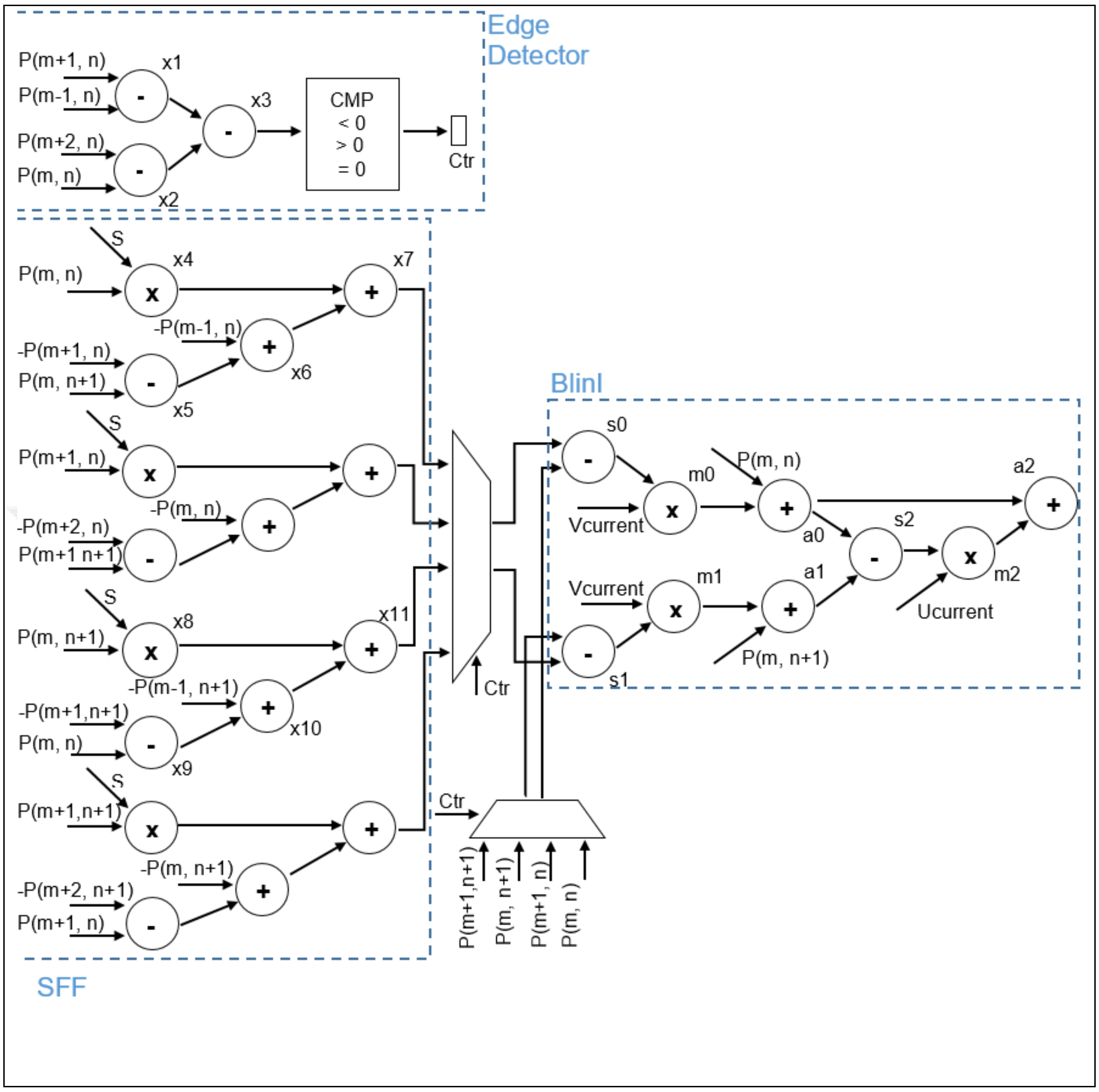
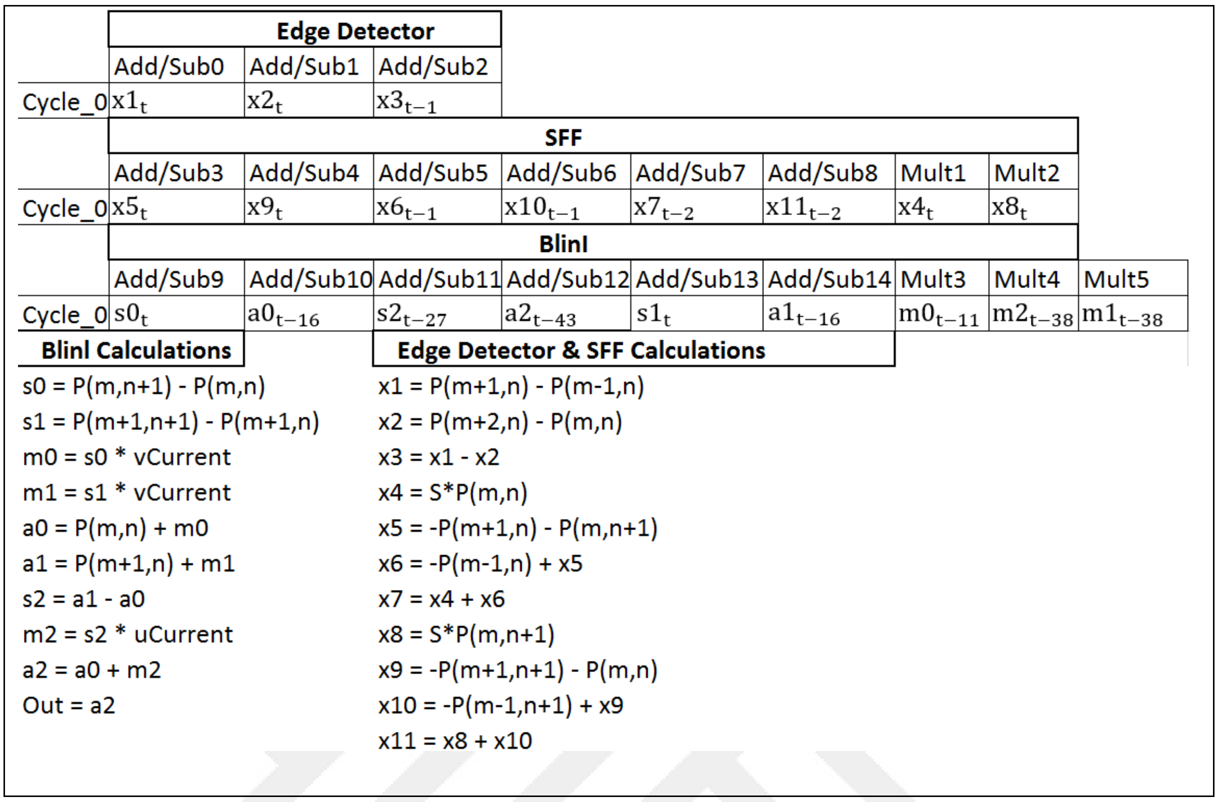


Figure 2.8. Dataflow diagram of ODD with filters

Therefore, a program that simulates the movement of each pixel window in Perl was implemented. This tool provides the exact size needed to use for line buffers in ODD.

Skipping empty windows allows to “share resources” so that a more area-efficient downscaler can be offered, ODD. In ODD, edge detection, sharpening spatial filter and bilinear interpolation calculations are carried out in more than a single cycle because of resource sharing. The dataflow diagram of the downscaler with filters is presented in Figure 2.8. The difference between input domain downscaler and ODD in terms of resource sharing is shown in Figure 2.9 and 2.10 by explaining the resource schedules in input domain downscaler and ODD, respectively.



