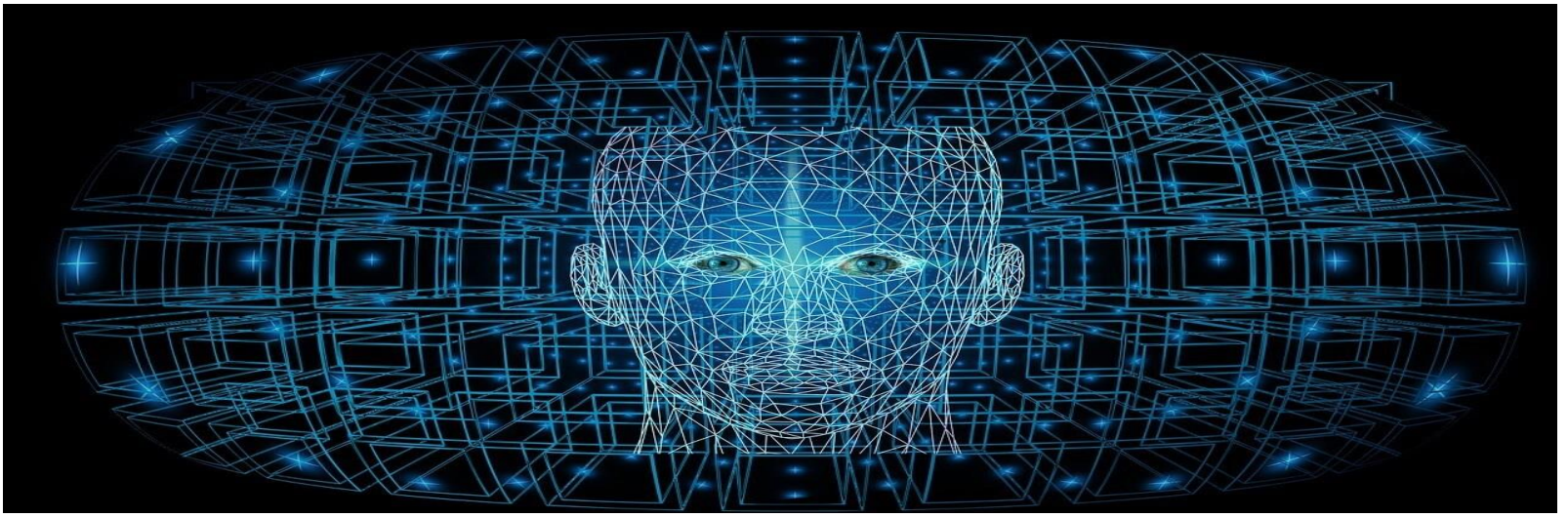


NED University of Engineering & Technology

Spring Semester 2020

CIS - TE

IMAGE PROCESSING ON FPGA

VLSI DESIGN
CS-319

Dr. Saad Qasim

Abdul Ahad	Syedah Nawal Munif	Ahsan Abbas
CS0-17022	CS-17024	CS-15130

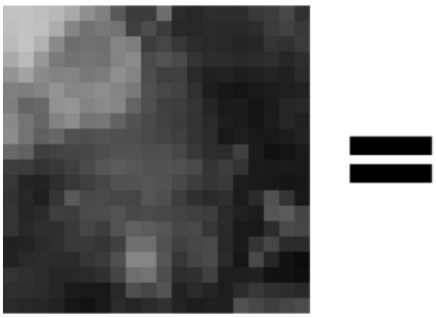
Table of Contents

Abstract

In order to implement the image processing algorithms and to process the amount of data captured from sources such as medical instruments, intelligent high speed real-time systems, etc. has become imperative. In this project, an efficient FPGA-based design and implementation of image processing algorithm will be presented using verilog hardware description language. The FPGA provides the necessary hardware for image processing algorithms with flexibility to support image processing by using different algorithms. Verilog coding is done in Vivado 2014.2.

Imaging Basics

Images are like matrix. If they are gray-scale then they are matrix of two-dimensional arrays but if they are colored then we say that they are three-dimensional. The following image is decomposed into pixels where each box represents a single pixel. 1 byte data represents the intensity of the pixel in case of a gray-scale image. If it is a colored image, each box will have three values representing the RGB values of that pixel.

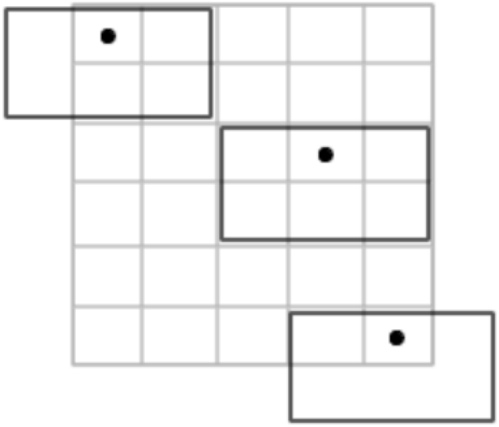


193	194	190	184	169	144	128	89	60	60	109	44	40	46	45	45	58	61	72	50
191	195	192	173	134	114	121	116	64	77	60	41	41	43	41	41	46	60	67	57
187	196	178	139	110	113	112	132	126	61	70	55	45	42	40	37	39	53	50	59
170	186	151	122	114	117	114	131	139	76	83	74	52	45	45	41	43	46	39	47
147	163	139	131	132	121	125	143	132	78	64	64	42	33	35	30	32	36	33	43
129	126	132	148	134	136	141	133	121	81	72	67	49	30	24	21	25	30	32	34
126	101	106	146	149	132	138	134	101	80	65	62	53	37	27	28	21	28	39	40
137	117	103	130	141	118	119	99	83	74	66	60	52	42	30	27	21	29	39	33
141	115	97	103	82	79	84	80	79	74	69	64	52	45	26	31	25	25	29	35
105	99	64	67	70	71	78	83	83	79	80	72	57	46	44	65	29	18	21	27
63	60	52	56	65	75	86	92	87	82	83	81	74	53	62	52	27	26	27	25
55	37	35	46	56	64	71	82	83	85	82	73	62	54	58	30	27	25	22	24
51	40	32	54	90	73	75	71	70	85	79	62	49	43	47	29	44	82	53	25
54	39	43	63	61	75	76	83	73	75	72	68	64	61	42	33	29	86	92	69
53	41	47	47	58	70	62	83	97	92	85	71	73	55	42	27	25	45	75	70
34	36	43	47	58	58	53	65	112	111	81	65	71	74	41	28	53	70	43	43
35	42	45	43	43	54	59	80	134	133	86	55	69	74	43	23	72	39	19	13
52	48	44	39	40	42	49	69	111	115	78	46	49	65	31	31	34	19	18	15
47	46	43	39	28	25	25	40	80	71	84	35	43	37	30	44	56	48	45	41
43	40	39	35	23	19	26	44	38	44	50	36	29	31	31	81	72	63	70	76

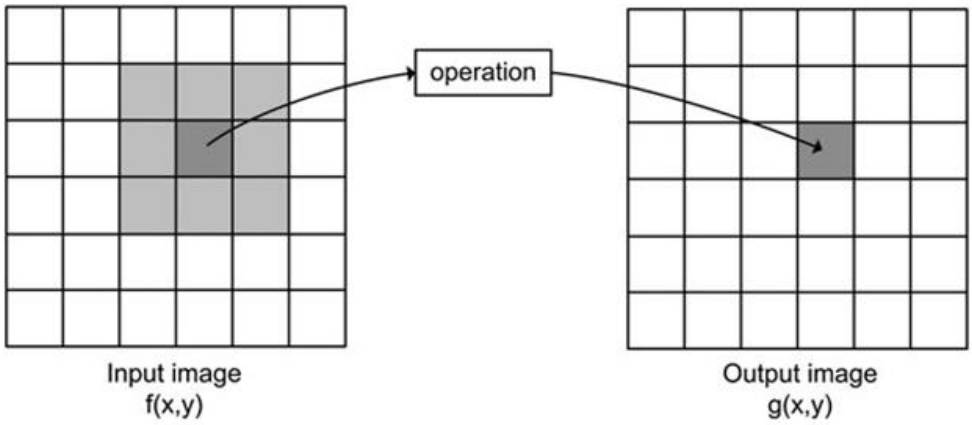
In this project, we have performed processing of gray-scale images which are two-dimensional array matrices. In fact a `gray' color is one in which the red, green and blue components all have equal intensity in RGB space, and so it is only necessary to specify a single intensity value for each pixel, as opposed to the three intensities needed to specify each pixel in a full color image.Often, the grayscale pixel intensity is stored as an 8-bit integer (1 byte) giving 256 possible different shades of gray from black to white.

Neighborhood Image Processing

A sliding neighborhood operation is an operation that is performed on a pixel at a time, with the value of any given pixel in the output image being determined by applying some algorithm to the values of the corresponding input pixel's *neighborhood.* A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel, which is called the *center pixel*. The neighborhood is a rectangular block, and as you move from one element to the next in an image matrix, the neighborhood block slides in the same direction.



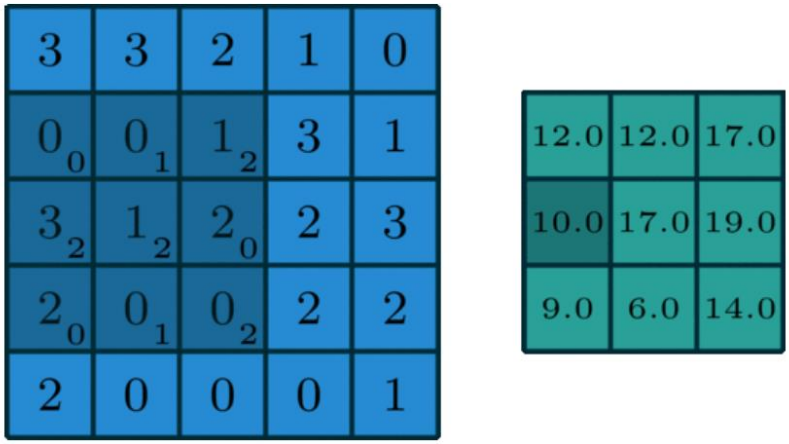
In neighborhood processing, we calculate the new pixel of the output image from the pixel of the input image, which is not only based on its value in the input image, but also on the its neighbors. There are eight immediate neighbors of each pixel in the image matrix.



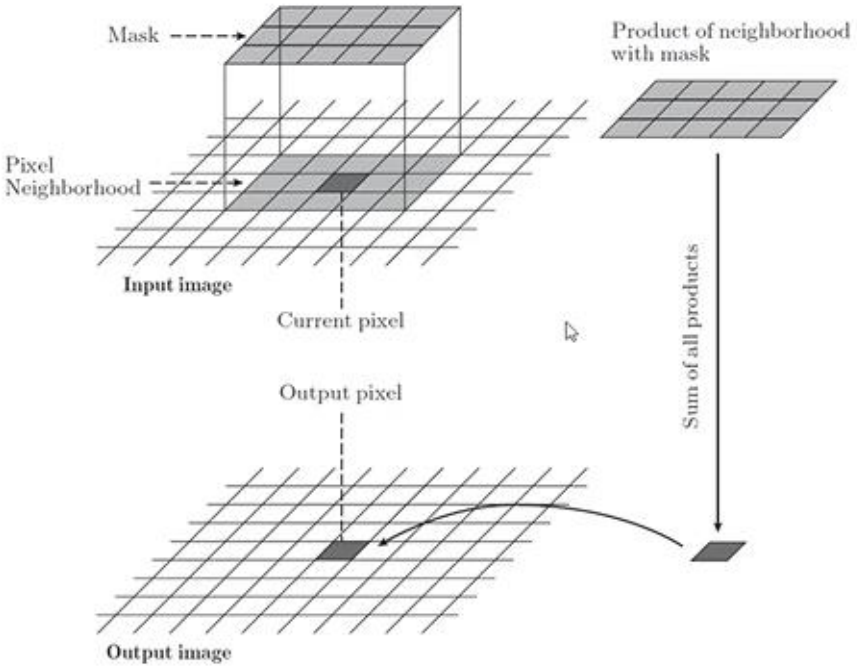
2D Convolution in Image Processing

Convolution is a simple mathematical operation which is fundamental to many common image processing operators. Convolution provides a way of `multiplying together' two arrays of numbers, generally of different sizes, but of the same dimensionality, to produce a third array of numbers of the same dimensionality.

In 2D convolution we start with a kernel, which is simply a small matrix of weights. This kernel “slides” over the 2D input data, performing an elementwise multiplication with the part of the input it is currently on, and then summing up the results into a single output pixel.

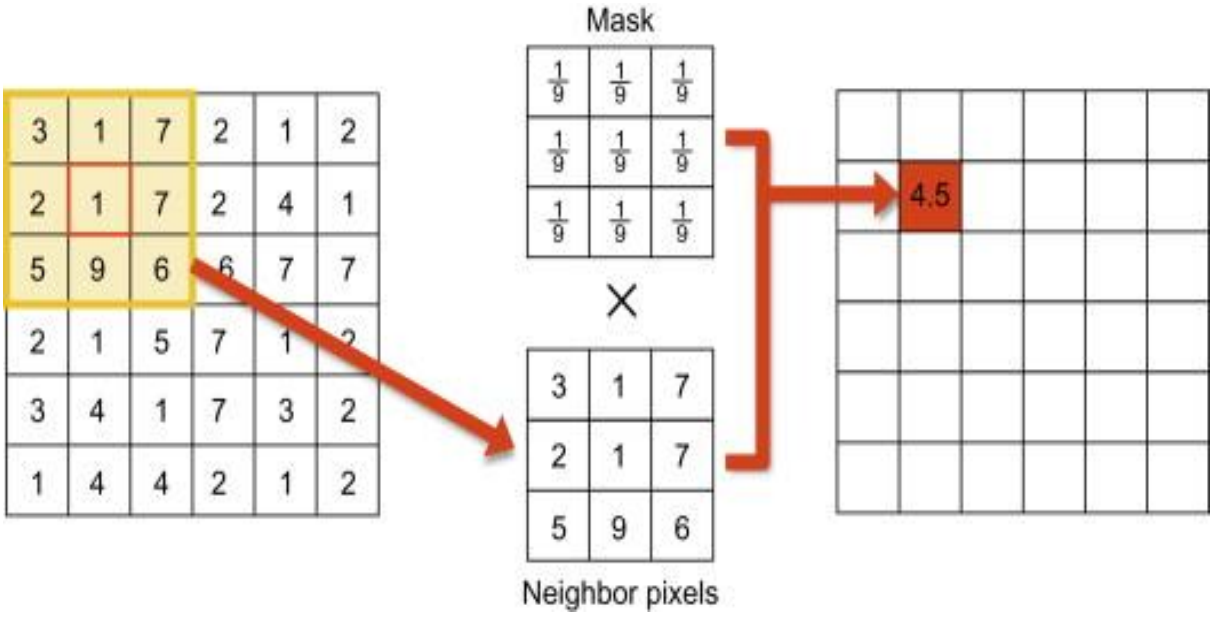


Kernel is also called a mask and it can be of different size depending upon the operation.



We assume a dummy row on the edges of the image in order to process the edge pixels of the input image. The output picture may not have values corresponding to the pixels on the edges if we don't consider a dummy row on the edges. So the resolution of the output image becomes roughly unequal than the resolution of the input image.

In this project, we are taking a 512 x 512 image as input then the output image would of 514 x 514 due to the fact that we have not considered a dummy row on the edges.

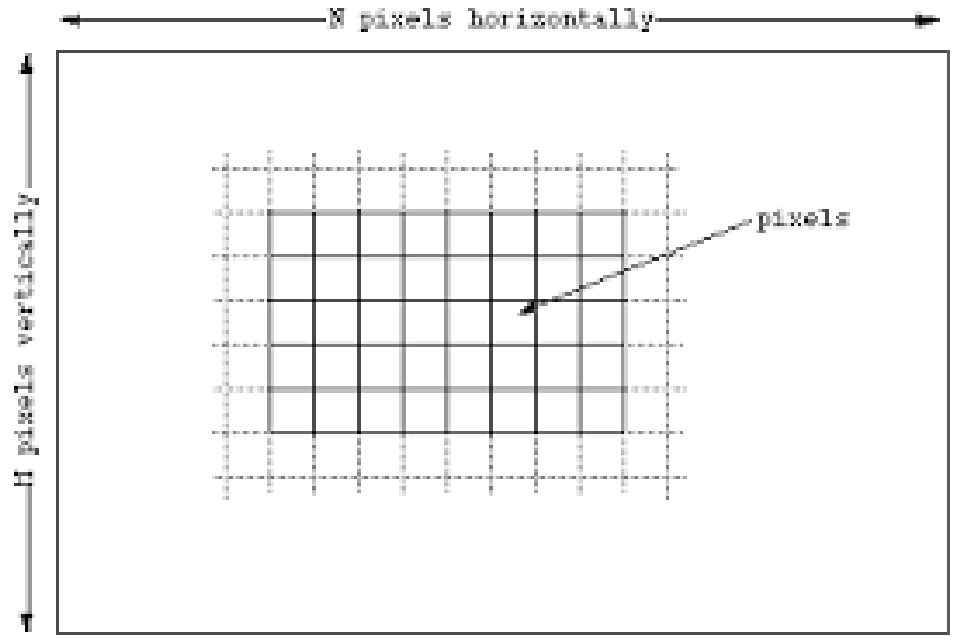


ZYNQ Board

The ZedBoard Znyq-7000 development kit produced by Xilinx Company was used in the study. As a result of grayscale conversion and convolution operations, such applications as edge detection, sharpening and blurring were realized on the test images. Several FPGAs were explored to choose the best development kit for high scale image processing. Although in this project implementation only a 512 x 512 image is used but an image of greater resolution could be easily and rapidly processed through the Zynq Board. This is basically signified in the hardware implementation. Since we have developed the project only on software therefore its complete powerful processing capabilities could not be clearly highlighted.

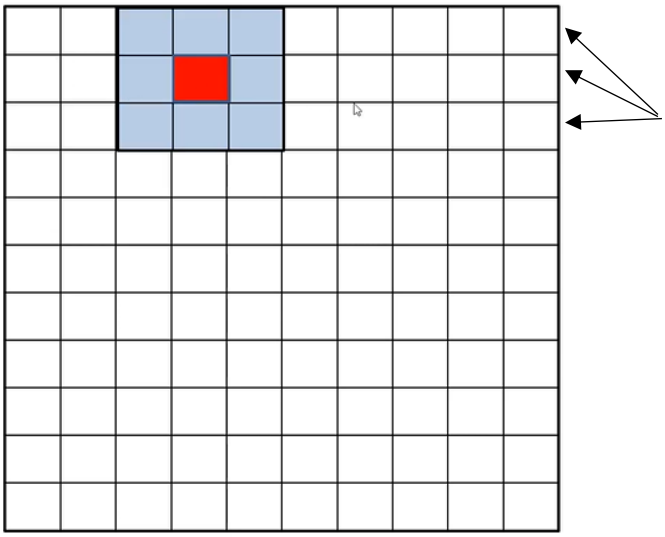
System Architecture / Design

We cannot use pure streaming design for neighborhood processing algorithm since pixels for processing are not consecutive. Hence, we would buffer the pixels inside the IP before processing them. It is not practical to buffer the entire image inside the IP, so we would be buffering just enough pixels for processing. For instance, a 512 x 512 grayscale image requires 262144 Bytes and if we would use a 3x3 kernel then we would just need three image pixel lines (rows) to start processing.