	Edge Detector		
	Add/Sub0	Add/Sub1	Add/Sub2
Cycle_0	$x1_t$	$x2_t$	$x3_{t-1}$

	SFF							
	Add/Sub3	Add/Sub4	Add/Sub5	Add/Sub6	Add/Sub7	Add/Sub8	Mult1	Mult2
Cycle_0	$x5_t$	$x9_t$	$x6_{t-1}$	$x10_{t-1}$	$x7_{t-2}$	$x11_{t-2}$	$x4_t$	$x8_t$

	BlinI								
	Add/Sub9	Add/Sub10	Add/Sub11	Add/Sub12	Add/Sub13	Add/Sub14	Mult3	Mult4	Mult5
Cycle_0	$s0_t$	$a0_{t-16}$	$s2_{t-27}$	$a2_{t-43}$	$s1_t$	$a1_{t-16}$	$m0_{t-11}$	$m2_{t-38}$	$m1_{t-38}$

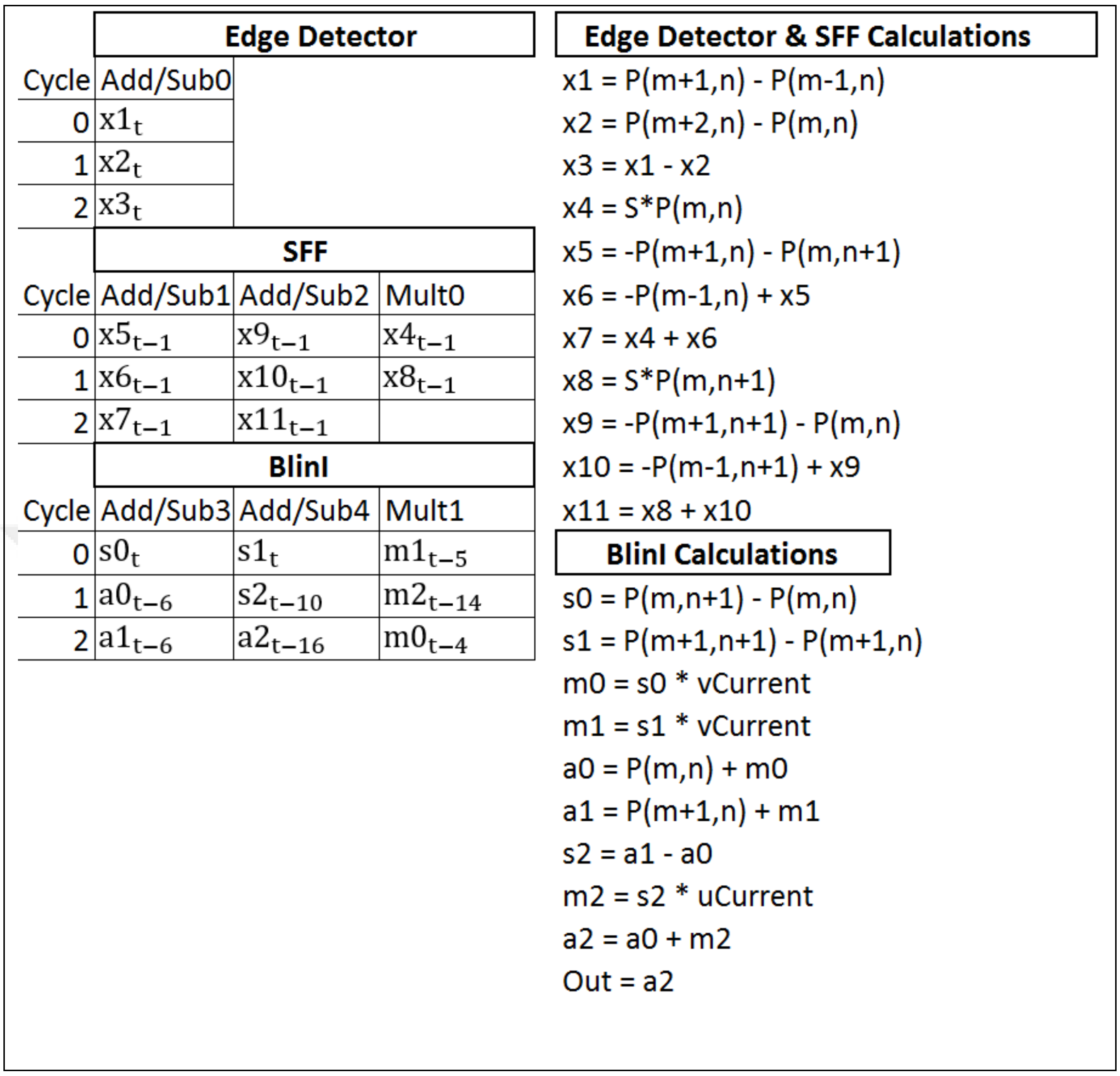
BlinI Calculations

s0 = P(m,n+1) - P(m,n)
s1 = P(m+1,n+1) - P(m+1,n)
m0 = s0 * vCurrent
m1 = s1 * vCurrent
a0 = P(m,n) + m0
a1 = P(m+1,n) + m1
s2 = a1 - a0
m2 = s2 * uCurrent
a2 = a0 + m2
Out = a2

Edge Detector & SFF Calculations

x1 = P(m+1,n) - P(m-1,n)
x2 = P(m+2,n) - P(m,n)
x3 = x1 - x2
x4 = S*P(m,n)
x5 = -P(m+1,n) - P(m,n+1)
x6 = -P(m-1,n) + x5
x7 = x4 + x6
x8 = S*P(m,n+1)
x9 = -P(m+1,n+1) - P(m,n)
x10 = -P(m-1,n+1) + x9
x11 = x8 + x10

Figure 2.9. Schedule of IDD for 1 input pixel rate and 3 output pixel rate



Edge Detector

Cycle	Add/Sub0
0	$x1_t$
1	$x2_t$
2	$x3_t$

SFF

Cycle	Add/Sub1	Add/Sub2	Mult0
0	$x5_{t-1}$	$x9_{t-1}$	$x4_{t-1}$
1	$x6_{t-1}$	$x10_{t-1}$	$x8_{t-1}$
2	$x7_{t-1}$	$x11_{t-1}$	

BlinI

Cycle	Add/Sub3	Add/Sub4	Mult1
0	$s0_t$	$s1_t$	$m1_{t-5}$
1	$a0_{t-6}$	$s2_{t-10}$	$m2_{t-14}$
2	$a1_{t-6}$	$a2_{t-16}$	$m0_{t-4}$

Edge Detector & SFF Calculations

x1 = P(m+1,n) - P(m-1,n)
x2 = P(m+2,n) - P(m,n)
x3 = x1 - x2
x4 = S*P(m,n)
x5 = -P(m+1,n) - P(m,n+1)
x6 = -P(m-1,n) + x5
x7 = x4 + x6
x8 = S*P(m,n+1)
x9 = -P(m+1,n+1) - P(m,n)
x10 = -P(m-1,n+1) + x9
x11 = x8 + x10

BlinI Calculations

s0 = P(m,n+1) - P(m,n)
s1 = P(m+1,n+1) - P(m+1,n)
m0 = s0 * vCurrent
m1 = s1 * vCurrent
a0 = P(m,n) + m0
a1 = P(m+1,n) + m1
s2 = a1 - a0
m2 = s2 * uCurrent
a2 = a0 + m2
Out = a2

Figure 2.10. Schedule of ODD for 1 input pixel rate and 3 output pixel rate

In Figure 2.8, the edge detection, sharpening spatial filter, and bilinear interpolation regions of the downscaler are shown in blue contours. In input domain downscaler, sharpening spatial filter first calculates the enhanced values of all four window pixels and then chooses between them by looking at the information shown in the “ctr” (control) block sent as the output of the edge detection. However, instead of doing these calculations and choosing between four pixels, ODD puts multiplexers before sharpening spatial filter so that it calculates only the enhanced value of the pixels to be used in bilinear interpolation sub-module. By doing like this, ODD eliminates two multiplications and six add/sub arithmetic units.