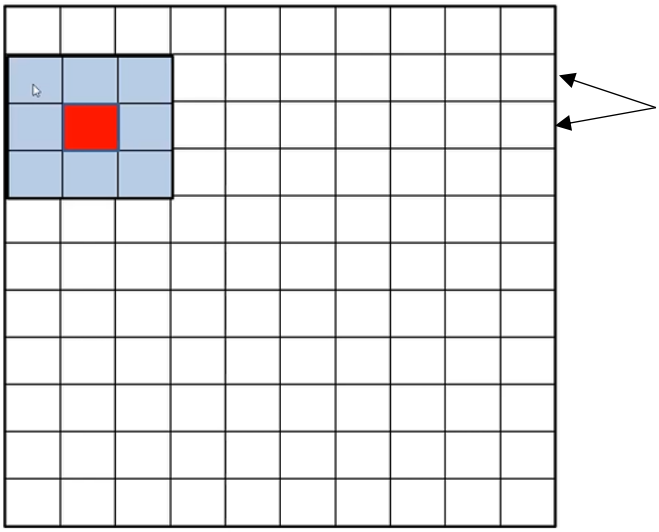


Line Buffers

Line buffers are inside FPGA used for storing one line/row of input image. For a 512 x 512 image, one line buffer would be of 512 Bytes in size and three line buffers would require 1536 Bytes. In actual, the size of the line buffer depends only on the width of the image. From DDR we will send three rows of pixels first to IP for processing, then IP would process it and will send back the processed pixels to DDR. Then we will send the data of next three line buffers. In convolution of neighborhood processing on image, we might have to reuse the same rows of image pixels multiple times, so we would use multiplexer to reuse the same line buffer for pixel processing. For better performance, we would not be sending same pixel rows again and again.

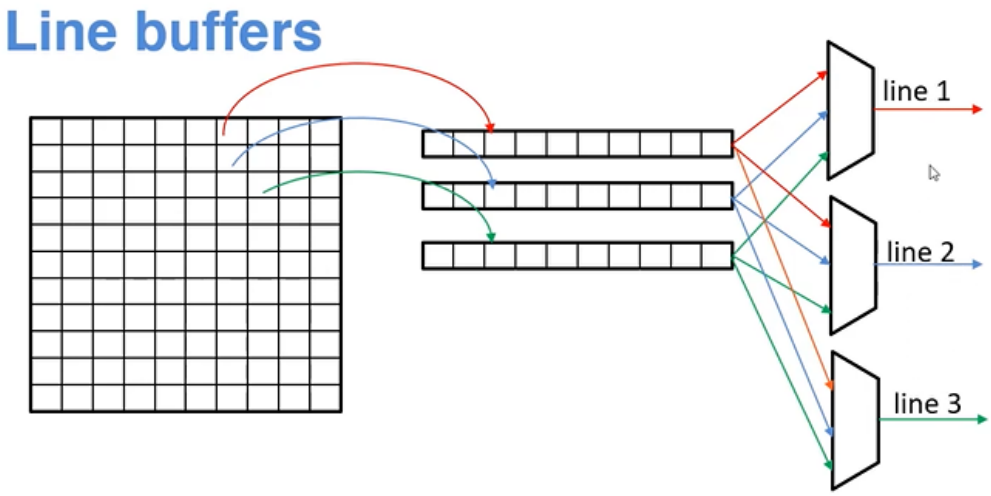


First we would process these three pixel rows

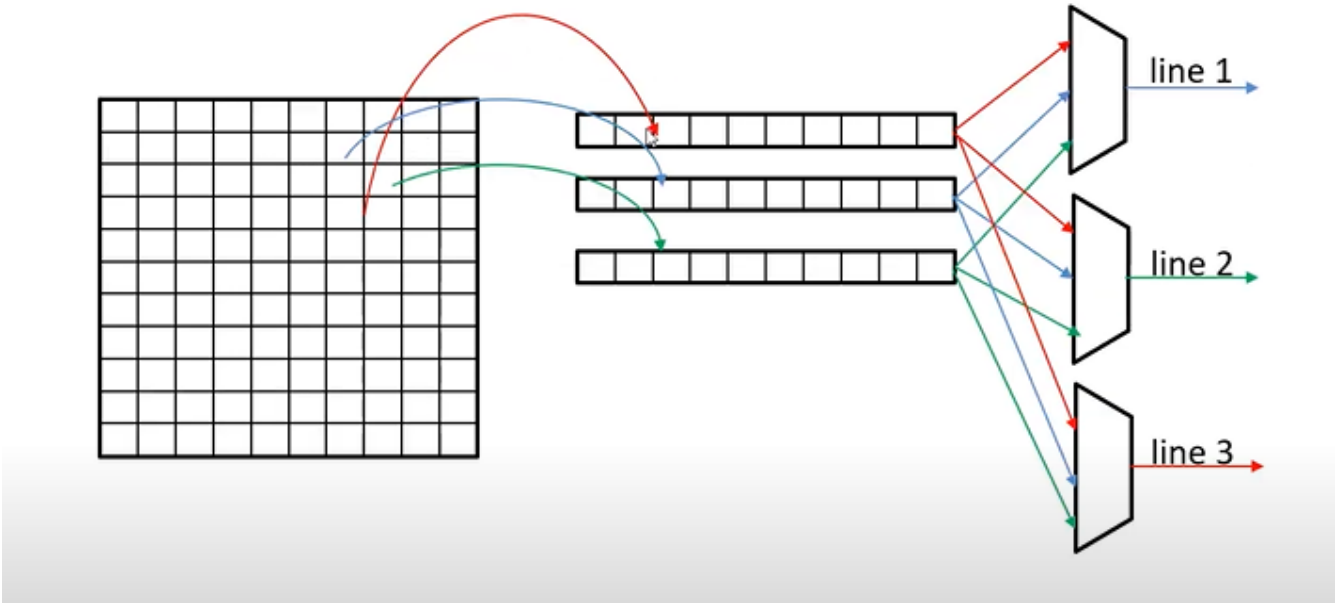


These two pixel rows would be needed again.

When we first perform convolution, the first three pixel lines would be used.



Then the first line buffer would be replaced with the fourth pixel row and the multiplexers would be adjusted in such a way that that now the second line buffer would be chosen as first pixel row, third line buffer as second pixel row and the first line buffer as the third pixel row. This would be done through select lines.



We can improve the system performance by adding a fourth line buffer. While the three line buffers are being processed, the next pixel row can be send to the fourth buffer so that data transfer happens in parallel to data processing.

DDR

It is basically our computer memory. The input image is stored in the computer and full path is provided in the code in order to process it.

DMA Controller and AXI Interface

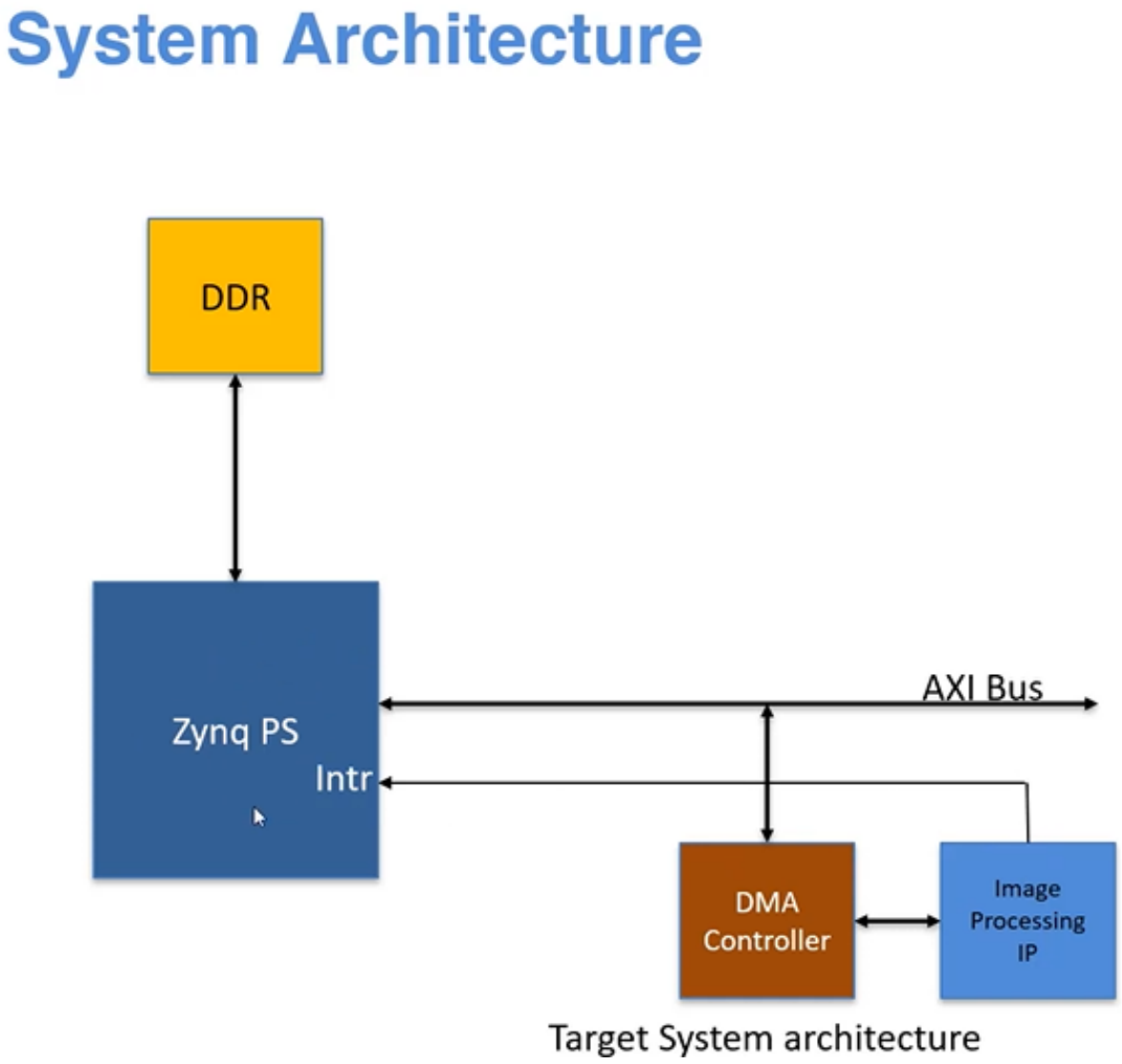
The Xilinx IP AXI Direct Memory Access (AXI DMA) core is a soft Xilinx IP core for use within the Xilinx Vivado Design Suite. The AXI DMA provides high-bandwidth direct memory access between memory (line buffers) and AXI4-Stream target peripherals. While packaging the image processing IP, the input output ports declared in the code are mapped with the AXI Interface. Primary high-speed DMA data movement between system memory (DDR) and stream target (Image Processing IP) is through the AXI4 Read Master to AXI4 memory-mapped to stream (MM2S) Master, and AXI stream to memory-mapped (S2MM) Slave to AXI4 Write Master. AXI DMA also enables up to 16 multiple channels of data movement on both MM2S and S2MM paths. This interface is basically used to stream pixel data to and fro the Image Processing IP rapidly without any delay and stored in the system memory. This interface would be helpful when we implement the design logic on hardware.

Interrupt based processing

We would have an interrupt signal coming from the image processing IP to the Zynq PS.

Initially the image is stored in DDR. Then we would configure the DMA Controller to stream the first four pixel rows to the image processing IP. Convolution is performed in the image processing IP and once it is finished then interrupt would be send to the processor (Zynq PS). After the interrupt is received, then next line of pixel row would be sent to the IP. After convolution of one row of pixel data, the IP would be streaming the processed pixels to DDR through AXI4 interface of the DMA Controller.

Block representation of the System Logic



Target System architecture

Verilog Coding

Xilinx IP provides different kinds of RAMs but the kind of flexibility we need is not provided by Xilinx IP, so we have written our own code for line buffers.

1. Designing Line Buffer

We have made the following assumptions:

1. One pixel at a time is written to the line buffer.
2. Size of the image is 512 x 512 , so we have to store 512 pixels in the line buffers.
3. For a 3 x 3 kernel, convolution is performed on three pixels. So three pixels are read in one short which are 24 bits of data.

Based on these assumptions, the following Verilog code is written.

lineBuffer.v

E:/ISE_147/vivado/software/Vivado/2014.2/examples/Vivado_Tutorial/Projects/image_processing_zynq/image_processing_zynq.srcs/sources_1/new/lineBuffer.v

```
21
22 //image is of 512x512 pixels
23
24 module lineBuffer(
25 input i_clk,
26 input i_rst, //active high reset
27 input [7:0] i_data, //write port (one byte pixel written to line buffer (composed of 8 bits))
28 input i_data_valid, //indicates whether the pixel data received is valid or not
29 output [23:0] o_data, //read port (three pixels are read from line buffer in one short for convolution (24 bits))
30 input i_rd_data //indicates that data should be read from the line buffer
31     );
32
33
34     reg [7:0] line [511:0]; //line buffer (memory) where the pixel data would be stored
35     reg [8:0] wrPntr;   //write pointer(tells in which memory location of the line buffer data will be stored)
36     reg [8:0] rdPntr;   //read pointer
37
38     //writing in memory part
39
40     always@(posedge i_clk)
41     begin
42         if(i_data_valid) //if there is a valid data then
43             line[wrPntr] <= i_data; //that data would be stored in a memory location pointed by the
44                                     //wrtPntr
45     end
46
47
48     always@(posedge i_clk)
49     begin
50         if(i_rst)
51             wrPntr <= 'd0; //initially the wrtPntr is pointing to the first memory location
52         else if(i_data_valid)
53             wrPntr <= wrPntr + 'd1; //then wrtPntr will be incremented afterwards
54     end
55
```

```
56
57      //reading from memory part
58
59      assign o_data = {line[rdPntr],line[rdPntr+1],line[rdPntr+2]}; //3 bytes read from memory (3 pixels = 24 bits))
60      //to read these three bytes we have not used sequential block because that would give us latency between reading the pixel value
61      //and incrementing the read pointer
62
63
64      always@(posedge i_clk)
65          begin
66              if(i_rst)
67                  rdPntr <= 'd0; //initially the rdPntr will read from the first memory location
68              else if(i_rd_data)
69                  rdPntr <= rdPntr + 'd1; //afterwards the read pointer would be incremented to read the next pixel
70          end
71
72
73
74  endmodule
```

2. Designing Multiply Accumulate Model (MAC)

MAC unit is a fundamental block in the computing devices, especially Digital Signal Processor. MAC unit performs multiplication and accumulation process. Basic MAC unit consists of multiplier, adder, and accumulator. Here the MAC unit would perform the convolution and multiplication with kernel of the pixel data read from the line buffers.

a. Box Blur Filter

A box blur (also known as a box linear filter) is a spatial domain linear filter in which each pixel in the resulting image has a value equal to the average value of its neighboring pixels in the input image. It is a form of low-pass ("blurring") filter. The simplest low-pass filter just calculates the average of a pixel and all of its eight immediate neighbors. The result replaces the original value of the pixel. The process is repeated for every pixel in the image.

A 3 by 3 box blur kernel can be written as matrix.

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

This kernel is multiplied with the pixel data of the input image and a blurred output image is obtained.

conv.v

E:/ISE_147/vivado/software/Vivado/2014.2/examples/Vivado_Tutorial/Projects/image_processing_zynq/image_processing_zynq.srcs/source

```
////////////////////////////////////////////////////////////////////////////////////
//box blur operation

module conv(
input i_clk,
input [71:0] i_pixel_data, // we cannot declare as 2D herewe have flatten it
                           // 24 x 3 = 72 bits (3 pixels)
input i_pixel_data_valid, //indicates the pixel data is valid
output reg [7:0] o_convolved_data, //one output pixel(8 bits)
output reg o_convolved_data_valid //indictates that the output data is valid
    );



    reg [7:0] kernel [8:0]; //box blur kernel, 2D
    integer i;
    reg [15:0] multData[8:0]; //to store mltiplication(with kernel)data
    reg [15:0] sumDataInt; //16 bits
    reg [15:0] sumData; //16 bits to store the pixel values after multiplicaton with kernel
    reg multDataValid;
    reg sumDataValid;
    reg convolved_data_valid;



    //initializing kernel

    initial
    begin
        for(i = 0; i < 9; i = i+1)
         begin
            kernel[i] = 1;
         end
     end

    //multiplicationh kernel

    always@(posedge i_clk)
    begin
       for(i = 0; i < 9; i = i+1)
            begin
               multData[i] <= kernel[i]*i_pixel_data[i*8+:8];
            end
        multDataValid <= i_pixel_data_valid;
    end


    //addition to calculate output pixel

    always@(*)
    begin
       sumDataInt = 0; //intermediate sum data
        for(i = 0; i < 9; i = i+1)
             begin
             sumDataInt = sumDataInt + multData[i];
             end
     end
   //intermediate and final sum data are different because we cannot mix the blocking and non-blocking assignment
   //also the sumData should be initially zero

     always@(posedge i_clk)
     begin
          sumData <= sumDataInt; //final sum data
          sumDataValid <= multDataValid;
     end


     //divison by 9

     always@(posedge i_clk)
     begin
       o_convolved_data <= sumData/9; //only integer part will be assigned
       o_convolved_data_valid <= sumDataValid;
     end



endmodule
```

b. Edge Detection using Sobel Operator

The Sobel operator is used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasizing edges. The operator uses two 3×3 kernels which are convolved with the original image to calculate the output pixels. The output image can be viewed as a bitmap image, with the areas of high gradient (the likely edges) visible as white lines. If we define A as the source image, and Gx and Gy are two images which at each point contain the vertical and horizontal derivative approximations respectively.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

The magnitude of the resultant pixel is calculated through the following:

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$

```
22 //edge detection
23 module edge_detection(
24 input         i_clk,
25 input [71:0] i_pixel_data,
26 input         i_pixel_data_valid,
27 output reg [7:0] o_convolved_data, //threshold pixel value
28 output reg   o_convolved_data_valid
29     );
30
31 integer i;
32 reg [7:0] kernel1 [8:0]; //two 2D kernels
33 reg [7:0] kernel2 [8:0];
34 reg [10:0] multData1[8:0]; //to represent max. mult data value (1 bit is for sign)
35 reg [10:0] multData2[8:0]; //8 bits of pixel + max negative no. in kernel is -2 (represented in 3 bits)
36 reg [10:0] sumDataInt1; //so 8+3 = 11 bits are required
37 reg [10:0] sumDataInt2;
38 reg [10:0] sumData1;
39 reg [10:0] sumData2;
40 reg multDataValid;
41 reg sumDataValid;
42 reg convolved_data_valid;
43 reg [20:0] convolved_data_int1;
44 reg [20:0] convolved_data_int2;
45 wire [21:0] convolved_data_int;
46 reg convolved_data_int_valid;
```

```
initial
begin
    kernel1[0] =  1; //first kernel
    kernel1[1] =  0;
    kernel1[2] = -1;
    kernel1[3] =  2;
    kernel1[4] =  0;
    kernel1[5] = -2;
    kernel1[6] =  1;
    kernel1[7] =  0;
    kernel1[8] = -1;

    kernel2[0] =  1; //second kernel
    kernel2[1] =  2;
    kernel2[2] =  1;
    kernel2[3] =  0;
    kernel2[4] =  0;
    kernel2[5] =  0;
    kernel2[6] = -1;
    kernel2[7] = -2;
    kernel2[8] = -1;
end

always @(posedge i_clk)
begin
    for(i=0;i<9;i=i+1)
    begin //multiplication with both the kernels is done in parallel (here signed
            //multiplier is use because we have negative numbers in kernel
        //if signed multiplier is not use then vivado will take the negative number in kernel
        //as unsigned number
        multData1[i] <= $signed(kernel1[i])*$signed({1'b0,i_pixel_data[i*8+:8]});
        multData2[i] <= $signed(kernel2[i])*$signed({1'b0,i_pixel_data[i*8+:8]});
    end
    multDataValid <= i_pixel_data_valid;
end

always @(*)
begin
    sumDataInt1 = 0; //intermediate sum of the pixel values after multiplying by kernel
    sumDataInt2 = 0;
    for(i=0;i<9;i=i+1)
    begin //sum is also done in parallel and is signed addition
        sumDataInt1 = $signed(sumDataInt1) + $signed(multData1[i]);
        sumDataInt2 = $signed(sumDataInt2) + $signed(multData2[i]);
    end
end

always @(posedge i_clk)
begin
    sumData1 <= sumDataInt1;
    sumData2 <= sumDataInt2;
    sumDataValid <= multDataValid;
end

always @(posedge i_clk)
begin //to compute the final pixel value after convolution (magnitude of pixel)
    convolved_data_int1 <= $signed(sumData1)*$signed(sumData1);
    convolved_data_int2 <= $signed(sumData2)*$signed(sumData2);
    convolved_data_int_valid <= sumDataValid;
end

assign convolved_data_int = convolved_data_int1+convolved_data_int2; //the squares of Gx and Gy are added (threshold)

////we have defined in this block that if threshold is above 4000 than
//the pixel would be white otherwise black
always @(posedge i_clk)
begin
    if(convolved_data_int > 4000)
        o_convolved_data <= 8'hff; //output pixel is of 8 bits
    else
        o_convolved_data <= 8'h00;
    o_convolved_data_valid <= convolved_data_int_valid;
end

endmodule
```

c. Unsharp masking

Unsharp masking (USM) is an image sharpening technique. Sharpening an image increases the contrast between bright and dark regions to bring out features. The following array is a kernel for a common high pass filter used to sharpen an image:

Sharpen filter $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

```
21 //sharp image operation
22
23 module sharpen(
24 input i_clk,
25 input [71:0] i_pixel_data, // we cannot declare as 2D hereve have flatten it
26                            // 24 x 3 = 72 bits (3 pixels)
27 input i_pixel_data_valid, //indicates the pixel data is valid
28 output reg [7:0] o_convolved_data, //one output pixel(8 bits)
29 output reg o_convolved_data_valid //indictates that the output data is valid
30     );
31
32
33
34     reg [7:0] kernel [8:0]; //box blur kernel, 2D
35     integer i;
36     reg [10:0] multData[8:0]; //to store mltiplication(with kernel)data
37     reg [10:0] sumDataInt; //11 bits (signed data)
38     reg [10:0] sumData; //16 bits to store the pixel values after multiplicaton with kernel
39     reg multDataValid;
40     reg sumDataValid;
41     reg convolved_data_valid;
42
```

```
44
45      //initializing kernel
46
47      initial
48          begin
49              kernel[0] =  0;
50              kernel[1] =  -1;
51              kernel[2] = 0;
52              kernel[3] =  -1;
53              kernel[4] =  5;
54              kernel[5] = -1;
55              kernel[6] =  0;
56              kernel[7] =  -1;
57              kernel[8] = 0;
58
59
60           end
61
62       //multiplicationh kernel
63
64       always@(posedge i_clk)
65       begin
66          for(i = 0; i < 9; i = i+1)
67                begin
68                   multData[i] <= $signed(kernel[i])*$signed({1'b0,i_pixel_data[i*8+:8]});
69                end
70           multDataValid <= i_pixel_data_valid;
71       end
72
74       //addition to calculate output pixel
75
76       always@(*)
77       begin
78          sumDataInt = 0; //intermediate sum data
79           for(i = 0; i < 9; i = i+1)
80                begin
81                sumDataInt = $signed(sumDataInt) + $signed(multData[i]);
82                end
83       end
84      //intermediate and final sum data are different because we cannot mix the blocking and non-blocking assignment
85      //also the sumData should be initially zero
86
87       always@(posedge i_clk)
88       begin
89             sumData <= sumDataInt; //final sum data
90             sumDataValid <= multDataValid;
91       end
92
93
94       //sending the output pixel data
95
96       always@(posedge i_clk)
97       begin
98         o_convolved_data <= sumData;
99         o_convolved_data_valid <= sumDataValid;
100      end
101
102
103 endmodule
```

d. High Pass Filtering

A high pass filter tends to retain the high frequency information within an image while reducing the low frequency information. The kernel of the high pass filter is designed to increase the brightness of the center pixel relative to neighboring pixels. The kernel array usually contains a single positive value at its center, which is completely surrounded by negative values. Below is just an example of one possible kernel for a high pass filter. Other filters may include more weighting for the center point.

	A_1	A_2	A_3
1	-7/9	-7/9	0
2	0	5	0
3	0	-7/9	-7/9

```
23 module brightness(
24 input i_clk,
25 input [71:0] i_pixel_data, // we cannot declare as 2D hereve have flatten it
26                           // 24 x 3 = 72 bits (3 pixels)
27 input i_pixel_data_valid, //indicates the pixel data is valid
28 output reg [7:0] o_convolved_data, //one output pixel(8 bits)
29 output reg o_convolved_data_valid //indictates that the output data is valid
30     );
31
32
33
34     reg [7:0] kernel [8:0]; // kernel, 2D
35     integer i;
36     reg [10:0] multData[8:0]; //to store mltiplication(with kernel)data
37     reg [10:0] sumDataInt; //11 bits (signed data)
38     reg [10:0] sumData; // to store the pixel values after multiplicaton with kernel
39     reg multDataValid;
40     reg sumDataValid;
41     reg convolved_data_valid;
42
43
44
45     //initializing kernel
46
47     initial
48         begin
49             kernel[0] =  -7;
50             kernel[1] =  -7;
51             kernel[2] = 0;
52             kernel[3] =  0;
53             kernel[4] =  5;
54             kernel[5] =  0;
55             kernel[6] =  0;
56             kernel[7] =  -7;
57             kernel[8] = -7;
```

```
60          end
61
62      //multiplicationh kernel
63
64      always@(posedge i_clk)
65      begin
66         for(i = 0; i < 9; i = i+1)
67              begin
68                 multData[i] <= $signed(kernel[i])*$signed({1'b0,i_pixel_data[i*8+:8]});
69              end
70          multDataValid <= i_pixel_data_valid;
71      end
72
73
74      //addition to calculate output pixel
75
76      always@(*)
77      begin
78         sumDataInt = 0; //intermediate sum data
79          for(i = 0; i < 9; i = i+1)
80                begin
81                sumDataInt = $signed(sumDataInt) + $signed(multData[i]);
82                end
83       end
84     //intermediate and final sum data are different because we cannot mix the blocking and non-blocking assignment
85     //also the sumData should be initially zero
86
87       always@(posedge i_clk)
88       begin
89             sumData <= sumDataInt; //final sum data
90             sumDataValid <= multDataValid;
91       end
92
93
94       //sending the output pixel data
95
96       always@(posedge i_clk)
97       begin
98         o_convolved_data <= sumData/9;
99         o_convolved_data_valid <= sumDataValid;
100      end
101
102 endmodule
103
```

e. Low Pass Filtering

A low pass filter provides the basis for smoothing operations. If an image contains too many variations to be able to determine specific features, smoothing can decrease the contrast so that some areas (especially the background) will not distract from viewing other areas of the image. Through low pass filtering kernel we have basically decreased the brightness of output image.