In input domain downscaler, total number of arithmetic units are thirteen add/sub (three from edge detection, six from sharpening spatial filter, six from bilinear interpolation) and four multipliers (two from sharpening spatial filter, two from bilinear interpolation) as shown in the resource schedule given in Figure 2.9. In ODD, total number of arithmetic units are only five add/sub (one from edge detection, two from sharpening spatial filter, two from bilinear interpolation) and two multipliers (one from sharpening spatial filter, one from bilinear interpolation) as shown in Figure 2.10. Having more than one cycle (in this case three cycles) to do the calculations made resource sharing possible and with resource sharing total number of arithmetic units was reduced to seven from thirteen.

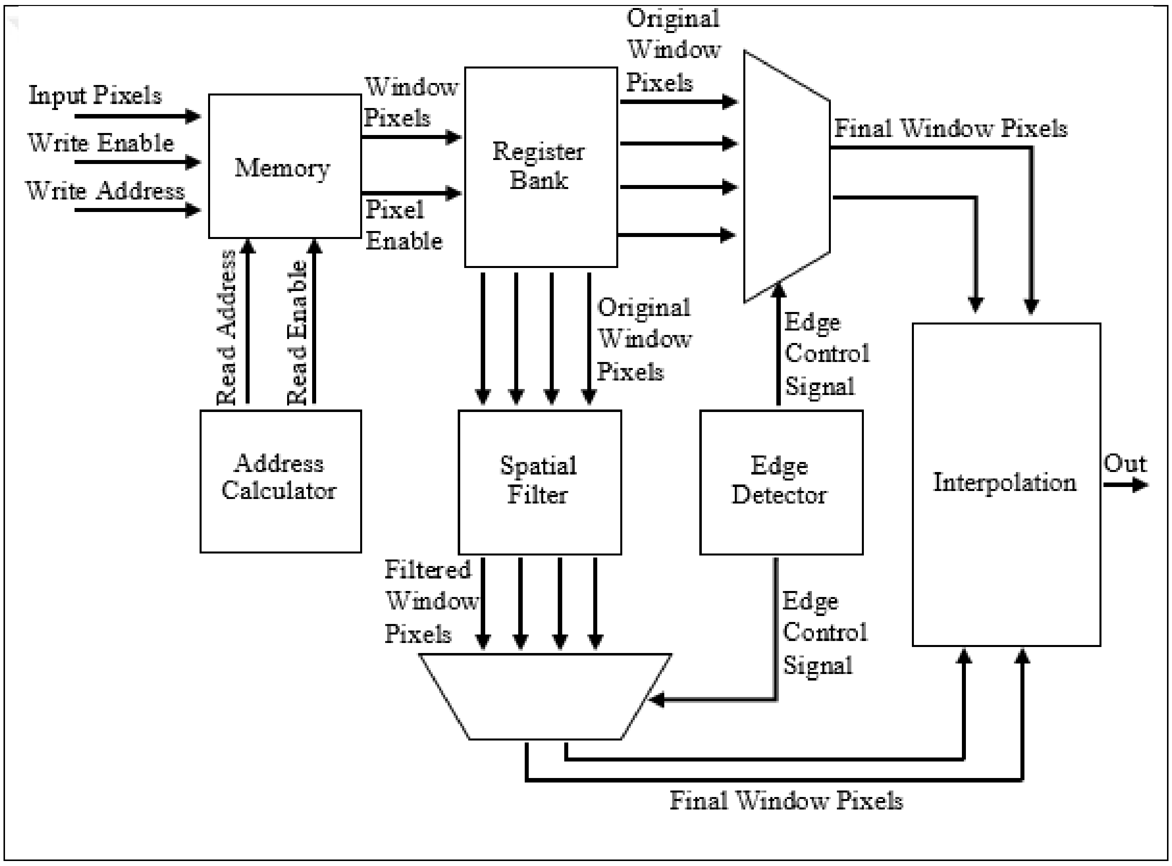


Figure 2.11. Top-level view of ODD

Schedules of all calculations can be seen in Figure 2.9 and 2.10, all calculations are shown under calculations sections. Which arithmetic unit is doing these calculations and when, are shown under edge detector, sharpening spatial filter, and bilinear interpolation sections. Note that pipelining is also used in ODD. Note *m* denotes the column and *n* denotes the row 2-D pixel index. In Figure 2.9 and 2.10, *k* denotes the one-dimensional pixel index. For example,

$x1_k$ is x1 calculation on the k^{th} pixel and $x1_{k-1}$ is x1 calculation on the $(k-1)^{th}$ pixel in one-dimension.

The top module view of ODD is presented in Figure 2.11. Input pixels come to the memory sub-module with write address and write enable flag. If incoming input pixels are valid, write enable signal becomes high and data is written into the indicated address. While reading pixels from the memory, address calculator sub-module comes into place. It calculates the current location of the window and sends necessary address information to the memory sub-module so that correct window pixels can be read from the memory.

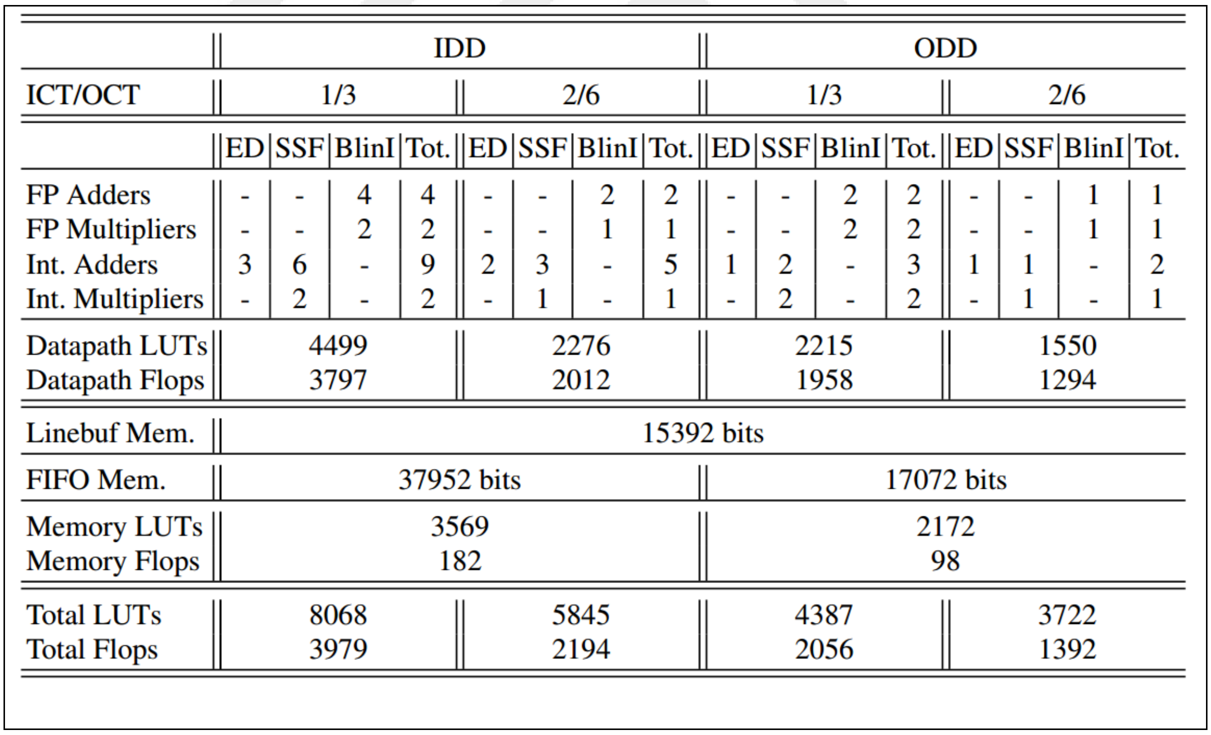
When the pixels read from the memory sub-module they are written in the register bank in a way mentioned in Chapter 2. After all the window pixels are written into registers, sharpening spatial filter begins to make its calculations. Then, the edge detector sub-module looks at the $P(m-1,n)$, $P(m,n)$, $P(m+1,n)$ and $P(m+2,n)$ pixels and determines the location of the edge, then finally sends this information to multiplexers. With the information from the edge detector sub-module multiplexers choose which pixels should go through and which ones should not be used. For example, if the edge is on the right side, pixels to the interpolation sub-module will be $P(m,n)$, $P(m,n+1)$, $P'(m+1,n)$, and $P'(m+1,n+1)$. After obtaining chosen pixels from both SSF and register bank, the interpolation sub-module starts its calculations and computes the final downscaled output pixel.