	A_1	A_2	A_3
1	0	0	0
2	0	5/9	0
3	0	0	0

```
23 module low_pass(
24 input i_clk,
25 input [71:0] i_pixel_data, // we cannot declare as 2D hereve have flatten it
26                           // 24 x 3 = 72 bits (3 pixels)
27 input i_pixel_data_valid, //indicates the pixel data is valid
28 output reg [7:0] o_convolved_data, //one output pixel(8 bits)
29 output reg o_convolved_data_valid //indictates that the output data is valid
30     );
31
32
33
34     reg [7:0] kernel [8:0]; // kernel, 2D
35     integer i;
36     reg [10:0] multData[8:0]; //to store mltiplication(with kernel)data
37     reg [10:0] sumDataInt; //11 bits (signed data)
38     reg [10:0] sumData; // to store the pixel values after multiplicaton with kernel
39     reg multDataValid;
40     reg sumDataValid;
41     reg convolved_data_valid;
42
43
```

```
    //initializing kernel

    initial
        begin
            kernel[0] =  0;
            kernel[1] =  0;
            kernel[2] = 0;
            kernel[3] =  0;
            kernel[4] =  5;
            kernel[5] =  0;
            kernel[6] =  0;
            kernel[7] =  0;
            kernel[8] = 0;


         end

     //multiplicationh kernel

     always@(posedge i_clk)
     begin
        for(i = 0; i < 9; i = i+1)
              begin
                 multData[i] <= $signed(kernel[i])*$signed({1'b0,i_pixel_data[i*8+:8]});
              end
         multDataValid <= i_pixel_data_valid;
     end

   //intermediate and final sum data are different because we cannot mix the blocking and non-blocking assignment
   //also the sumData should be initially zero

    always@(posedge i_clk)
    begin
          sumData <= sumDataInt; //final sum data
          sumDataValid <= multDataValid;
    end


     //sending the output pixel data

     always@(posedge i_clk)
     begin
       o_convolved_data <= sumData/9;
       o_convolved_data_valid <= sumDataValid;
     end
     endmodule
```

3. Designing Control Logic

This Control Unit designed instantiates all the line buffers and the multiplexers. It controls storing pixel data into line buffers as well as it controls data to be send to the MAC Module.

```
module imageControl(
input                   i_clk,
input                   i_rst,
input [7:0]             i_pixel_data, //input pixel stream to be stored in line buffers
input                   i_pixel_data_valid, //indicating the validity of incoming pixels data
output reg [71:0]       o_pixel_data, //72 pixels  ( 24x3) pixels are output
output                  o_pixel_data_valid, //indcates the validity of output pixel data
output reg              o_intr
);

reg [8:0] pixelCounter; //it counts from 0 to 511. one line buffer should have 512 pixels.(0-511 represented in 9 bits)
                    //when the counter reaches 511, pixels are stored in the next line buffer.

reg [1:0] currentWrLineBuffer; //to count the number of line buffer (0-3)
reg [3:0] lineBuffDataValid; //to determine valid data to be written in each line buffer (0-3)
reg [3:0] lineBuffRdData; //to determine valid data to be read from 3 line buffers in parallel
reg [1:0] currentRdLineBuffer; //to determine from which line buffer data should be read

wire [23:0] lb0data;
wire [23:0] lb1data;
wire [23:0] lb2data;
wire [23:0] lb3data;

reg [8:0] rdCounter; //counts from 0 upto 511. Determines the number of pixels read from the current line buffer
reg rd_line_buffer; //enable signal sent to three line buffers at a time from where data should be read
reg [11:0] totalPixelCounter; //determines the valid data in all line buffers (512x4) 12 bits are required
reg rdState;

localparam IDLE = 'b0,
           RD_BUFFER = 'b1;

assign o_pixel_data_valid = rd_line_buffer; //whenever rd_line_buffer signal becomes high, we would have valid output
                                            //which is being read from the line buffers

always @(posedge i_clk) //determines the total data in all the line buffers
begin
    if(i_rst)
        totalPixelCounter <= 0;
    else
    begin
        if(i_pixel_data_valid & !rd_line_buffer) //we have data but not reading from the line buffers
            totalPixelCounter <= totalPixelCounter + 1;
        else if(!i_pixel_data_valid & rd_line_buffer) //we dont have data but we are reading from line buffers
            totalPixelCounter <= totalPixelCounter - 1;
    end
end

//state machine to control the enable signal of the multiplexer (currentRdLineBuffer)
always @(posedge i_clk)
begin
    if(i_rst)
    begin
        rdState <= IDLE; //initially no enable signal is send which would indicate no data is read
        rd_line_buffer <= 1'b0; //from any of the line buffers in IDLE state
        o_intr <= 1'b0;
    end
    else
    begin
        case(rdState)
            IDLE:begin
                o_intr <= 1'b0;
                if(totalPixelCounter >= 1536) //512x3 =1536
                begin                         //we should have data in three line buffers
                    rd_line_buffer <= 1'b1; //to go to the RD_BUFFER state where pixels are read and enable signal to read data is made high
                    rdState <= RD_BUFFER;
                end
            end
```

```
            RD_BUFFER:begin
                if(rdCounter == 511) //checks that 512 pixels are read from one line buffer
                begin
                    rdState <= IDLE;
                    rd_line_buffer <= 1'b0; //signal is made low again because we finished reading
                    o_intr <= 1'b1;
                end
            end
        endcase
    end
end


always @(posedge i_clk) //this block is used to count pixels (8 bits) one by one
begin
    if(i_rst)
        pixelCounter <= 0; //initially the pixel counter is set to zero
    else
    begin
        if(i_pixel_data_valid)
            pixelCounter <= pixelCounter + 1; //input pixel stream to be stored in line buffers are incremented
                                              //when the pixel_data is valid
    end
end


always @(posedge i_clk)  //block which tells in which line buffer the pixels would be saved
begin
    if(i_rst)
        currentWrLineBuffer <= 0; //initially pixels are stored in line buffer 0
    else
    begin
        if(pixelCounter == 511 & i_pixel_data_valid)
            currentWrLineBuffer <= currentWrLineBuffer+1; //when the pixel counter reaches 511, pixels are saved in the next line buffer
    end
end


always @(*)
begin
    lineBuffDataValid = 4'h0; //all of the data valid signal of line buffer should be zero
    lineBuffDataValid[currentWrLineBuffer] = i_pixel_data_valid; //except for the current line buffer in which pixel data is written
end

always @(posedge i_clk) //this block determines how much data has been read from the current line buffer
begin
    if(i_rst)
        rdCounter <= 0;
    else
    begin
        if(rd_line_buffer)
            rdCounter <= rdCounter + 1; //increments by 1 after each pixel is read
    end
end

always @(posedge i_clk) //this logic controls the currentRdLineBuffer
begin
    if(i_rst)
    begin
        currentRdLineBuffer <= 0; //initially the data is read from line buffer 0
    end
    else
    begin
        if(rdCounter == 511 & rd_line_buffer) //after reading 0-511 pixels from the current line buffer
            currentRdLineBuffer <= currentRdLineBuffer + 1; //data is read from the nest line buffer
    end
```

```
always @(*) //to determine from which line buffer data should be read
begin
    case(currentRdLineBuffer)
        0:begin
            o_pixel_data = {lb2data,lb1data,lb0data}; //data is read from line buffers 0,1,2
        end
        1:begin
            o_pixel_data = {lb3data,lb2data,lb1data}; //data is read from line buffers 1,2,3
        end
        2:begin
            o_pixel_data = {lb0data,lb3data,lb2data}; //data is read from line buffers 2,3,0
        end
        3:begin
            o_pixel_data = {lb1data,lb0data,lb3data}; ////data is read from line buffers 3,0,1
        end
    endcase
end

//implementing multiplexer logic
always @(*) //this block is used to fetch data for the fourth line buffer
begin
    case(currentRdLineBuffer) //this is the control signal to the multiplexer
        0:begin
            lineBuffRdData[0] = rd_line_buffer; //rd_line_buffer is enable signal sent to indicate
            lineBuffRdData[1] = rd_line_buffer;  //that data should be read from these line buffers
            lineBuffRdData[2] = rd_line_buffer;
            lineBuffRdData[3] = 1'b0;
        end
         1:begin
              lineBuffRdData[0] = 1'b0;
              lineBuffRdData[1] = rd_line_buffer;
              lineBuffRdData[2] = rd_line_buffer;
              lineBuffRdData[3] = rd_line_buffer;
          end
         2:begin
               lineBuffRdData[0] = rd_line_buffer;
               lineBuffRdData[1] = 1'b0;
               lineBuffRdData[2] = rd_line_buffer;
               lineBuffRdData[3] = rd_line_buffer;
         end
        3:begin
               lineBuffRdData[0] = rd_line_buffer;
               lineBuffRdData[1] = rd_line_buffer;
               lineBuffRdData[2] = 1'b0;
               lineBuffRdData[3] = rd_line_buffer;
         end
     endcase
end

  //instantiation of line buffers
lineBuffer lB0(
    .i_clk(i_clk),
    .i_rst(i_rst),
    .i_data(i_pixel_data),
    .i_data_valid(lineBuffDataValid[0]),
    .o_data(lb0data),
    .i_rd_data(lineBuffRdData[0])
);

lineBuffer lB1(
    .i_clk(i_clk),
    .i_rst(i_rst),
    .i_data(i_pixel_data),
    .i_data_valid(lineBuffDataValid[1]),
    .o_data(lb1data),
    .i_rd_data(lineBuffRdData[1])
 );

 lineBuffer lB2(
     .i_clk(i_clk),
     .i_rst(i_rst),
     .i_data(i_pixel_data),
     .i_data_valid(lineBuffDataValid[2]),
     .o_data(lb2data),
     .i_rd_data(lineBuffRdData[2])
  );

  lineBuffer lB3(
      .i_clk(i_clk),
      .i_rst(i_rst),
      .i_data(i_pixel_data),
      .i_data_valid(lineBuffDataValid[3]),
      .o_data(lb3data),
      .i_rd_data(lineBuffRdData[3])
   );

endmodule
```

4. Image Processing IP Packaging

Image Processing IP is interfaced with DMA Controller which has an AXI Stream interface. Data comes in and is sent via this interface. Built-in fifo generator is used from the IP catalog as an output buffer, in order to minimize any mismatch which would occur if DMA Controller is not ready to accept output pixel data from the Image Processing IP. So the output data is not sent directly to the DMA Controller from the IP, it is stored in the output buffer and is sent from it. Once a line buffer has finished processing the pixels, interrupt is sent to the processor from that line buffer which indicates that this line buffer is free now and next pixel row should be sent in to this line buffer. When interrupt signal is high, it represents that a line buffer is free. Ports are mapped in IP Packaging.