2.5. SYNTHESIS RESULTS

For testing, a development kit which included a Virtex-7 FPGA was used to implement both ODD and IDD. The frame rate was 90 frame per second (FPS) and the resolution was 1920x1080. As mentioned before, ODD gains from resource sharing by stretching the time to make calculations that the input domain method does not. The latter shifts the window with every cycle and has to make all calculations in one cycle time. The total arithmetic units needed for both ODD and IDD are given in Table 2.1, 2.2, 2.3, and 2.4.

As shown in Tables 2.1, 2.2, 2.3, and 2.4, ODD always uses less arithmetic units than IDD because of resource sharing. Total register and LUT numbers used by each arithmetic unit for Virtex-7 FPGA are given in Table 2.5. By using these numbers and adding the area occupied by line buffers, total area for each method was calculated. For 1920x1080 resolution frames with 90 FPS, the total LUT number of line buffers was 2192 for ODD and total LUT number of line buffers was 1092 for IDD. After adding these numbers to the total area, total register and LUT numbers used by each method are given in Table 2.6.

As seen in Tables 2.6 and 2.7, register gains are not changing when line buffer sizes are changed. This is because line buffers are only using LUT type area and not register type area. Thus only LUT type area is changing when line buffer size changes. When input rate increases, IDD can also start to make resource sharing. That is why in input rate: 2 output rate: 6 case output domain method starts to fall back in terms of LUT type area although it is still ahead in terms of register type area. Another reason is that ODD requires more line buffer memory area than does input domain method, but there can be some cases where input pixel method uses the same line buffer size as output pixel method, as in the case seen in table VII. ODD has a significant lead against input domain method in terms of both LUT and register type areas.



	IDD								ODD							
ICT/OCT	1/3				2/6				1/3				2/6			
	ED	SSF	BlinI	Tot.	ED	SSF	BlinI	Tot.	ED	SSF	BlinI	Tot.	ED	SSF	BlinI	Tot.
FP Adders	-	-	4	4	-	-	2	2	-	-	2	2	-	-	1	1
FP Multipliers	-	-	2	2	-	-	1	1	-	-	2	2	-	-	1	1
Int. Adders	3	6	-	9	2	3	-	5	1	2	-	3	1	1	-	2
Int. Multipliers	-	2	-	2	-	1	-	1	-	2	-	2	-	1	-	1
Datapath LUTs	4499				2276				2215				1550			
Datapath Flops	3797				2012				1958				1294			
Linebuf Mem.	15392 bits															
FIFO Mem.	37952 bits								17072 bits							
Memory LUTs	3569								2172							
Memory Flops	182								98							
Total LUTs	8068				5845				4387				3722			
Total Flops	3979				2194				2056				1392			

Figure 2.12. General comparison of ODD and IDD

Table 2.1. Total arithmetic units of ODD
for input rate: 1 output rate: 3 ratio: 1.8

	Edge Detector	**SSF**	**BlinI**
Floating Point (FP) Adder	None	None	2
FP Multiplier	None	None	2
Integer Point (IP) Adder	1	2	None
IP Multiplier	None	2	None

Table 2.2. Total arithmetic units of input domain downscaler
for input rate: 1 output rate: 3 ratio: 1.8

	Edge Detector	**SSF**	**BlinI**
FP Adder	None	None	4
FP Multiplier	None	None	2
IP Adder	3	6	None
IP Multiplier	None	2	None

Table 2.3. Total arithmetic units of ODD
for input rate: 2 output rate: 6 ratio: 1.8

	Edge Detector	**SSF**	**BlinI**
FP Adder	None	None	1
FP Multiplier	None	None	1
IP Adder	1	1	None
IP Multiplier	None	1	None

Table 2.4. Total arithmetic units of input domain downscaler
for input rate: 2 output rate: 6 ratio 1.8

	Edge Detector	**SSF**	**BlinI**
FP Adder	None	None	2
FP Multiplier	None	None	1
IP Adder	2	3	None
IP Multiplier	None	1	None

Table 2.5. Resource requirement of arithmetic units
for Virtex-7 FPGA

	IP Adder	**IP Multiplier**	**FP Adder**	**FP Multiplier**
Register	27	42	610	615
LUT	37	104	582	807

Table 2.6. Area results of input domain downscaler and ODD
at 1920x1080 resolution and 90 FPS when input domain uses less line buffer

	IR: 1 OR: 3 Ratio: 1.8	**IR: 2 OR: 6 Ratio: 1.8**
IDD	Total Registers: 3797 Total LUTs: 5575	Total Registers: 2012 Total LUTs: 3352
ODD	Total Registers: 1958 Total LUTs: 4387	Total Registers: 1294 Total LUTs: 3722
Total Gain ODD	Register Gain: 48% LUT Gain: 21%	Register Gain: 35% LUT Gain: -9%

Table 2.7. Area results of input domain downscaler and ODD at 1920x1080 resolution and 90FPS same line buffer

	IR: 1 OR: 3 **Ratio: 1.8**	**IR: 2 OR: 6** **Ratio: 1.8**
IDD	Total Registers: 3797 Total LUTs: 6671	Total Registers: 2012 Total LUTs: 4448
ODD	Total Registers: 1958 Total LUTs: 4387	Total Registers: 1294 Total LUTs: 3722
Total Gain ODD	Register Gain: 48% LUT Gain: 34%	Register Gain: 35% LUT Gain: 16%

3. WARPING

Warping module is one of the biggest modules in optical flow. It takes the next frame prediction input sent from iteration module (iteration module was explained in the “Iteration Module” section) and creates a new image by using this input. After creating this next frame prediction, it compares this images pixels one by one with the original next frame and finds out differences (errors made by prediction). After finding out the differences, warping module sends this information to iteration module so that the next prediction made by iteration module will be more accurate.

In order to find differences between frames, warping module requires two frames to work with. First one is called current frame or image 1 which was stored in the FPGA’s local RAM. Second one is called next frame or image 2 which is the upcoming frame, the frame after the current frame. By using this two frames and the pixel movement prediction gained from iteration module warping module finds out the differences and sends this data to iteration module. How warping module works was explained step by step below:

i) Current frame and upcoming frame was taken in pixel by pixel.

ii) U (X vector of pixel movement prediction) and V (Y vector of pixel movement prediction) vector predictions made by iteration was taken in.

iii) Module starts applying pixel movement prediction vectors to upcoming frame in order to find out where those pixels were came from.

iv) First checks if those pixels were came from outside of the images boundaries, boundary check process can be seen in Figure 3.1-2. If pixels came from outside of the boundaries module will act like they came from the edge of the image.

v) After finding out where the pixels from upcoming frame came, warping module finds out four neighbouring pixels around that area to calculate that pixels RGB value.

vi) After finding out surrounding four pixels bilinear interpolation process begins and calculates the RGB value of the pixel.

vii) After the pixels value was calculated warping module compares that predicted pixels value to its actual value which is already known since current frame was stored in the RAM.

viii) Warping module sends the difference between predicted pixel and actual pixel to iteration module so that it can make its next prediction more accurate.

3.1. BOUNDARY CONDITIONS

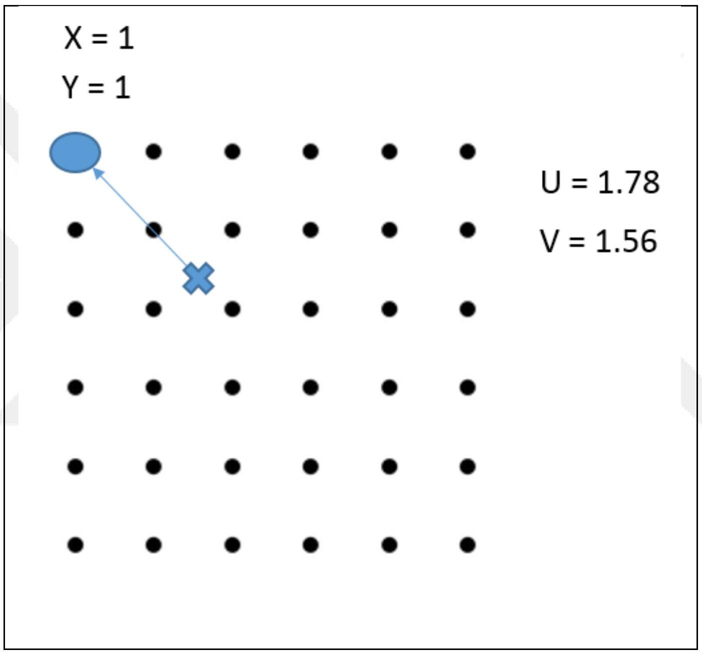


Figure 3.1. Predicted pixel came from inside of the image boundaries

As seen in Figure 3.1 iteration module predicted that pixel came from $(X_{predicted} = X_{actual} + U = 1 + 1.78 = 2.78)$ and $(Y_{predicted} = Y_{actual} + V = 1 + 1.56 = 2.56)$. So it can be seen that predicted pixel came from inside of the image boundaries.

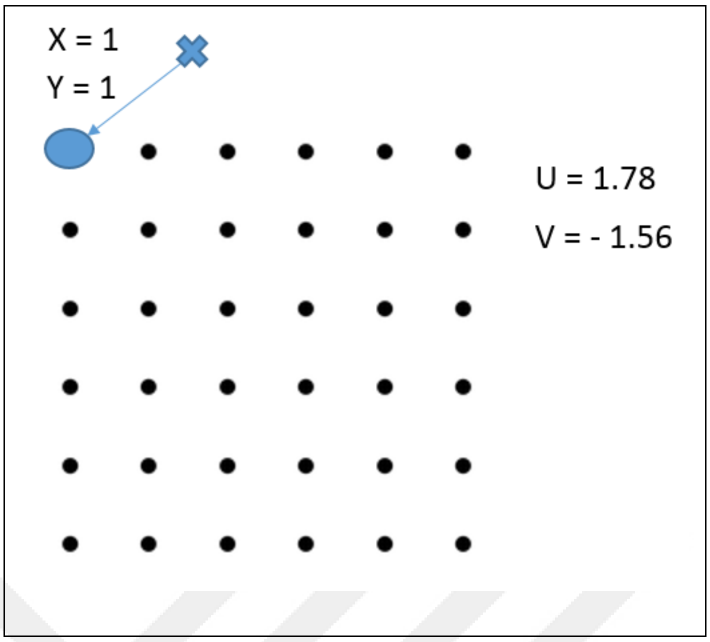


Figure 3.2. Predicted pixel came from outside of the image boundaries

As seen in Figure 3.2 iteration module predicted that pixel came from $(X_{predicted} = X_{actual} + U = 1 + 1.78 = 2.78)$ and $(Y_{predicted} = Y_{actual} + V = 1 - 1.56 = -0.44)$. So it can be seen that predicted pixel came from outside of the image boundaries. In this situation warping module changes the coordinates of the pixel as if it came from the edge of the image. So pixels new coordinates will be X = 2.78 and Y = 0.

Another key point about warping modules boundary conditions is, it has a limited memory support. Cost holding a full frame in a block ram would be very high, because of that, warping module implemented in a way so that it could support different memory sizes. As mentioned before, both warping and downscaling uses a very similar generic FIFO structure, warping unit currently supports total of nine full frame rows in its FIFO. Holding nine row means warping unit can support and accurately calculate motion vectors between plus and minus four. Increasing the number of frame rows supported would mean increasing the total block ram usage, likewise decreasing the number of frame rows supported would mean decreasing the total block ram usage.

3.2. MODULE CONNECTIONS

After checking the boundary conditions, warping module calculates the predicted pixels value since its value is unknown because it was predicted as coming not from a legit pixel location but from a location in between pixels. There are several methods available to calculate such unknown pixel values as mentioned before in downscaler module section of the thesis. Best solution is to use bilinear interpolation algorithm to use unknown pixels four neighbouring pixels to calculate its value. All the references and explanations about bilinear interpolation method can be found in the bilinear interpolation section of the thesis.

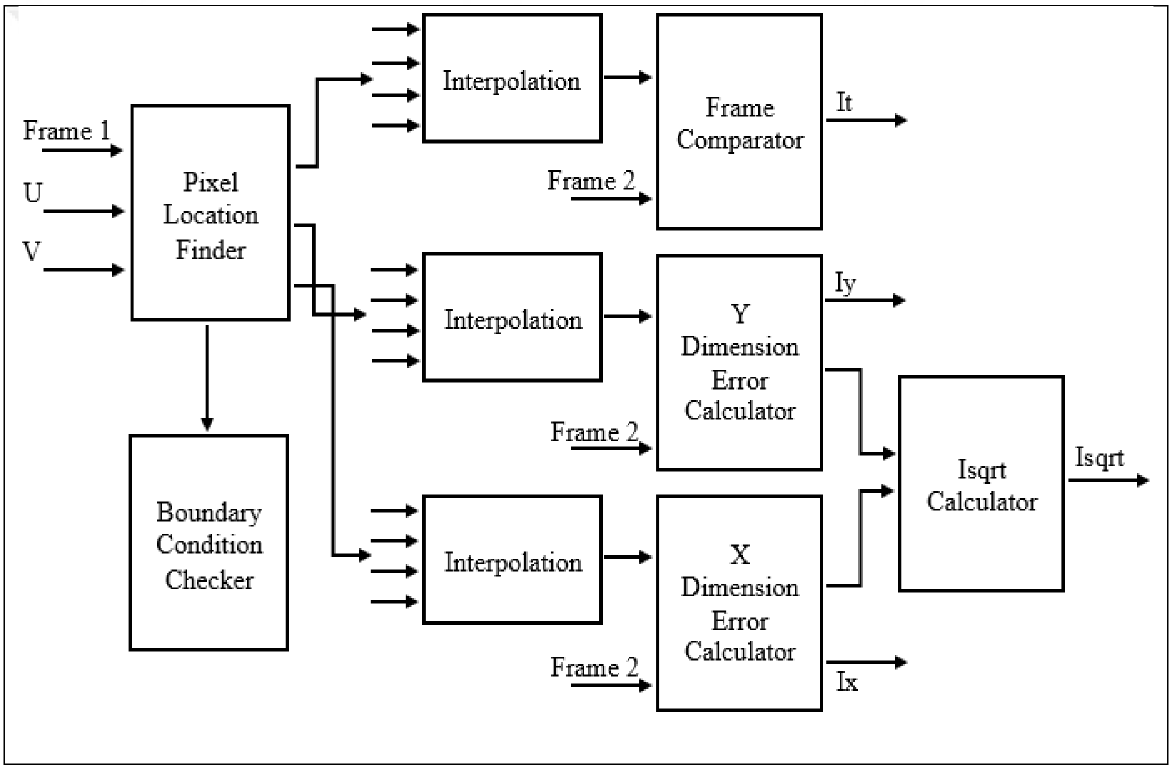


Figure 3.3. Block diagram of the warping module

Top module diagram of the warping module can be seen in Figure 3.3. As shown in the figure, after the bilinear interpolation operation completed and value of the predicted pixels was calculated warping module starts comparing the predicted pixels value with actual pixels value so that prediction error can be determined and sent to the iteration module.

Warping module compares the values of the actual pixel and predicted pixel in three ways. First way is to directly subtract the value of the predicted pixel from actual pixel, if the result is higher or smaller than zero that means there is an error. But if the result is equals to zero that means prediction was correct and there is no error. Second way is increasing the predicted pixel’s x coordinate value by zero point five, calculating the value of the pixel by using bilinear interpolation. Then decreasing the pixels x coordinate value by the same value zero point five and calculating its new value again by using bilinear interpolation and calculating the deviation between them. Third way is to do exactly same thing in second way, but this time increasing and decreasing the y coordinate instead of the x coordinate and calculating the deviation. After all the calculations are completed, warping module sends error information to iteration module. Connection between the warping module and the iteration module can be seen in Figure 3.4.

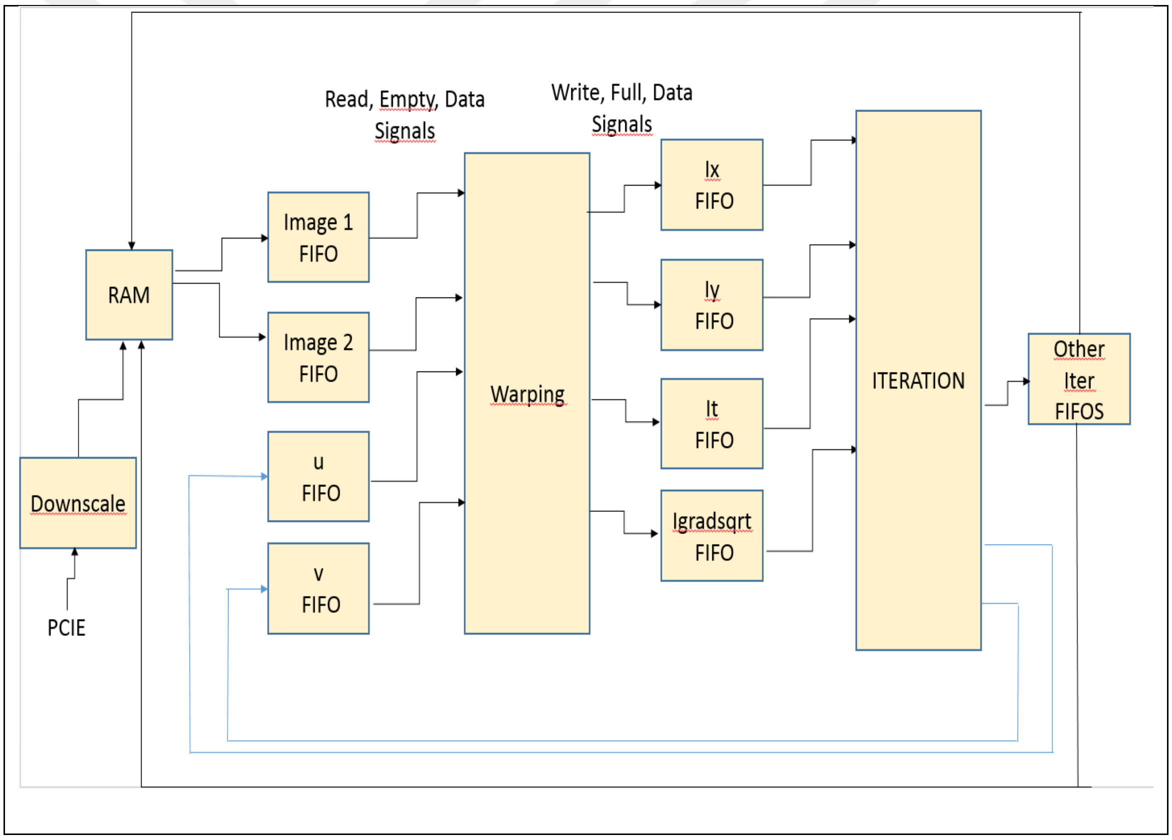


Figure 3.4. Connection diagram of warping and iteration modules

Warping module and iteration modules are connected to each other by using FIFO (first in first out). Reason behind this is because they can be work in different speeds. For example if warping module gives output data in every ten cycles but iteration module takes input data in twenty cycles, connecting them to each other without using fifos causes information loss.

A mathematical formula should be deployed in order to calculate optimum line buffer size, however making this calculation is very complex and costly. Therefore, a program that simulates the movement of each pixel window in Perl was implemented. This tool provides the exact size needed to use for line buffers in ODD. Working principle of a FIFO was shown in Figure 3.5.

3.3. FIFO

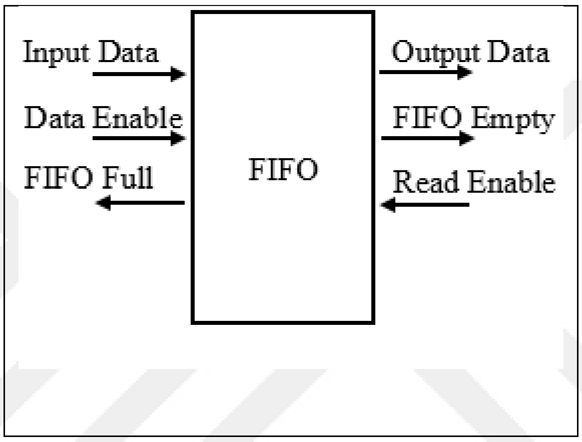


Figure 3.5. FIFO connections

In Figure 3.5 input data port is the port which transmits input data to fifo, data enable port tells the FIFO if the incoming data is legit or not, FIFO full port tells the user of the FIFO if the FIFO is full or not. Output data port is the exit port of the FIFO, FIFO empty port tell the user if the FIFO is empty or not, thus if the data user reading was legit or not, finally read enable port tells the FIFO if any data has been read from FIFO.

3.4. ITERATION MODULE

Iteration module predicts the movement of the pixels between video frames. It uses two frames to compare with each other and makes a prediction about how fast pixels changed their locations and in which direction. This thesis focuses mainly on downscaler, warping and image fusion algorithms, iteration algorithm will be explained shortly in order to clarify how optical flow algorithm works. The module called iteration because the calculations in this module iteratively repeat themselves in order to obtain more accurate pixel movement predictions. More iterations means more accurate predictions.

Iteration module takes u, v, w and p vectors as input

- U: Optical flow vector of the related pixel in x coordinate
- V: Optical flow vector of the related pixel in y coordinate
- W: Auxiliary optical flow vector of the related pixel
- P: Gradients of u,v and w

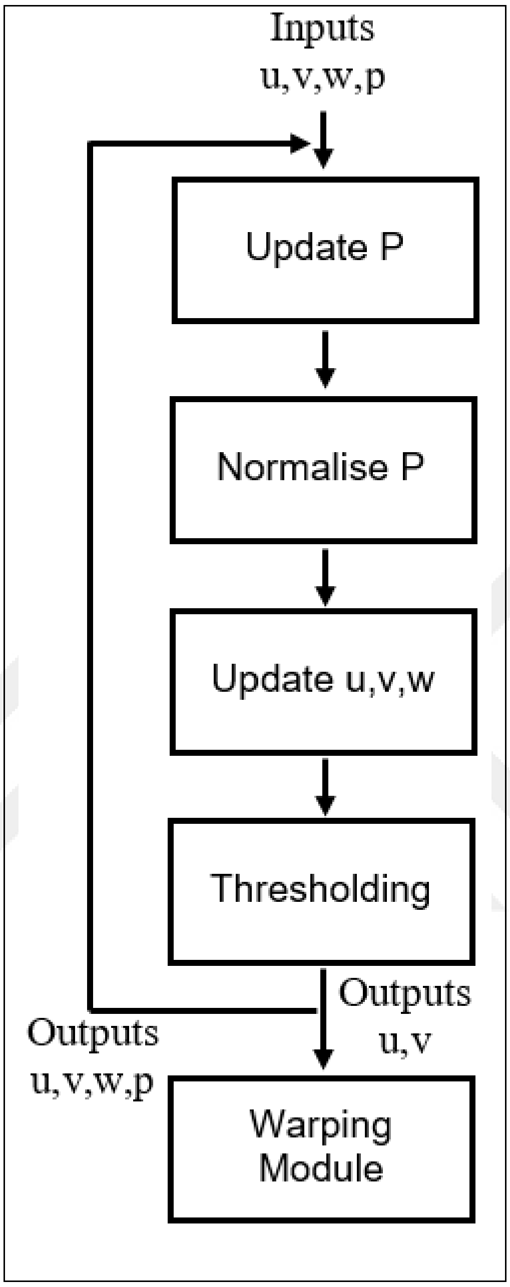


Figure 4.6. Iteration module block diagram

As seen on Figure 3.6 iteration module constantly updates its u, v, w and p coefficients at the same time sends u and v vectors to warping module so that warping module can calculate the estimate error made by iteration module. All calculations in iteration module was made by using floating point units. Area and performance results of the iteration module can be seen in Table 3.1.

Table 3.1. Area and performance results of iteration module

Total logic elements	300.595
Total memory requirement	18.356
Maximum frequency	215 MHz

3.5. SYNTHESIS RESULTS

All calculations in warping module was done with fixed point calculation units. Total number of adder/subtractor and multiplier was shown in Table 3.2. Total area and performance results of warping module was shown in Table 3.3.

Table 3.2. Total number of adder/subtractors and multipliers used in warping module

Calculation unit	**Total Number**
Adder/Subtractor	9
Multiplier	2

Table 3.3. Total area and performance results of warping module

Total logic elements	8,840 / 149,760 (< 7 %)
Total register number	15,488
Total memory requirement	262,274 / 6,635,520 (< 7 %)
Total 9-bit multipliers	42 / 720 (< 8 %)
Maximum frequency	183 MHz

4. TEST AND VERIFICATION ENVIRONMENT

In this thesis, for each of the proposed methods (downscaler, warping and iteration) a dedicated test and verification environment was used in order to check whether the algorithms were working as desired or not. All algorithms was implemented in MATLAB environment first, to check if they are working and to obtain a correct data for later comparison with FPGA output. First step of testing was to write random numbers in the input FIFOs of all algorithm modules and to check if calculations inside the modules was made correctly. Second step of testing was sending an image (single frame) data to module input FIFOs and check if there was a meaningful image (frame) at the output. Finally the third step was to sending continuous video and check if the output frames was correct.

All the random data, image and video comparisons was made with the data provided by MATLAB implementation of the algorithms. Input data of the module FIFOs was sent from a computer with peripheral component interconnect express (PCIe) bus. After modules made their calculations and generated the manipulated frames, output data was sent to the computer, again by using PCIe bus.

In computer, all the comparisons between MATLAB data and the FPGA data was made in a verification environment written with java. This environments duty was to compare the results from MATLAB and FPGA and to create a report summary file so that the differences between MATLAB data and FPGA data can be seen clearly. Flow chart of the verification flow can be seen in Figure 4.1.

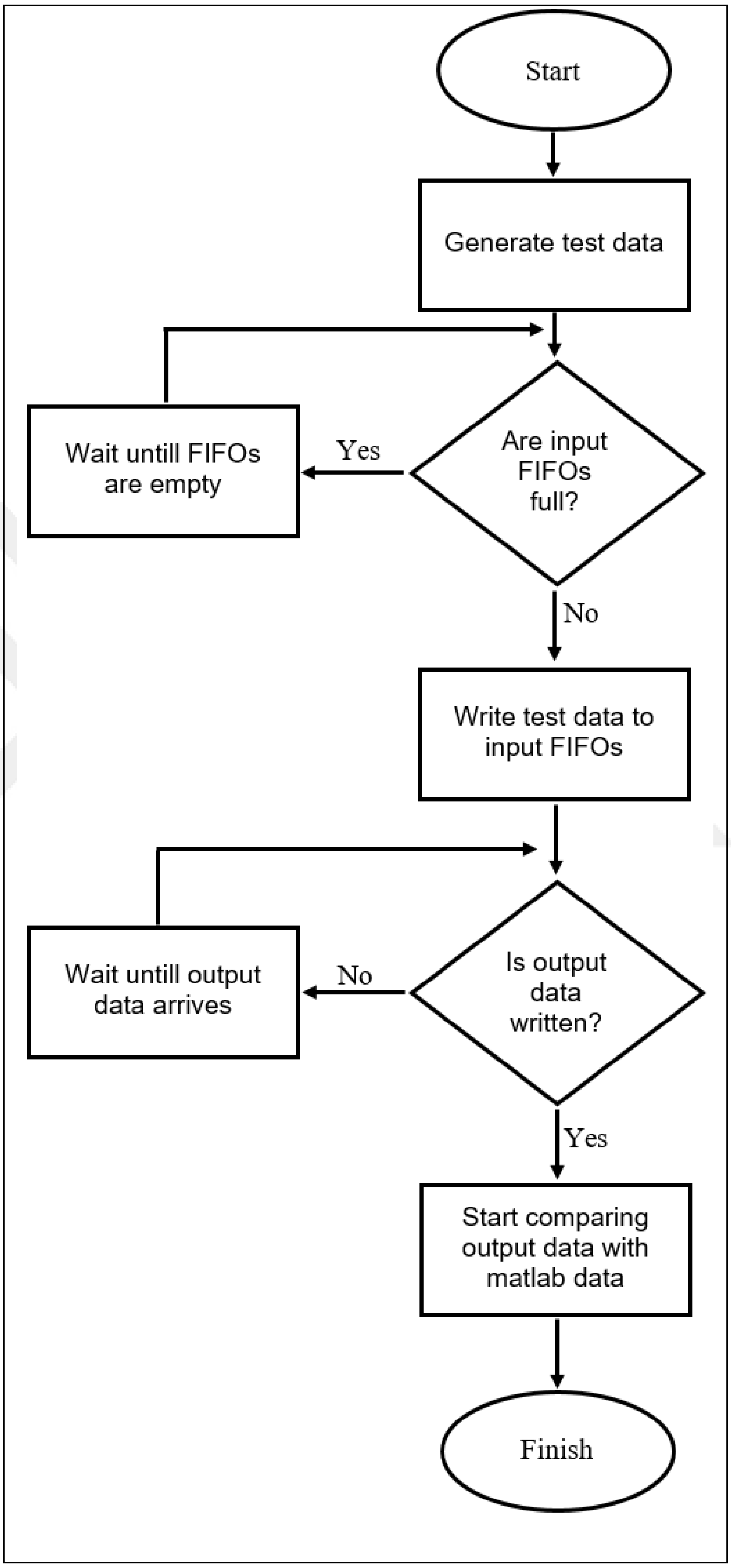

Start
Generate test data
Are input FIFOs full?
Yes
Wait untill FIFOs are empty
No
Write test data to input FIFOs
Is output data written?
No
Wait untill output data arrives
Yes
Start comparing output data with matlab data
Finish

Figure 4.1. Test and verification environment flow chart

Test cases for both warping and downscaler module can be seen in Figures 4.2-3. In case of downscaler, in first step, data taken from video frame was written into the input FIFOs of the downscaler module by using PCIe bus. In second step downscaler module makes its calculations and starts to generate its output data. In third step output data was written into the output FIFOs of the downscaler module and sent to computer by using PCIe. In last step verification environment starts to compare the data came from FPGA with the data came from MATLAB and generates a result summary report. The same steps and the same verification environment was also used on warping and iteration modules. Only the inputs and outputs were different.

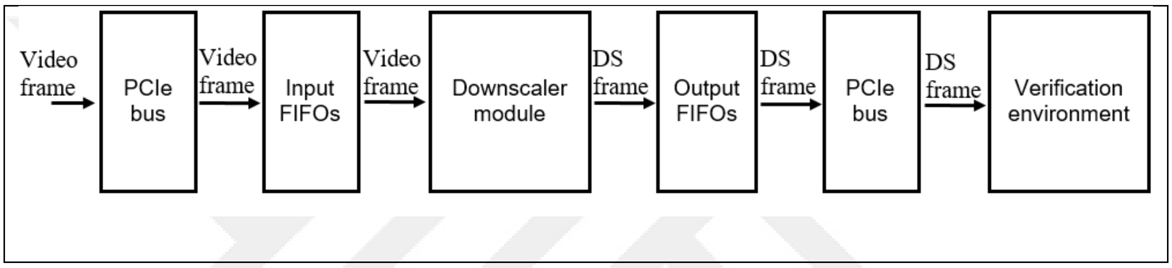


Figure 4.2. Block diagram of downscaler verification process

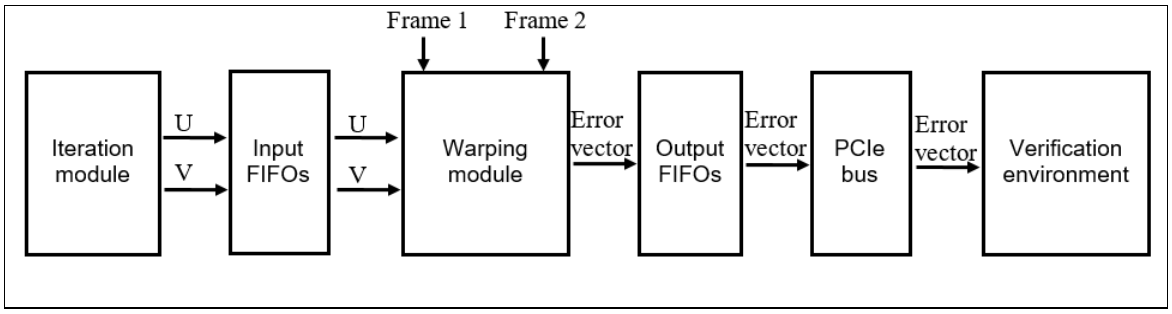


Figure 4.3. Block diagram of warping verification process

5. CONCLUSION

In this thesis, a low cost, low memory downscale method called ODD and an implementation of a warping algorithm was presented. Downscale method includes edge detection system and a sharpening spatial filter with it. I have shown that output domain method is superior to traditional downscale methods in terms of register and LUT type areas. Presented method gains in the order of 48% register type area and 21% LUT type area. These values were obtained for 1920x1080 resolution, 90 FPS and 1-3 input output ratio. Warping unit was implemented by using a high level synthesis tool called MAFURES [18]. Usage of this tool reduced the complexity and increased the design speed. Also another tool was used to calculate the fifo sizes between the warping and the iteration modules. Both downscaling and warping units shared the bilinear interpolation method as their core pixel value calculator. And both downscaler and warping units shared the similar general FIFO structure for memory read and write operations. Resource sharing, loop unrolling and pipelining design methods was used for both downslcaing and warping units in order to reduce the total arithmetic unit usage and costs.

In the future, further optimizations can be made to both downscaling and warping implementations in order to reduce the total area required and increase the maximum frequency. By changing the total pixel movement coverage, total area requirement of the warping unit can be reduced. And by modifying the window skip method, total area requirement of the ODD can be also reduced.

REFERENCES

1. H. Kim, Y. Cha, and S. Kim. Curvature Interpolation Method for Image Zooming. *IEEE Transactions on Image Process*ing, 7: 1895–1903, 2011.

2. T. M. Lehmann, C. Gonner, and K. Spitzer. Survey: Interpolation Methods in Medical Image Processing. *IEEE Transactions on Medical Imaging,* 11: 1049–1075, 1999.

3. S. Tao, J. Apostolopoulos, and R. Guerin. Real-Time Monitoring of Video Quality in IP Networks. *IEEE Transactions on Networking*, 5: 1052–1065, 2008.

4. R. Lukac, K. N. Plataniotis, and D. Hatzinakos. Color Image Zooming on the Bayer Pattern. *IEEE Transactions on Circuits and Systems for Video Technology*, 11: 1475–1492, 2005.

5. V. Caselles, J. M. Morel, and C. Sbert. An Axiomatic Approach to Image Interpolation. *IEEE Transactions on Image Process*ing, 3: 376–386, 1998.

6. E. Meijering, K. J. Zuiderveld, and M. A. Viergever. Image Reconstruction by Convolution with Symmetrical Piecewise *n*th-Order Polynomial Kernels. *IEEE Transactions on Image Process*ing, 2: 192–201,1999.

7. M. A. Nuno-Maganda and M. O. Arias-Estrada. Real-Time FPGA-Based Architecture for Bicubic Interpolation: An Application for Digital Image Scaling. *2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05)*, Puebla City, 2005.

8. K. S. Ni and T. Q. Nguyen. Adaptable k-Nearest Neighbor for Image Interpolation. *IEEE International Conference on Acoustics, Speech and Signal Processing*, 1297-1300, 2008.

9. K. Jensen and D. Anastassiou. Subpixel Edge Localization and the Interpolation of Still Images. *IEEE Transactions on Image Process*ing, 3: 285–295, 1995.

10. W. Y. V. Leung, P. J. Bones and R. G. Lane. Statistical Interpolation of Sampled Images, *Optical Engineering*, 8: 547–553, 2001.

11. S. L. Chen. VLSI Implementation of an Adaptive Edge-Enhanced Image Scalar for Real-Time Multimedia Applications. *IEEE Transactions on Circuits and Systems for Video Technology*. 9: 1510-1522, 2013.

12. S. Schaller, J. E. Wildberger, R. Raupach, M. Niethammer, and K. Klingenbeck-Regn. Spatial Domain Fltering for Fast Modification of the Tradeoff Between Image Sharpness and Pixel Noise in Computed Tomography. *IEEE Transactions on Medical Imaging*, 7: 846–853, 2003.

13. P. Y. Chen, C. Y. Lien, and C. P. Lu. VLSI Implementation of an Edgeoriented Image Scaling Processor. *IEEE Transactions on Very Large Scale Integrated Systems*, 9: 1275–1284, 2009.

14. G. Ramponi. Warped Distance for Space-Variant Linear Image Interpolation. *IEEE Transactions on Image Process*ing, 5: 629–639, 1999.

15. G. Priya and G. Vairavel. VLSI Implementation of Image Scaling processor. *Electronics and Communication Systems (ICECS), 2014 International Conference on*, Coimbatore, 2014.

16. S. L. Chen, H. Y. Huang and C. H. Luo. A Low-Cost High-Quality Adaptive Scalar for Real-Time Multimedia Applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 11: 1600-1611, 2011.

17. M. Büyükmıhcı, V. E. Levent, A. E. Guzel, O. Ateş, M. Tosun, T. Akgün, C. Erbaş, S. G. Uğurdağ, H. F. Uğurdağ. Output Domain Downscaler. In: Czachórski T., Gelenbe E., Grochla K., Lent R. (eds) Computer and Information Sciences. ISCIS 2016. Communications in Computer and Information Science, vol 659. Springer, Cham, 2016

18. A. E. Guzel, V. E. Levent, M. Tosun, M. A. Özkan, T. Akgun, D. Büyükaydin, H. F. Ugurdag. Using high-level synthesis for rapid design of video processing pipes. In *East-West Design and Test Symposium (EWDTS) IEEE*, 10: 1-4, 2016.