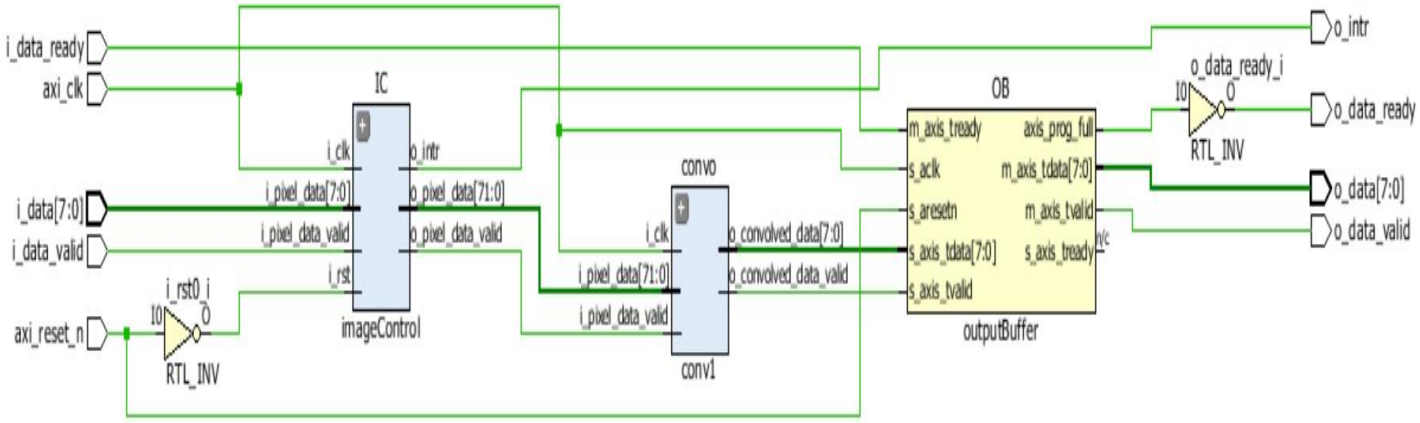
```
module imageProcessTop(
input   axi_clk,
input   axi_reset_n,
//slave interface //interface from which data would be coming through DMA Controller to IP
input   i_data_valid,
input [7:0] i_data,
output  o_data_ready, //this ready signal is given to DMA Controller when we are exiting data
//master interface //data is send back to DMA Controller from IP through this interface
output  o_data_valid,
output [7:0] o_data,
input   i_data_ready, //this ready signal is coming from DMA Controller
//interrupt
output  o_intr //to indicate that line buffer is free

    );

wire [71:0] pixel_data;
wire pixel_data_valid;
wire axis_prog_full;
wire [7:0] convolved_data; //output after convolution
wire convolved_data_valid;

assign o_data_ready = !axis_prog_full; //output buffer accepts data until not half full
                                       //axis_prog_full signal is asserted when DMA signal is not accepting data

 //instantiation of modules
imageControl IC(
    .i_clk(axi_clk),
    .i_rst(!axi_reset_n), //active low reset
    .i_pixel_data(i_data),
    .i_pixel_data_valid(i_data_valid),
    .o_pixel_data(pixel_data),
    .o_pixel_data_valid(pixel_data_valid),
    .o_intr(o_intr)
  );

  conv1 convo(
      .i_clk(axi_clk),
      .i_pixel_data(pixel_data),
      .i_pixel_data_valid(pixel_data_valid),
      .o_convolved_data(convolved_data),
      .o_convolved_data_valid(convolved_data_valid)
   );

  outputBuffer OB( //built-in fifo generator is used from IP Catalog
     .s_aclk(axi_clk),                  // input wire s_aclk
     .s_aresetn(axi_reset_n),           // input wire s_aresetn
     .s_axis_tvalid(convolved_data_valid),    // input wire s_axis_tvalid
     .s_axis_tready(),    // output wire s_axis_tready //there is no case when fifo wont be ready to accept data
     .s_axis_tdata(convolved_data),     // input wire [7 : 0] s_axis_tdata
     .m_axis_tvalid(o_data_valid),    // output wire m_axis_tvalid
     .m_axis_tready(i_data_ready),    // input wire m_axis_tready
     .m_axis_tdata(o_data),    // output wire [7 : 0] m_axis_tdata
     .axis_prog_full(axis_prog_full)  // output wire axis_prog_full
   );



endmodule
```

5. Synthesizing the Design

After coding the design files in verilog, the design was synthesized. The synthesis process checks code syntax, analyzes the hierarchy of the design, recognizes the functionality described, and ensures that the design is optimized for the device architecture selected.

a) RTL Schematic

After the synthesis phase of the design, a schematic representation of the synthesized design is displayed. RTL View is a Register Transfer Level graphical representation of the synthesized design.



b) Utilization Report

```
Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2014.2 (win64) Build 932637 Wed Jun 11 13:33:10 MDT 2014
| Date         : Sun Sep 13 13:11:25 2020
| Host         : DESKTOP-FQ1K1PB running 64-bit major release  (build 9200)
| Command      : report_utilization -file imageProcessTop_utilization_synth.rpt -pb imageProcessTop_utilization_synth.pb
| Design       : imageProcessTop
| Device       : xc7z020
| Design State : Synthesized
------------------------------------------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+------+-------+-----------+-------+
|          Site Type         | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs*                | 1943 |     0 |     53200 |  3.65 |
|   LUT as Logic             |  790 |     0 |     53200 |  1.48 |
|   LUT as Memory            | 1153 |     0 |     17400 |  6.62 |
|     LUT as Distributed RAM | 1152 |     0 |           |       |
|     LUT as Shift Register  |    1 |     0 |           |       |
| Slice Registers            |  230 |     0 |    106400 |  0.21 |
|   Register as Flip Flop    |  230 |     0 |    106400 |  0.21 |
|   Register as Latch        |    0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |   32 |     0 |     26600 |  0.12 |
| F8 Muxes                   |    0 |     0 |     13300 |  0.00 |
+----------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, i


2. Memory
---------

+----------------+------+-------+-----------+-------+
|    Site Type   | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| Block RAM Tile |    0 |     0 |       140 |  0.00 |
|   RAMB36/FIFO* |    0 |     0 |       140 |  0.00 |
|   RAMB18       |    0 |     0 |       280 |  0.00 |
+----------------+------+-------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if

```

```
3. DSP
------

+----------------+------+-------+-----------+-------+
|    Site Type   | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| DSPs           |    2 |     0 |       220 |  0.90 |
|   DSP48E1 only |    2 |       |           |       |
+----------------+------+-------+-----------+-------+


4. IO and GT Specific
---------------------

+-----------------------------+------+-------+-----------+-------+
|          Site Type          | Used | Fixed | Available | Util% |
+-----------------------------+------+-------+-----------+-------+
| Bonded IOB                  |   11 |     0 |       200 |  5.50 |
| Bonded IPADs                |    0 |     0 |         2 |  0.00 |
| Bonded IOPADs               |    0 |     0 |       130 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |         4 |  0.00 |
| PHASER_REF                  |    0 |     0 |         4 |  0.00 |
| OUT_FIFO                    |    0 |     0 |        16 |  0.00 |
| IN_FIFO                     |    0 |     0 |        16 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |         4 |  0.00 |
| IBUFGDS                     |    0 |     0 |       192 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |        16 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |        16 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |       200 |  0.00 |
| ILOGIC                      |    0 |     0 |       200 |  0.00 |
| OLOGIC                      |    0 |     0 |       200 |  0.00 |
+-----------------------------+------+-------+-----------+-------+


5. Clocking
-----------

+------------+------+-------+-----------+-------+
|  Site Type | Used | Fixed | Available | Util% |
+------------+------+-------+-----------+-------+
| BUFGCTRL   |    0 |     0 |        32 |  0.00 |
| BUFIO      |    0 |     0 |        16 |  0.00 |
| MMCME2_ADV |    0 |     0 |         4 |  0.00 |
| PLLE2_ADV  |    0 |     0 |         4 |  0.00 |
| BUFMRCE    |    0 |     0 |         8 |  0.00 |
| BUFHCE     |    0 |     0 |        72 |  0.00 |
| BUFR       |    0 |     0 |        16 |  0.00 |
+------------+------+-------+-----------+-------+


6. Specific Feature
-------------------

+-------------+------+-------+-----------+-------+
|  Site Type  | Used | Fixed | Available | Util% |
+-------------+------+-------+-----------+-------+
| BSCANE2     |    0 |     0 |         4 |  0.00 |
| CAPTUREE2   |    0 |     0 |         1 |  0.00 |
| DNA_PORT    |    0 |     0 |         1 |  0.00 |
| EFUSE_USR   |    0 |     0 |         1 |  0.00 |
| FRAME_ECCE2 |    0 |     0 |         1 |  0.00 |
| ICAPE2      |    0 |     0 |         2 |  0.00 |
| STARTUPE2   |    0 |     0 |         1 |  0.00 |
| XADC        |    0 |     0 |         1 |  0.00 |
+-------------+------+-------+-----------+-------+
```

```
7. Primitives
-------------

+----------+------+---------------------+
| Ref Name | Used | Functional Category |
+----------+------+---------------------+
| RAMD64E  | 1152 |  Distributed Memory |
| LUT6     |  489 |                 LUT |
| FDRE     |  230 |        Flop & Latch |
| LUT2     |  144 |                 LUT |
| LUT4     |   80 |                 LUT |
| LUT3     |   50 |                 LUT |
| CARRY4   |   42 |          CarryLogic |
| LUT5     |   38 |                 LUT |
| LUT1     |   35 |                 LUT |
| MUXF7    |   32 |               MuxFx |
| IBUF     |    9 |                  IO |
| OBUF     |    2 |                  IO |
| DSP48E1  |    2 |    Block Arithmetic |
| SRL16E   |    1 |  Distributed Memory |
+----------+------+---------------------+


8. Black Boxes
--------------

+--------------+------+
|   Ref Name   | Used |
+--------------+------+
| outputBuffer |    1 |
+--------------+------+


9. Instantiated Netlists
------------------------

+----------+------+
| Ref Name | Used |
+----------+------+
```

6. Simulating the Design

Behavioral simulation employs a high level of abstraction to model the design. The testbench reads the data from the bitmap image and stores the processed pixels back in a bitmap image.

a. Testbench

```
`define headerSize 1080//image header
`define imageSize 512*512

module tb(
    );


reg clk;
reg reset;
reg [7:0] imgData;
integer file,file1,i,r;
reg imgDataValid;
integer sentSize;
wire intr;
wire [7:0] outData;
wire outDataValid;
integer receivedData=0;

initial
begin
   clk = 1'b0;
   forever
   begin
       #5 clk = ~clk; //after every 5ns
   end
end
```

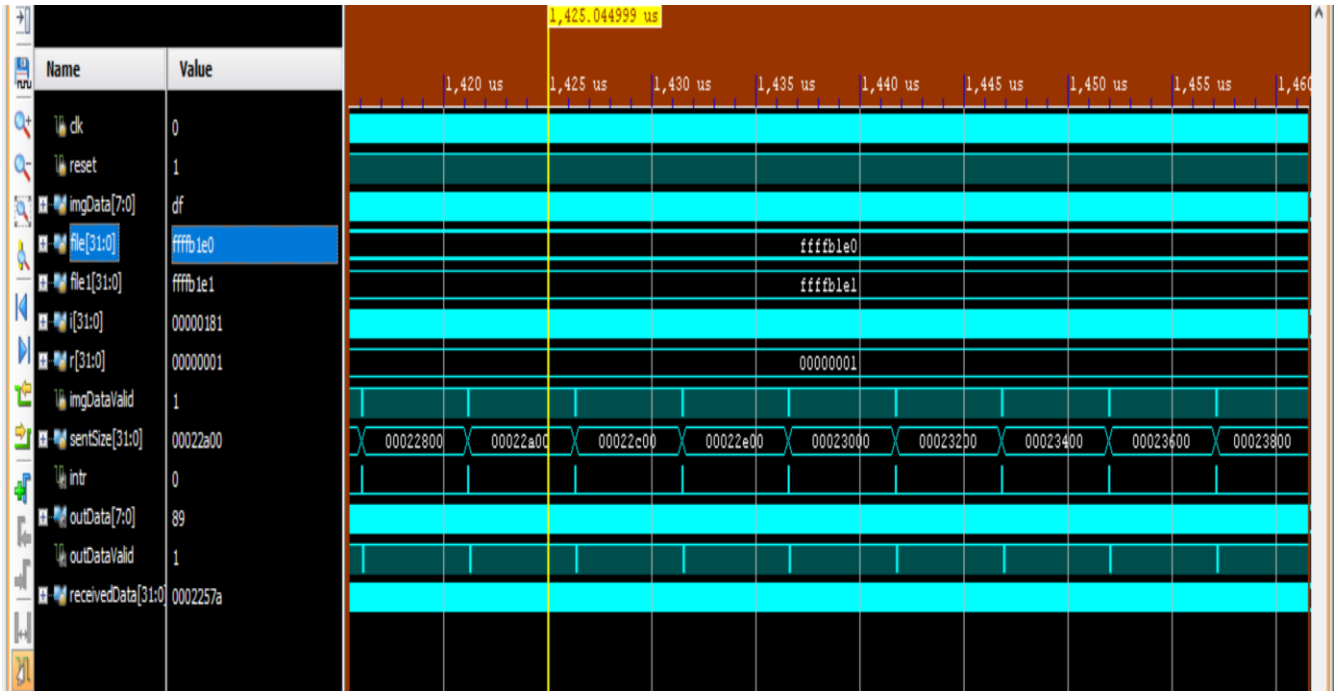
```
initial
begin
   reset = 0; //reset is active low
   sentSize = 0;
   imgDataValid = 0;
   #100;
   reset = 1; //reset is disabled after 100ns
   #100;
   file = $fopen("C:/Users/hp/Desktop/clown.bmp","rb"); //input image is open
   //file1 = $fopen("C:/Users/hp/Desktop/blurred_lena.bmp","wb");
   //file1 = $fopen("C:/Users/hp/Desktop/brightness_clown.bmp","wb");
   file1 = $fopen("C:/Users/hp/Desktop/inverted_clown.bmp","wb"); //output image is open

   for(i=0;i<`headerSize;i=i+1) //the header part of image is not sent for processing
   begin
      r = $fscanf(file,"%c",imgData); //1 byte is read from the input image
      //$display(file1);
      $fwrite(file1,"%c",imgData); //1 byte is written into output image
      //$display(file);
   end

   for(i=0;i<4*512;i=i+1) //only the pixels of input image are sent for processing
   begin
       @(posedge clk);
       r = $fscanf(file,"%c",imgData);
       imgDataValid <= 1'b1;
   end
   sentSize = 4*512; //4 line buffers are used
   @(posedge clk);
   imgDataValid <= 1'b0;
   while(sentSize < `imageSize)
```

```
    begin
        @(posedge intr); //after sending 4 pixel rows, we wait for interrupt signal to indicate any of the line buffer is free
        for(i=0;i<512;i=i+1)
        begin
            @(posedge clk);
            r = $fscanf(file,"%c",imgData);
            imgDataValid <= 1'b1;
        end
        @(posedge clk);
        imgDataValid <= 1'b0; //after sending 1 pixel row to 1 line buffer
        sentSize = sentSize+512;
    end
    @(posedge clk); //two dummy lines are sent in the end for neighborhood processing
    imgDataValid <= 1'b0;
    @(posedge intr);
    for(i=0;i<512;i=i+1)
    begin
        @(posedge clk);
        imgData <= 0; //first dummy line data
        imgDataValid <= 1'b1;
    end
    @(posedge clk);
    imgDataValid <= 1'b0;
    @(posedge intr);
    for(i=0;i<512;i=i+1)
    begin
        @(posedge clk);
        imgData <= 0; //second dummy line data
        imgDataValid <= 1'b1;
    end
    @(posedge clk);
    imgDataValid <= 1'b0;
    $fclose(file); //input file closed
end

always @(posedge clk) //this block determines the piels sent (equal to image size)
begin
    if(outDataValid)
    begin
        $fwrite(file1,"%c",outData);
        receivedData = receivedData+1;
    end
    if(receivedData == `imageSize) //data is finished processing and is saved in output image
    begin
       $fclose(file1); //output file closed
       $stop;
    end
end

imageProcessTop dut(
    .axi_clk(clk),
    .axi_reset_n(reset),
    //slave interface
    .i_data_valid(imgDataValid),
    .i_data(imgData),
    .o_data_ready(),
    //master interface
    .o_data_valid(outDataValid),
    .o_data(outData),
    .i_data_ready(1'b1), //IP is always ready to accept data
    //interrupt
    .o_intr(intr)
);

endmodule
```

The image header in bmp image contains general data regarding the image like information about the type, size, and layout of a device-independent bitmap file. It is not altered and is copied to the output image as it is.

b. Waveform

After running the behavioral simulation, press F3 or Run All option which runs the simulation until there are no more $stop or $finish command in the code. The simulation waveform is too large to attach here. So it is shown at a particular instant only.



Active low reset is used. The files are not open for reading and writing the pixel data until the reset goes high and positive edge of the clock arrives. As the pixels are incremented (integer i) the imgDataValid signal goes high which indicates that valid data is being read from the input file. The image header is at is copied to the output file so the outData is 00 because this pixel data is not sent to image processing IP. After that the pixel data is processed. Pixels are sent from the input file into the line buffers. From there they are sent to the image processing IP and the processed pixel data is stored in the output buffer which minimizes any sort of delays in receiving the pixel data. The interrupt signal goes high when data from one line buffer is read from the image processing IP and is processed. The interrupt signal goes high which signals that there is no valid data to be read now and the line buffer is free. So the next pixel row should be streamed to it.

7. Output Images

a. Input Image (Clown.bmp)

b. Box Blur (blurred_clown.bmp)

c. Edge Detection (edgeDetection_clown.bmp)

d. Sharpening (sharpen_clown.bmp)

e. High Pass Filter (brightness_ clown.bmp)

f. Low Pass Filtering (lowpass_ clown.bmp)

8. References

- *https://www.youtube.com/watch?v=TcjqZG2pbHw&feature=youtu.be&ab_channel=VipinKizheppatt*
- *https://en.wikipedia.org/wiki/Sobel_operator*
- *https://en.wikipedia.org/wiki/Kernel_(image_processing)*
- *https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf*