Hardware Accelerator for Image Convolution

Prepared by:
Muhammad Tariq Waseem
BE Electronics
NED University of Engineering & Technology

Submitted to:
IC Design Summer School
Dr. Majida Kazmi
NED University of Engineering & Technology

Table of Content

1. Introduction

Project Objective

The objective of this project is to design and implement a hardware-based image convolution system using System Verilog. Image convolution is a core operation in image processing, utilized for tasks such as edge detection, sharpening, and blurring. This project focuses on creating a modular and parameterized system capable of handling various image sizes (32x32, 128x128, 1024x768, etc.) and kernel configurations (3x3 to 10x10 or more). Comprehensive testing ensures the system's functionality, starting with 32x32 images and scaling up to larger resolutions with kernels like Gaussian blur and sharpening.

Scope of Work

The project involves designing a modular system comprising a Line Buffer, RAM, and Convolution Process module, integrated into a Top Module for data management. The parameterized design supports image sizes up to 1024x768 and beyond, as well as flexible kernel dimensions. Testing includes module-level validation and system-level evaluation with diverse image data and convolution kernels.

The design is synthesized on Quartus, achieving an operational frequency of 200 MHz with standard memory and up to 385 MHz with optimized memory IPs. This scalable and efficient system is suitable for real-time image processing tasks in fields like robotics, computer vision, and multimedia.

2. System Overview

System Architecture

Here is the top-level architecture of the hardware-based image convolution system, showcasing the key components and their interactions.

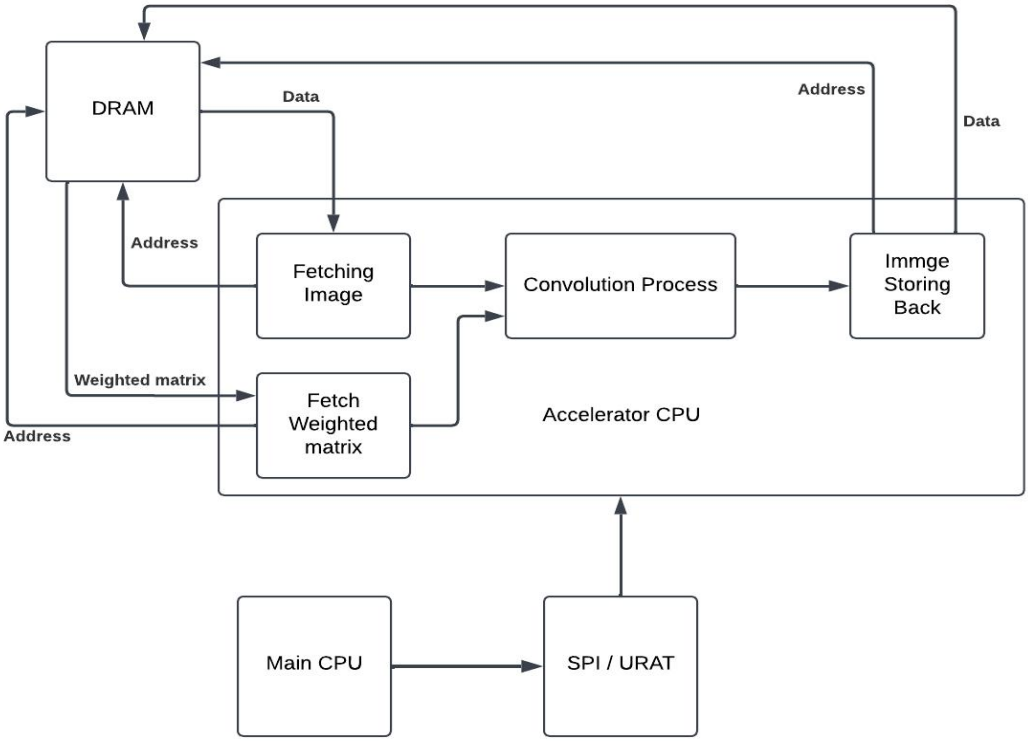

Figure 1 : Architecture of Convolution Accelerator

Figure 1 illustrates the top-level hardware architecture of the image convolution system, which integrates several key components. Here's a detailed explanation of the system:

Functionality Description

The system incorporates a DRAM, generated using the IP catalog, currently with a storage capacity of 65,385 entries. This is a dual-port RAM module, enabling simultaneous data access for different operations. One port is used to fetch the image data, while the other handles the weighted matrix (kernel) and stores the processed (convoluted) image back into memory.

The **line buffer module** manages the fetching of both the image data and the kernel (weighted matrix) from the DRAM. It also handles the storage of the final convoluted image back into the memory. The DRAM is instantiated within this

module, ensuring efficient data handling and communication between the memory and other components.
The **convolution process module** performs the core operation of convolution. This involves multiplying the corresponding pixels of the image with the kernel values and summing them up to compute the output pixel. This module is designed for efficient computation, enabling high-speed processing of image data.

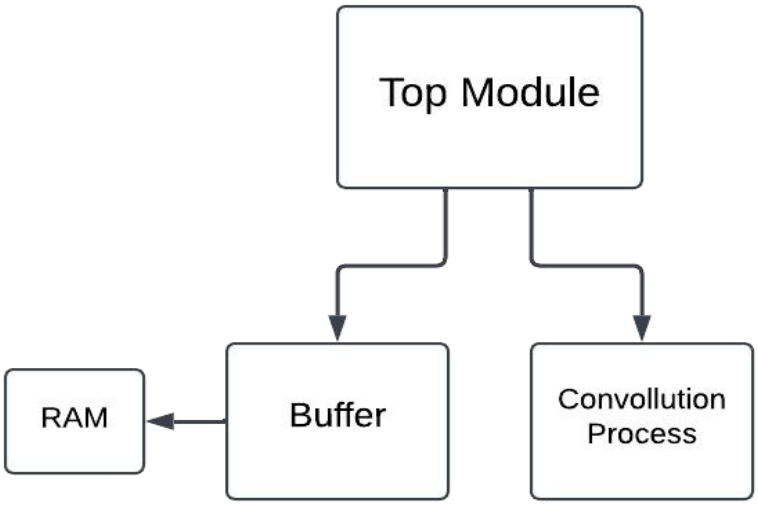

Figure 2 : Modular Design Hierarchy

The **top module** integrates these sub-modules, orchestrating the overall data flow and operation of the system. It ensures seamless communication between the line buffer and convolution modules, managing the entire convolution work flow.

Although the block diagram includes a UART/SPI interface for communication with an external CPU, this part is currently not implemented, as it falls outside the scope of the project. For the current design, the system takes input directly, performs convolution, and generates a “done” signal upon completion.

This design provides a robust and efficient framework for hardware-based image convolution, ensuring modularity and scalability. The DRAM-based architecture allows handling of large images, while the modular approach simplifies testing, debugging, and potential future extensions.

3. Module Design

RAM Module

The RAM in this design plays a crucial role as the primary storage block for image data and convolution results. It is used to store the input image and the convoluted output, ensuring efficient access during processing. Since the color depth in this design is 8-bit, a dual-port RAM is implemented, allowing simultaneous operations for fetching the image data and storing the convoluted result. This dual-port structure ensures faster processing by enabling parallel access to the data without conflicts.

The RAM is byte-wide, making it well-suited for 8-bit color depth, but the design is highly scalable. For applications requiring higher color depths, such as 24-bit, the memory width can be increased to 24 bits, or the number of ports can be expanded to maintain efficient data flow without increasing the memory width. This scalability makes the design adaptable for a wide range of image processing applications, regardless of the required color depth.

Line Buffer

The line buffer is a critical component in the design, providing a significant advantage in reducing data fetching time. In image convolution, the same data often needs to be accessed multiple times for overlapping kernel operations. By temporarily storing this data in the buffer, the need for repeated memory access is minimized, leading to improved efficiency and reduced processing delays.

This line buffer is parameterized, with its size determined by the kernel's number of rows and the image's width. If the kernel Matrix is **3x5** and the image is **1024x768** the the buffer needed is of **4 row**s and **768 columns**. For larger image it will consume a large area. This large area can be divided to **3x394** or **3x197** but it’s a trade off between latency and area.

To explain the fetching mechanism in the line buffer, let’s consider a 3x3 kernel and an image with a width of *n* bits. The line buffer ensures efficient access to data for convolution by minimizing memory access time while maintaining flexibility for larger image sizes.

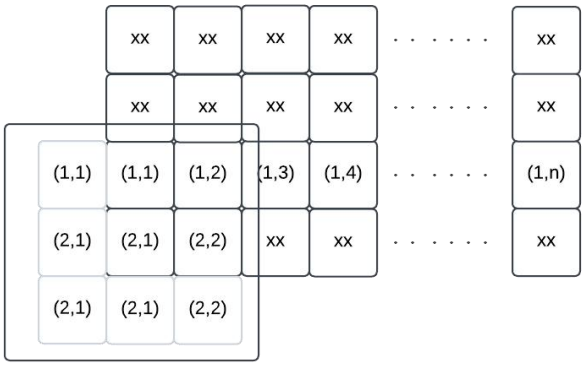

Figure 3 : Fetching Data from the Buffer for First Pixel

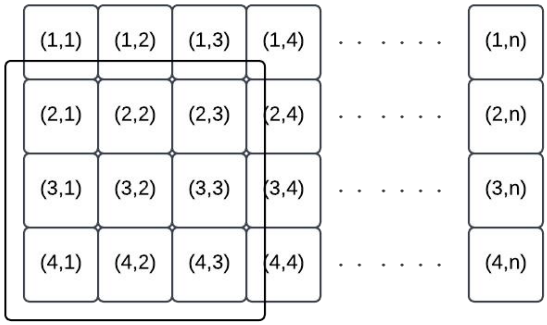

Figure 4 : Fetching Data row 3 column 2

First Pixel Fetch

For the first pixel of the image, the line buffer fetches the first 9 pixels data required for convolution (3x3 matrix) only 4 pixel will used, for the other 5 element the neighbour data will replicated. The data is fetched row-wise from the buffer (e.g., rows 1, 2, and 3), ensuring that all pixels needed for the convolution process are available.

Middle Pixels Fetch:

For all pixels in the middle of the image except the second last one of each row, the line buffer ensures that the required 9 pixels are already available in the buffer. As the convolution window slides across the image, data is fetched directly from rows 2, 3, and 4 of the buffer. This mechanism avoids redundant data fetching from the memory, saving significant time.

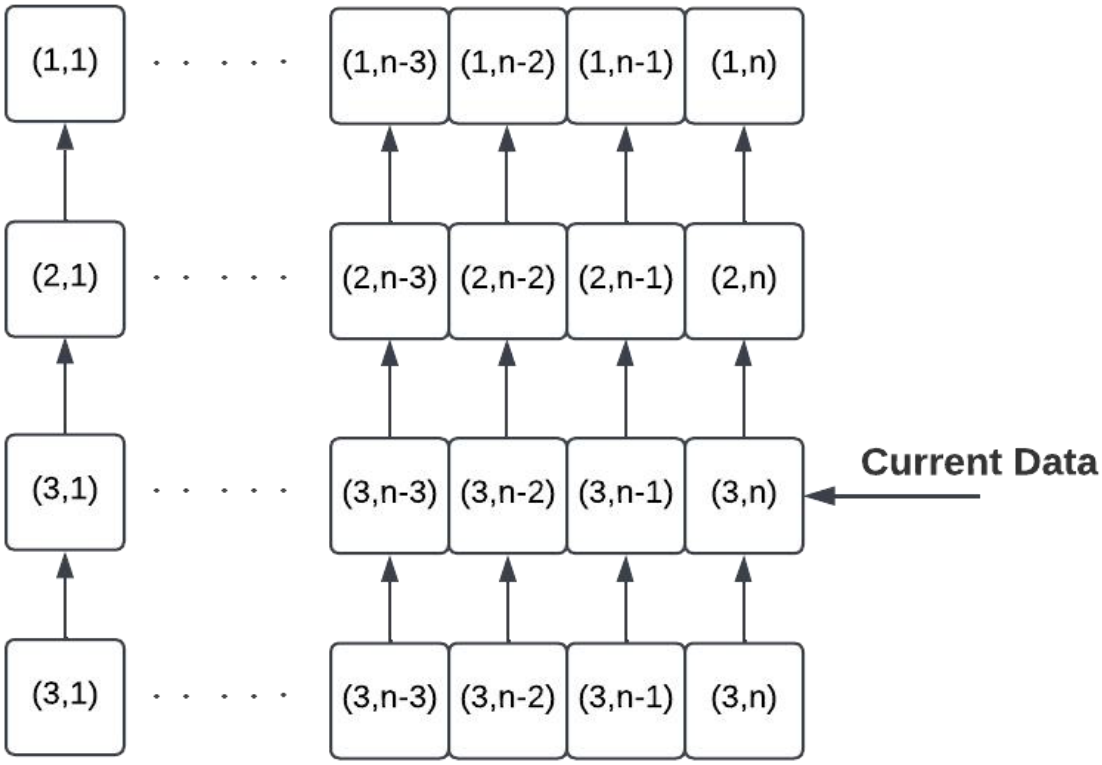

Figure 5 : Row Shifting

Row Shifting for Efficiency:

The critical fetching mechanism occurs after every *n*-bit row of data is processed and stored in the buffer. At this point:

1. Data in row 4 is shifted to row 3.
2. Data in row 3 is shifted to row 2.
3. Data in row 2 is shifted to row 1.
4. Data in row 1 is discarded.

This row-shifting mechanism allows the buffer to update with new row data while retaining the previously required data, which is essential for overlapping kernel operations. For the second last pixel of every row, data is fetched from rows 1, 2, and 3 of the buffer, ensuring continuity in the convolution process.

Fetching Second Last and Last Pixels

During this process the convolution process is 2 cycle late as the last row is fetched but the convolution pixel is in **n-2** row this will provide a problem for the the convolution of every rows 2nd last and last pixel for this we have two choices again one is to hold our fetching for 2 cycle to do this and then hold the shift then hold the convolution for 2 cycles to fetch the data in for next pixel in this way in every row there will be 4 wasted cycle cycle which will

created a lot if we think for larger image so we go with an extra to fetch retain the data for the 2nd last and the last row as shown in Figure 5 and Figure 6.

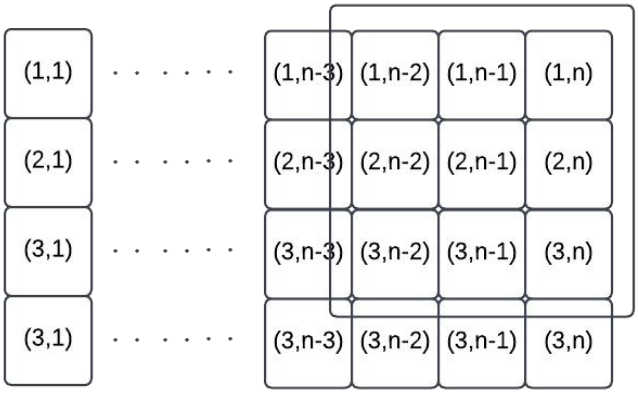

Figure 6 : Fetching 2nd last pixel of 2nd Row

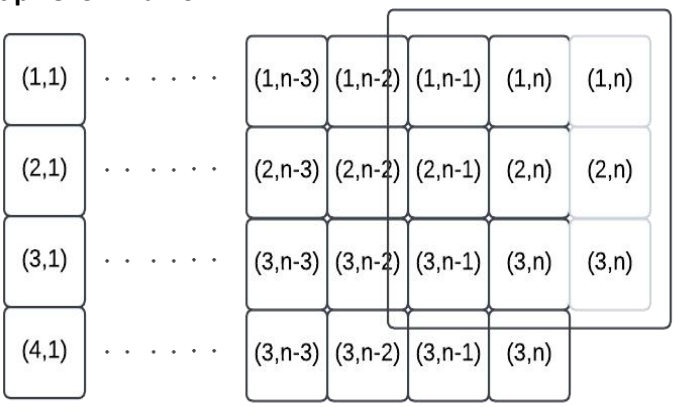

Figure 7 : Fetching last pixel of 2nd Row

The Figures 5 and 6 illustrate the fetching mechanism for the second last and last pixels of a row. These figures demonstrate how the data required for these pixels is efficiently retrieved from the line buffer. Similarly, for every subsequent row, the fetching process for the second last and last pixels follows the same pattern, ensuring seamless data availability for convolution without additional clock cycle delays. This systematic approach ensures consistent performance and efficiency throughout the image processing pipeline.

Line Buffer Fetch the data matrix and the kernel give them at its output from there the top module transfer it to the Convolution Process for Convolution.

Convolution Process

The Convolution Process block takes the data matrix from the Line Buffer and the kernel matrix as its inputs. Within this block, each element of the data matrix is multiplied by its corresponding element in the kernel matrix. The resulting products from these multiplications are then summed together to produce the final convolution result.

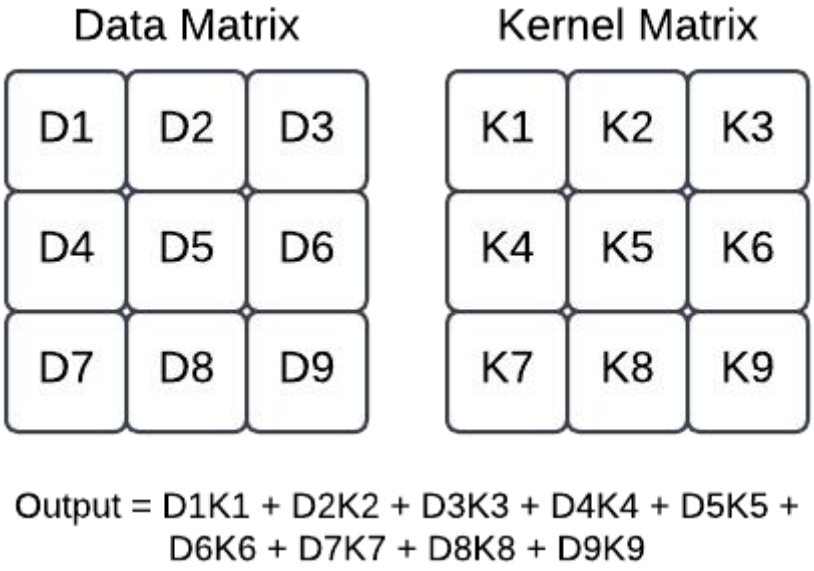

Figure 8 : Convolution Process

Since the input is based on an 8-bit color depth, the output of the summation is a 16-bit value, which must be converted back to 8 bits. To convert this result back to a 8 bit there are different method with some advantages and disadvantages which are as follow:

Truncation Method

Truncation simply discards the least significant 8 bits of the 16-bit number, keeping the 8 most significant bits. This method is computationally efficient but may result in a loss of precision.[1]

This method is simple in computation but this will lose the data so the image will be not that accurate.

Gamma Correction

Applies a gamma correction function to the 16-bit value before reducing it to 8 bits. The gamma function is

Vout = 255 . (Vin/65535)^y

where y(Gamma) is typically between 0.8 and 2.2. This method is used to adjust for human perception of brightness.[2]

This Method is good for the precision of brightness but not good for convolution and it need more computation.

Bit Retention and Replacement in Convolution Output

In this method, the first two most significant bits (MSBs) of the original 8-bit pixel colour are retained to preserve the higher-value significance of the data. The remaining six bits are replaced with LSB of the result of the convolution operation, such as normalization or scaling. This approach ensures that the convolution output fits within the 8-bit color depth while maintaining a balance between precision and compatibility. By carefully selecting and modifying the bits, the method effectively reduces data loss and adapts the output for further image processing tasks.

Top Module

The **Top Module** serves as the main control unit for the hardware-based image convolution system. It integrates two key modules: the **Line Buffer** and the **Convolution Process** module. The Line Buffer provides the Data Matrix and Kernel Matrix, which serve as the inputs to the Convolution Process module. The Top Module ensures the seamless operation of these components by managing their interactions and coordinating their processes.

To control the Line Buffer and the Convolution Process, a **Finite State Machine (FSM)** is implemented within the Top Module. The FSM has five states that manage the data flow and operational timing:

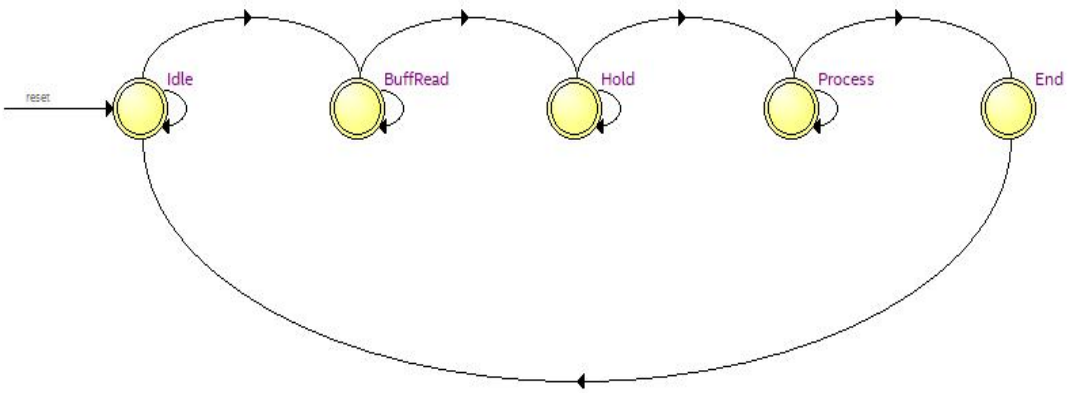

Figure 9 : State Diagram

1. **Idle**:
   The system waits in this initial state until a start signal is received. This state ensures the system is inactive until explicitly commanded to begin.
2. **Buff Read**:
   During this state, the Line Buffer reads the kernel matrix from memory and in the mean time it start fetching the image data. The buffer prepares the data for subsequent processing.
3. **Hold**:
   This state temporarily halts the operation to align the buffer data or ensure synchronization between the Line Buffer and the Convolution Process module. It adds a certain delay so that there is enough data present to start the convolution.
4. **Process**:
   The convolution operation is performed in this state. The Convolution Process module processes the data provided by the Line Buffer and produces the convoluted image and stored back in the Memory.
5. **End**:
   This final state signals(Done) the completion of the convolution process. The system reset to the Idle state or await further instructions.

The FSM cycles through these states to ensure efficient and coordinated execution of the image convolution pipeline. This modular and state-driven approach enables a flexible and scalable design, supporting varying image sizes and kernel configurations.

4. Parameterization

Scaling to Larger Images

The design was extended to handle larger images, such as 128x128 and 256x256 or 1024x768 by parameterizing key components. The line buffer and memory modules were scaled using parameters for image width and height, enabling dynamic adjustment based on input image size. This approach allows the system to support various resolutions without altering the core structure. The scalable design ensures efficient handling of larger data sets while maintaining modularity and ease of implementation.

Kernel Flexibility

The convolution process was designed to support multiple kernels, such as sharpen, Gaussian blur, and edge detection, by parameterizing the kernel matrix. Each kernel type is defined as a programmable parameter, enabling the system to dynamically switch between different convolution operations. This flexibility allows for a wide range of image processing applications without requiring hardware modifications.

5. Testing and Results

Buffer Module Testing

The Buffer Module testing was conducted using a comprehensive self-checking testbench designed to validate the functionality of the line buffer implementation. The primary purpose of this buffer is to organize image data into a matrix format suitable for convolution operations. The testbench systematically processes input image data and generates corresponding kernel matrices for convolution. To verify the accuracy of the buffer module, test vectors containing sample image data were fed through the testbench, and the resulting output matrices were compared against expected results. The simulation demonstrated that the buffer module successfully maintains data integrity while transforming the input stream into the required matrix format. This testing approach ensures reliable data handling and proper synchronization between the image data output and convolution kernel generation, which is crucial for subsequent image processing operations.

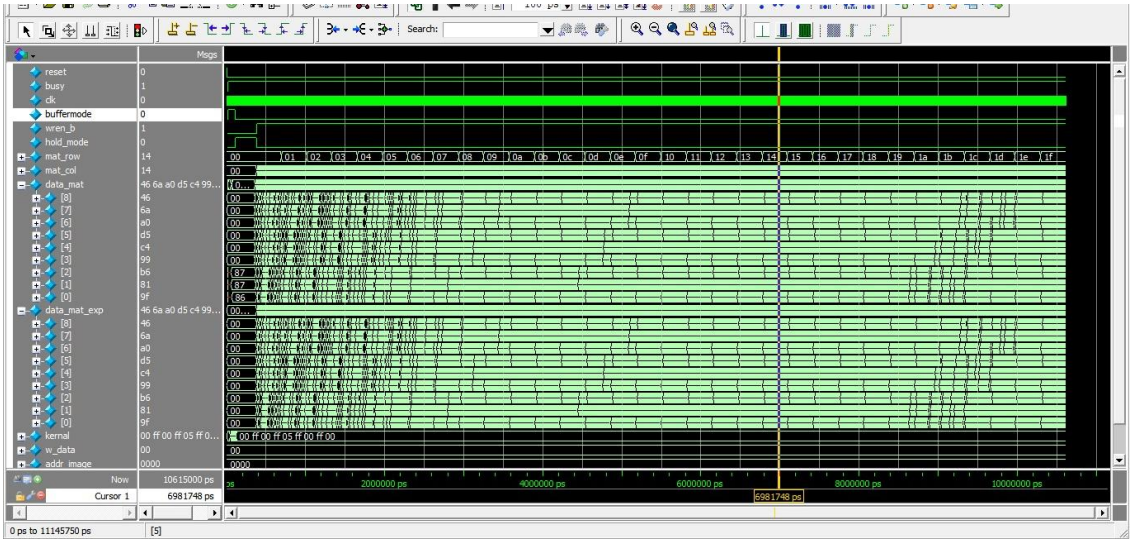

Figure 10 : Simulation of Buffer (32x32 image)

The Figure 10 is the simulation of the Line Buffer module for a 32x32 image the buffer is also tested with a 128x128 image data that was shown in Figure 12. The Expected output is matched with the real output at instance of every pixel hence the module is perfectly designed.

```
# Errors: 0, Warnings: 0
# End time: 20:28:23 on Jan 12,2025, Elapsed time: 0:04:18
# Errors: 0, Warnings: 0
# vsim -novopt tb_buffer
# Start time: 20:28:23 on Jan 12,2025
# Loading sv_std.std
# Loading work.tb_buffer
# Loading work.buffer
# Loading work.ram
# Loading work.altsyncram
# Loading work.altsyncram_body
# Loading work.ALTERA_DEVICE_FAMILIES
# Loading work.ALTERA_MF_MEMORY_INITIALIZATION
# matched =        1023
# miss_matched =           0
# ** Note: $stop    : tb_buffer.sv(396)
#    Time: 10615 ns  Iteration: 1  Instance: /tb_buffer
# Break in Module tb_buffer at tb_buffer.sv line 396

VSIM 61>
```

Figure 11 : Self Checking Results

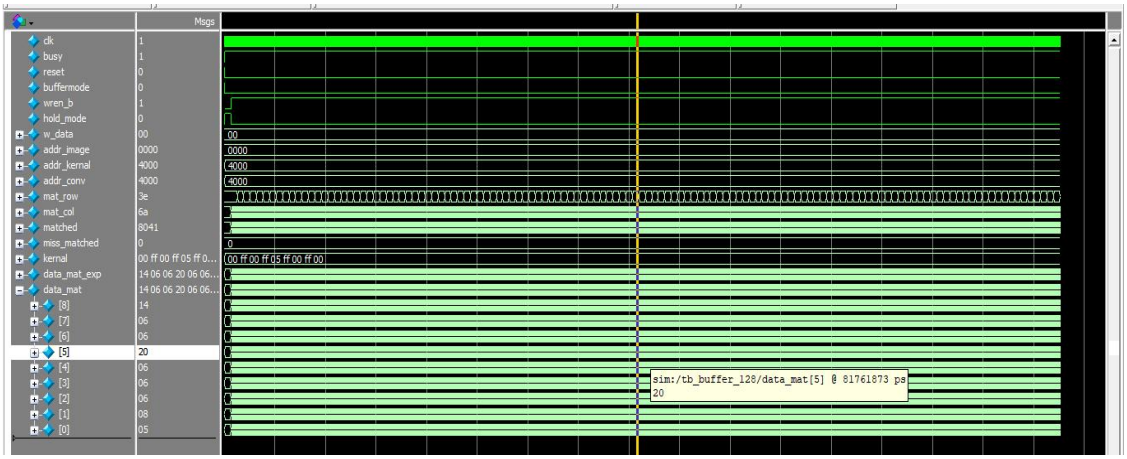

Figure 12 : Line Buffer Simulation with 128x128 image

```
# Loading work.buffer
# Loading work.ram
# Loading work.altsyncram
# Loading work.altsyncram_body
# Loading work.ALTERA_DEVICE_FAMILIES
# Loading work.ALTERA_MF_MEMORY_INITIALIZATION
# matched =       16384
# miss_matched =           0
# ** Note: $stop    : tb_buffer_128.sv(281)
#    Time: 165185 ns  Iteration: 1  Instance: /tb_buffer_128
# Break in Module tb_buffer_128 at tb_buffer_128.sv line 281

VSIM 3>
```

Figure 13 : Self Checking Result of Simulation

Convolution Process Testing

This module is tested using a self checking test bench as well while generating random numbers.

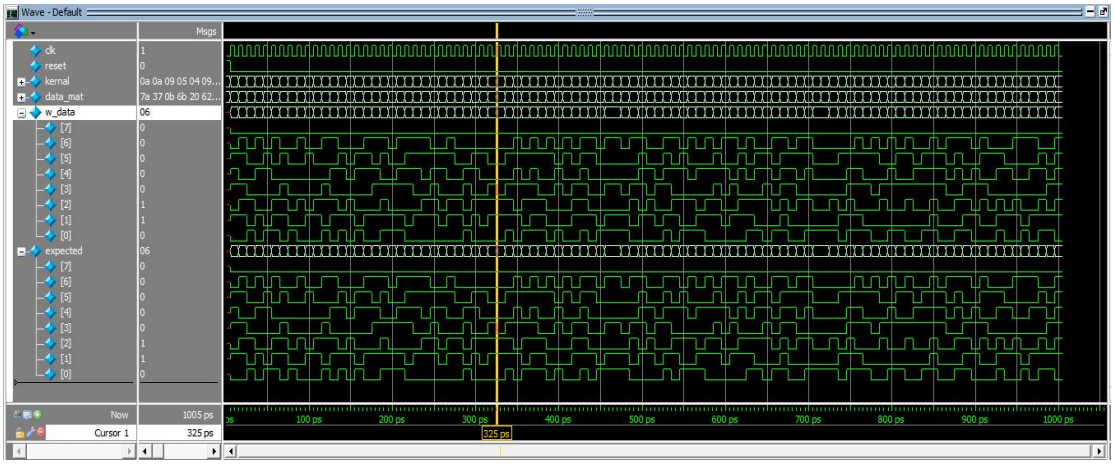

Figure 14 : Simulation of Convolution Process

```
# End time: 00:26:25 on Jan 15,2025, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# End time: 00:26:31 on Jan 15,2025, Elapsed time: 0:02:01
# Errors: 0, Warnings: 0
# vsim -novopt tb_conv_proc
# Start time: 00:26:32 on Jan 15,2025
# Loading sv_std.std
# Loading work.tb_conv_proc
# Loading work.conv_proc
# mismatched =           0
# matched =            100
# ** Note: $stop    : tb_conv_proc.sv(51)
#    Time: 1005 ps  Iteration: 0  Instance: /tb_conv_proc
# Break in Module tb_conv_proc at tb_conv_proc.sv line 51

VSIM 6>
```

Figure 15 : Self Checking Result

Top Module Testing

The Top module is tested by plotting the convoluted data in to image and comparing it with the real image while using different kernel matrix. The data has been converted into an image using a Matlab for different colour mapping style has been used the gray scale, the hot color, the jet colour and the parula colour.

The Real Image of 32x32 pixel :

32x32 Image

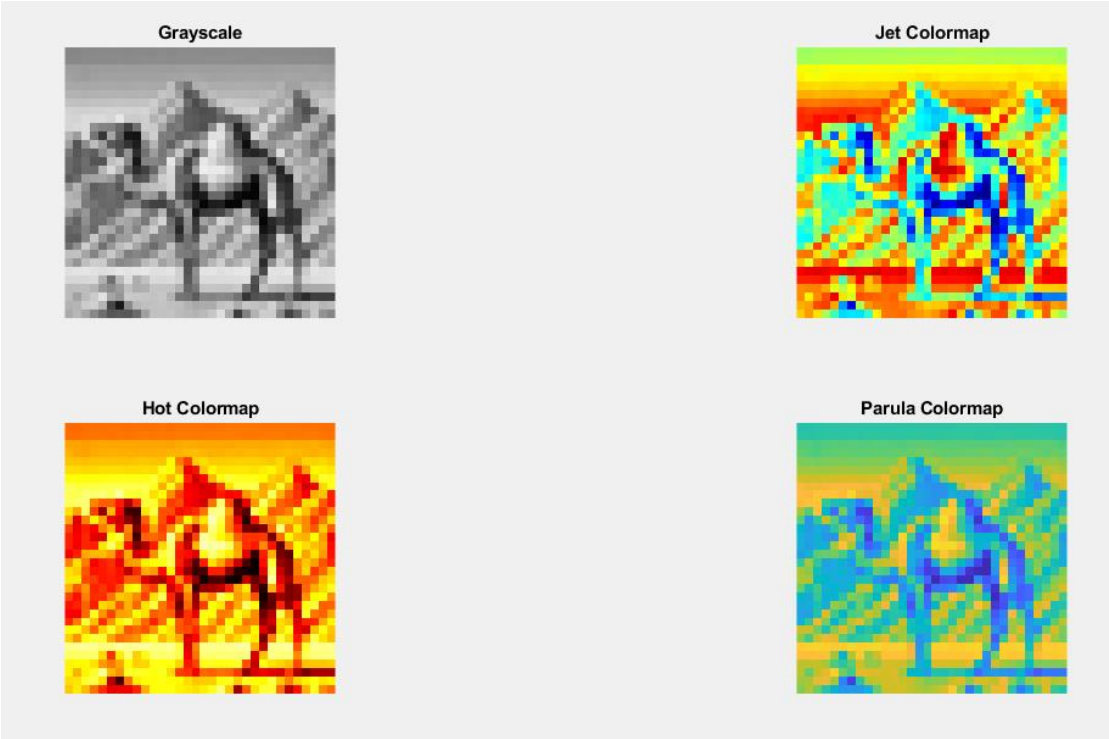

Figure 16 : Real Image of 32x32 pixel

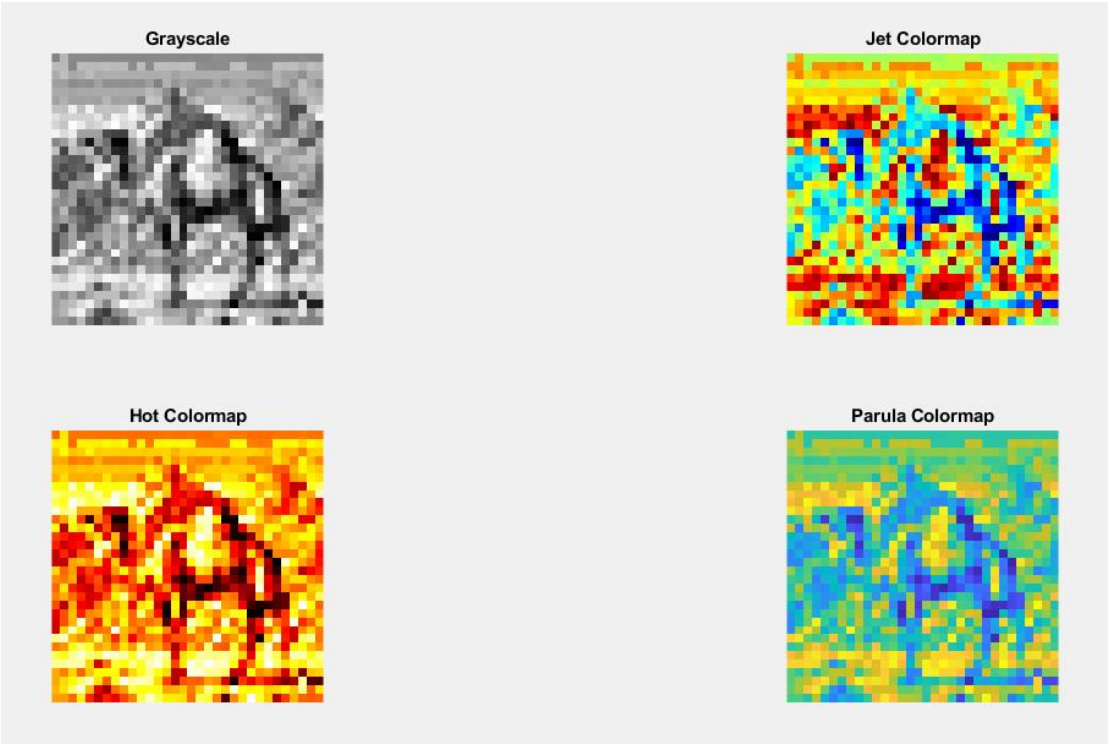

Figure 17 : Convoluted image of 32x32 with Sharpen Kernel

The image is too blur for comparison as it's a 32x 32 pixel so lets have a good view of 128x128 pixel of same image

128x128 Image

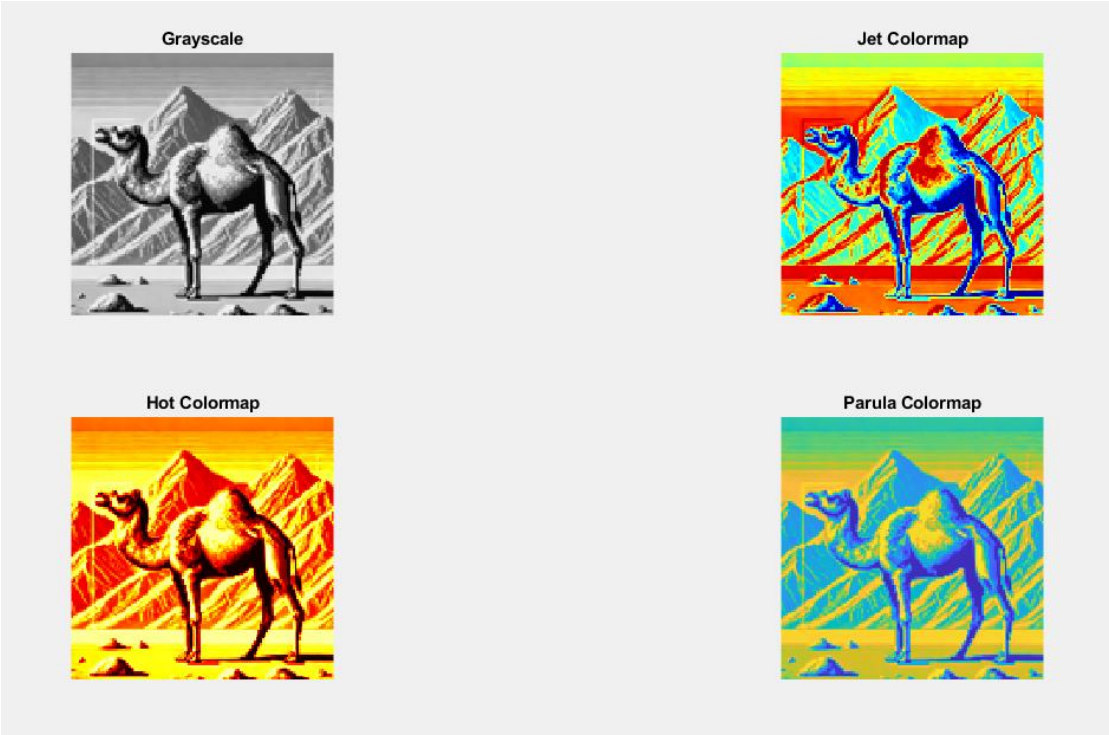

Figure 18 : Real Image of 128x128 Pixel

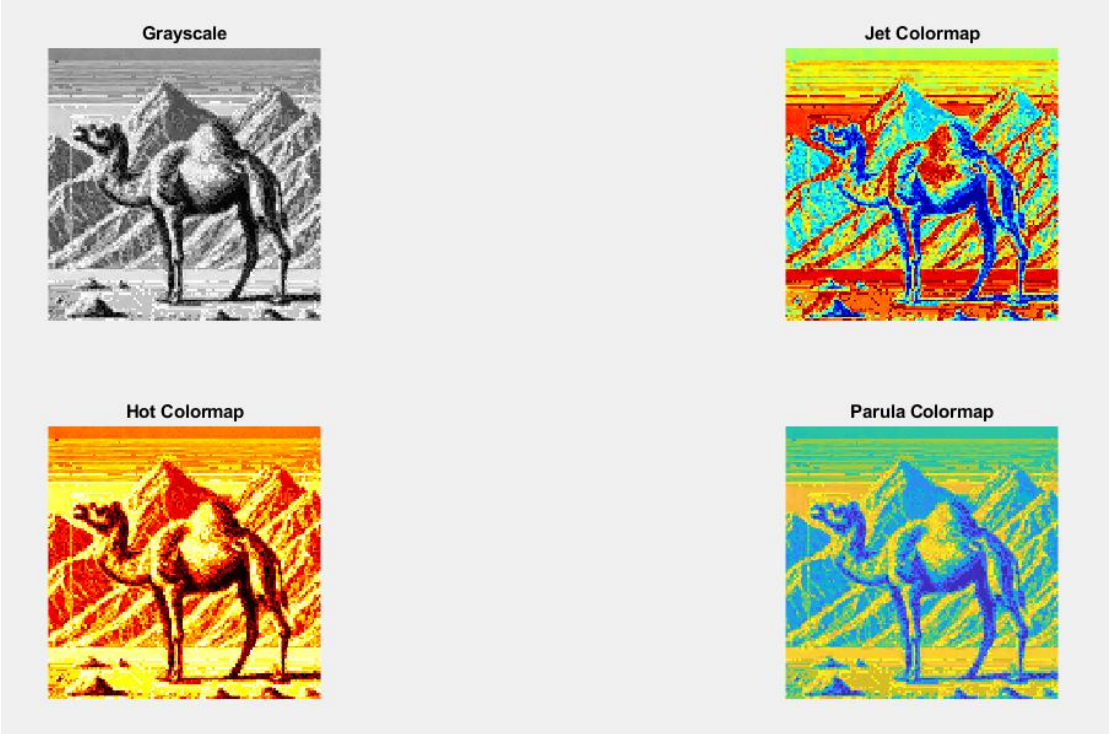

Figure 19 : Convoluted Image of 128x128 with Sharpen kernel

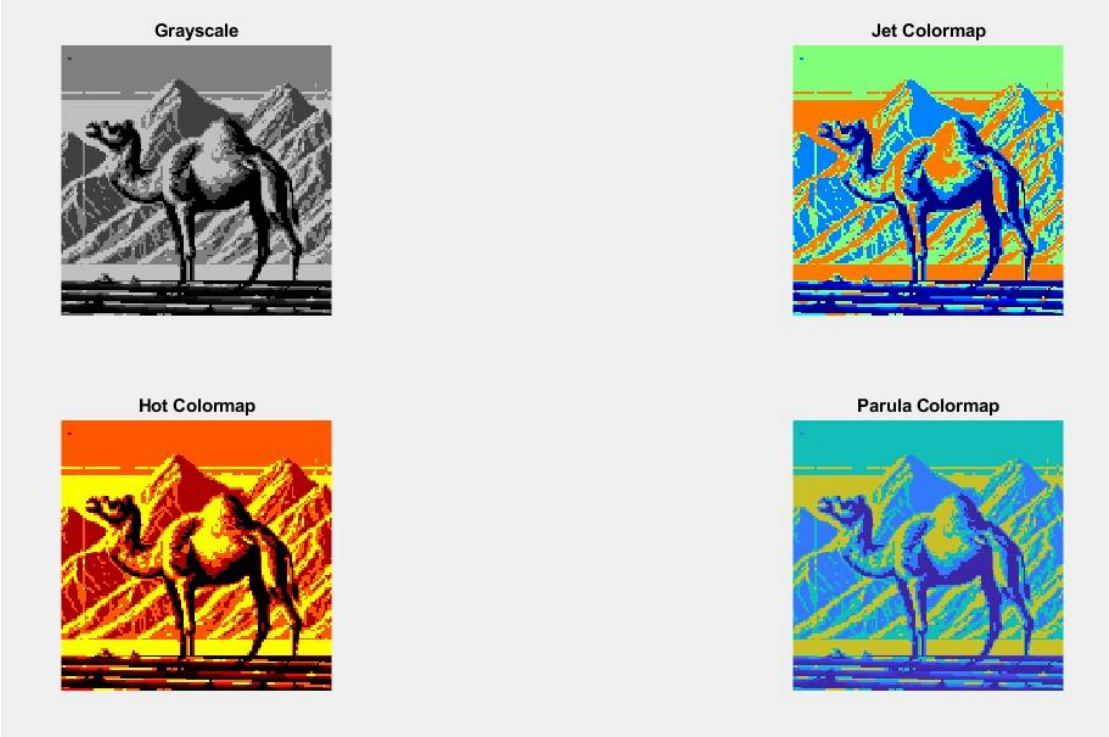

Figure 20 : Convoluted Image of 128x128 with Gaussian blur Kernel

Comparative analysis of different image resolutions revealed significant variations in the effectiveness of convolution operations. The 32x32 pixel image demonstrated limited visual differentiation when processed with the sharpen kernel, producing results that were difficult to distinguish from the original image due to the low resolution. In contrast, the 128x128 pixel image exhibited markedly superior results, showing clear and discernible improvements when processed with both the sharpen and Gaussian blur kernels. This higher resolution provided sufficient pixel density to effectively demonstrate the impact of the convolution operations, allowing for better visualization of edge enhancement with the sharpen kernel and smooth transitions with the Gaussian blur filter. The enhanced performance at 128x128 resolution validates the effectiveness of the implemented convolution algorithms while highlighting the importance of adequate image resolution for optimal filter performance.

6. Performance Analysis

The design achieves an overall operational frequency of 200 MHz, determined by the line buffer's speed. The top module itself is capable of running at a maximum frequency of 385 MHz; however, the slower line buffer becomes the bottleneck in this implementation. This performance is sufficient for most real-time image processing applications but could be significantly improved by integrating a faster RAM IP. Optimizing the memory subsystem would allow the design to fully utilize the top module's potential, thereby enhancing the processing throughput for larger images and more complex convolution operations.

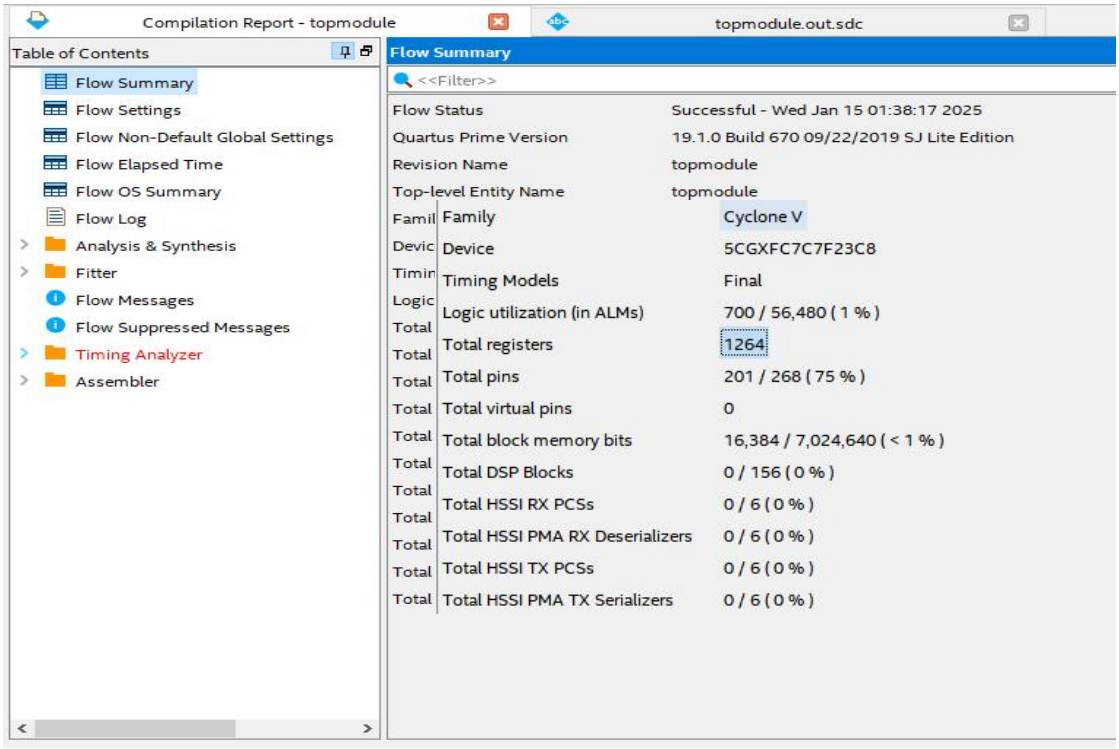

Figure 21 : Synthesis Report

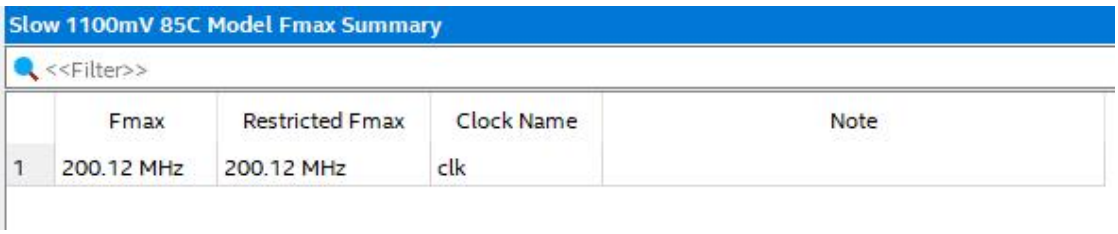

Figure 22 : Maximum Frequency of Top Module

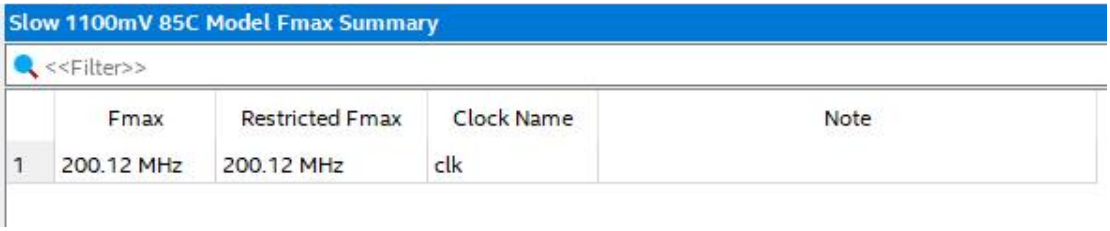

Figure 23 : Maximum Frequency of Line Buffer

7. Conclusion

In this project, a scalable and efficient convolution processing system was designed and implemented, demonstrating its capability to handle real-time image processing tasks. The modular architecture, consisting of a line buffer and convolution process, provides flexibility for various image sizes and kernel configurations. Parameterization allows the system to scale seamlessly to larger image dimensions, while the inclusion of an FSM ensures precise control and synchronization of operations. The performance analysis highlights the system's potential, with an operational speed of 200 MHz on DE1 Soc, which could be further improved with faster memory modules. Overall, the design provides a robust and adaptable solution for high-speed image convolution, paving the way for further advancements in real-time digital image processing applications.

8. References

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2008.
2. C. A. Poynton, *A Technical Introduction to Digital Video*. New York, NY, USA: John Wiley & Sons, 1996.

Appendices

Top Module

```
module topmodule #(
        parameter IMAGE_WIDTH   = 128,
        parameter IMAGE_HEIGHT  = 128,
        parameter KERNAL_WIDTH  = 3  ,
        parameter KERNAL_HEIGHT = 3  ,
        parameter COLOUR_DEPTH  = 8
) (
        input  logic       clk      ,
        input  logic       reset    ,
        input  logic       start    ,
        input  logic [14:0] addr_image ,
        input  logic [14:0] addr_kernel,
        input  logic [14:0] addr_conv  ,
        output logic [15:0] offset    ,
        output logic       busy     ,
        output logic       done
);

localparam ADDR_W = $clog2(IMAGE_HEIGHT),
       ADDR_H = $clog2(IMAGE_WIDTH );
localparam Idle    = 3'd0,
                BuffRead = 3'd1,
                Hold    = 4'd2,
                Process  = 3'd3,
                End     = 3'd4;
logic [          2:0] D                              ;
logic [          2:0] Q                              ;
logic [         10:0] counter                           ;
logic                  buffermode                      ;
logic                  hold_mode                       ;
logic                  wren_b                         ;
logic [COLOUR_DEPTH-1:0] w_data                              ;
logic [     ADDR_H-1:0] mat_row                         ;
logic [     ADDR_W-1:0] mat_col                         ;
logic [COLOUR_DEPTH-1:0] data_mat  [KERNAL_WIDTH*KERNAL_HEIGHT-1:0];
logic [COLOUR_DEPTH-1:0] kernal    [KERNAL_WIDTH*KERNAL_HEIGHT-1:0];

//*****************************************************************************
//********** COUNTER FOR THE CLOCK COUNT FOR FSM ***************************
//*****************************************************************************

always_ff @(posedge clk or posedge reset) begin
  if(reset)     counter <= 0        ;
  else if (busy) counter <= counter + 1;
end
always_ff @(posedge clk) begin
        if(wren_b) begin
                offset <= offset + 1;
                if (mat_col == ((1 << ADDR_W)-1)) begin
                        mat_col <= 0;
                        mat_row <= mat_row + 1;
                end
                else begin
                  mat_col <= mat_col + 1;
                end
        end
        else begin
                mat_col <= 0;
                mat_row <= 0;
                offset  <= 0;
        end
end
//*****************************************************************************
//************FSM to control the whole Process*******************************
//*****************************************************************************

//***********Combinational logic for next states*****************************
//*****************************************************************************
always_comb begin
        case(Q)
                Idle    : if(start)                     D = BuffRead;
                        else                          D = Idle    ;
```

```
                    BuffRead : if(counter == (KERNAL_WIDTH*KERNAL_HEIGHT-1))  D = Hold    ;
                            else                                    D = BuffRead;
                    Hold     : if(counter == IMAGE_WIDTH + 4)               D = Process ;
                            else                                    D = Hold    ;
                    Process  : if(offset == (1 << (ADDR_W+ADDR_H)) + 1)      D = End     ;
                            else                                    D = Process ;
                    End      :                                       D = Idle    ;
            endcase
end

//*********************FF of State Diagram***********************************
//*****************************************************************************

always@(posedge clk or posedge reset) begin
            if(reset) Q <= 3'd0;
            else      Q <= D   ;
end

//*************ombinational logic for output*******************************
//*****************************************************************************

always_comb begin
            case(Q)
                    Idle     : {done,wren_b,hold_mode,buffermode,busy} <= 6'b00000;
                    BuffRead : {done,wren_b,hold_mode,buffermode,busy} <= 6'b00011;
                    Hold     : {done,wren_b,hold_mode,buffermode,busy} <= 6'b00101;
                    Process  : {done,wren_b,hold_mode,buffermode,busy} <= 6'b01001;
                    End      : {done,wren_b,hold_mode,buffermode,busy} <= 6'b10001;
            endcase
end

//*****************************************************************************
//********************** INSTANTIATION OF BUFFER ****************************
//*****************************************************************************

buffer #(
            .WIDTH      (IMAGE_WIDTH  ),
            .HEIGHT     (IMAGE_HEIGHT ),
            .K_WIDTH    (KERNAL_WIDTH ),
            .K_HEIGHT   (KERNAL_HEIGHT),
            .ADDR_W     (ADDR_W       ),
            .ADDR_H     (ADDR_H       ),
            .COLOUR_DEPTH(COLOUR_DEPTH ),
            .MEMORY_ADDR (15          )
) buffer (
            .clk       (clk       ),
            .busy      (busy      ),
            .reset     (reset     ),
            .buffermode (buffermode ),
            .wren_b    (wren_b    ),
            .hold_mode  (hold_mode  ),
            .w_data    (w_data    ),
            .addr_image (addr_image ),
            .addr_kernal(addr_kernal),
            .addr_conv  (addr_conv  ),
            .mat_row   (mat_row   ),
            .mat_col   (mat_col   ),
            .data_mat   (data_mat   ),
            .kernal    (kernal    )
);

//*****************************************************************************
//********************** INSTANTIATION OF PROCESS ***************************
//*****************************************************************************

conv_proc #(
            .KERNAL_WIDTH (KERNAL_WIDTH ),
            .KERNAL_HEIGHT(KERNAL_HEIGHT),
            .COLOUR_DEPTH (COLOUR_DEPTH )
) conv_proc (
            .clk    (clk    ),
            .reset   (reset   ),
            .data_mat(data_mat),
            .kernal  (kernal  ),
            .w_data  (w_data  )
);

Endmodule
```

Test-Bench of Top Module

```
`timescale 1ns/1ps
module tb_topmodule ;

        logic       clk          = 0;
        logic       reset         = 1;
        logic       start         = 0;
        logic [14:0] addr_image     = 0;
        logic [14:0] addr_kernal    = 0;
        logic [14:0] addr_conv      = 0;
        logic [15:0] offset          ;
        logic       busy             ;
        logic       done             ;

        topmodule #(
                .IMAGE_WIDTH  (128),
                .IMAGE_HEIGHT (128),
                .KERNAL_WIDTH (3  ),
                .KERNAL_HEIGHT(3  ),
                .COLOUR_DEPTH (8  )
        ) i_topmodule (
                .clk          (clk          ),
                .reset        (reset        ),
                .start        (start        ),
                .addr_image    (addr_image    ),
                .addr_kernal   (addr_kernal   ),
                .addr_conv     (addr_conv     ),
                .offset       (offset       ),
                .busy         (busy         ),
                .done         (done         )
        );

always #5 clk <= ~clk;

initial begin
        #10 reset      = 0     ;
        #50 start      = 1     ;
           addr_image  = 0     ;
           addr_kernal = 15'h4009;
           addr_conv   = 15'h4000;
        #20 start      = 0     ;
        #165000;
        $stop;
end

endmodule
```

Line Buffer

```
module buffer #(
        parameter WIDTH       = 32,
        parameter HEIGHT      = 32,
        parameter K_WIDTH     = 3 ,
        parameter K_HEIGHT    = 3 ,
        parameter ADDR_W      = 5 ,
        parameter ADDR_H      = 5 ,
        parameter COLOUR_DEPTH = 8 ,
 parameter MEMORY_ADDR  = 11
) (
        input  logic                clk                     ,
        input  logic                busy                    ,
        input  logic                reset                   ,
        input  logic                buffermode              ,
        input  logic                wren_b                  ,
        input  logic                hold_mode               ,
        input  logic [COLOUR_DEPTH-1:0] w_data              ,
        input  logic [ MEMORY_ADDR-1:0] addr_image          ,
        input  logic [ MEMORY_ADDR-1:0] addr_kernal         ,
        input  logic [ MEMORY_ADDR-1:0] addr_conv           ,
        input  logic [     ADDR_H-1:0] mat_row              ,
        input  logic [     ADDR_W-1:0] mat_col              ,
        output logic [COLOUR_DEPTH-1:0] data_mat [K_WIDTH*K_HEIGHT-1:0],
        output logic [COLOUR_DEPTH-1:0] kernal   [K_WIDTH*K_HEIGHT-1:0]
);

logic           buffermode_d  ;
logic           buffermode_2d ;
```

```
logic [COLOUR_DEPTH-1:0] data_port_a   ;
logic [COLOUR_DEPTH-1:0] data_port_b   ;
logic [ MEMORY_ADDR-1:0] address_port_a;
logic [ MEMORY_ADDR-1:0] address_port_b;

//*********************************************************************
//************** BUFFER INITIALIZATION ******************************
//*********************************************************************

logic [COLOUR_DEPTH-1:0] buffer [K_HEIGHT:0][WIDTH-1:0];

logic [   ADDR_H+ADDR_W  :0] offset_image ;
logic [K_HEIGHT*K_WIDTH-1:0] offset_kernal;
logic [   ADDR_H+ADDR_W-1:0] offset_conv  ;

logic [  ADDR_W:0] real_image_col;
logic [ADDR_H-1:0] conv_image_row;
logic [ADDR_W-1:0] conv_image_col;
logic [ K_WIDTH:0] buffer_col    ;

logic [ADDR_H-1:0]f_row;
logic [ADDR_H-1:0]l_row;
logic [ADDR_H-1:0]e_row;
logic [ADDR_W-1:0]f_col;
logic [ADDR_W-1:0]l_col;
logic [ADDR_W-1:0]e_col;
logic [ADDR_H-1:0]uk_row;
logic [ADDR_W-1:0]uk_col;

assign f_row  = 0;
assign f_col  = 0;
assign l_row  = (1 << (ADDR_H)) -1;
assign l_col  = (1 << (ADDR_W)) -1;
assign e_row  = (1 << (ADDR_H)) -2;
assign e_col  = (1 << (ADDR_W)) -2;
assign uk_col = 'b?;
assign uk_row = 'b?;

//*********************************************************************
//****************** DATA FETChING FROM MEMORY ************************
//*********************************************************************

always_ff @(posedge clk or posedge reset) begin
        if(reset) offset_image <= 0;
        else begin
                if(busy) begin
                        if (offset_image == (1 << (ADDR_W+ADDR_H)))
                                offset_image <= offset_image    ;
                        else
                                offset_image <= offset_image + 1;
                end
                else offset_image <= 0;
        end
end

always_ff @(posedge clk or posedge reset) begin
        if(reset)
                address_port_a <= addr_image;
        else if(busy)
          address_port_a <= addr_image + offset_image;
end

//*********************************************************************
//****************** DATA STORING IN BUFFER ***************************
//*********************************************************************

always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
                for (int i = 0; i < K_HEIGHT; i++) begin
                        for (int j = 0; j < WIDTH; j++) begin
                                buffer[i][j] <= 0;
                        end
                end
 end
        else begin
                if (real_image_col == ((1 << ADDR_W) + 1)) begin
                        for (int i = 0; i < K_HEIGHT; i++) begin
                                for (int j = 0; j < WIDTH; j++) begin
                                buffer[i][j] <= buffer[i+1][j];
```

```
                                end
                            end
                            buffer[K_HEIGHT-1][WIDTH-1] <= data_port_a;
                    end
                    else if(offset_image != (1 << (ADDR_W+ADDR_H)) + 1)
                            buffer[K_HEIGHT][real_image_col-2] <= data_port_a;
                    else
                            buffer[K_HEIGHT][real_image_col-2] <= 0;
            end
end

always_ff @(posedge clk or posedge reset) begin
            if (reset) real_image_col <= 0;
            else begin
                    if(real_image_col == ((1 << ADDR_W) + 1))
                            real_image_col <= 2             ;
                    else if ((busy)&(offset_image != (1 << (ADDR_W+ADDR_H)) + 1))
              real_image_col <= real_image_col + 1;
                    else
                            real_image_col <= 0             ;
            end
end

//*********************************************************************
//************ KERNAL DATA FETCHING FROM MEMORY **********************
//*********************************************************************

always_ff @(posedge clk or posedge reset) begin
            if(reset) offset_kernal <= 0;
            else begin
                    if(buffermode | buffermode_d) offset_kernal <= offset_kernal+1;
   else                  offset_kernal <= 0           ;
            end
end

always_ff @(posedge clk or posedge reset) begin
            if (reset) address_port_b <= addr_kernal;
            else begin
                    if (buffermode)
              address_port_b <= addr_kernal + offset_kernal;
             else if (hold_mode)
                    address_port_b <= addr_conv             ;
             else if (wren_b)
                    address_port_b <= addr_conv + offset_conv   ;
             else
                    address_port_b <=  addr_kernal          ;
            end
end

//*********************************************************************
//******** KERNAL DATA STORING IN MATRIX FORM ************************
//*********************************************************************

always_ff @(posedge clk) begin
            buffermode_d <= buffermode;
            buffermode_2d <= buffermode_d;
end

always_comb begin
            if (reset) begin
                    for (int k = 0; k < K_HEIGHT*K_WIDTH; k++) begin
                            kernal[k] <= 0;
                    end
            end
            else if (buffermode_2d)
                    kernal[buffer_col-2] = data_port_b;
end

always_ff @(posedge clk or posedge reset) begin
 if (reset) buffer_col <= 0;
 else begin
            if(buffermode | buffermode_d)
                            buffer_col <= buffer_col + 1;
             else
                    buffer_col <= 0;
 end
end

//*********************************************************************
```

```
//******** CONVOLUTED DATA STORING IN MEMORY ************************
//*****************************************************************

always_ff @(posedge clk or posedge reset) begin
        if(reset) offset_conv <= 0;
        else begin
                if(wren_b)
                  offset_conv <= offset_conv + 1;
                else
                  offset_conv <= 0          ;
        end
end

//*****************************************************************
//************* OUTPUT LOGIC OF DATA MATRIX ************************
//*****************************************************************

always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
                for (int l = 0; l < K_HEIGHT*K_WIDTH; l++) begin
                        data_mat[l] <= 0;
                end
        end
        else begin
          casez ({mat_row,mat_col})
                {f_row, f_col}  : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                        data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                                {buffer[2][mat_col  ], buffer[2][mat_col], buffer[2][mat_col+1] ,

                                                        buffer[2][mat_col  ], buffer[2][mat_col],
buffer[2][mat_col+1] ,

                                                        buffer[3][mat_col  ], buffer[3][mat_col],
buffer[3][mat_col+1]};

              {f_row, e_col}  : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                        data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                                {buffer[1][mat_col-1], buffer[1][mat_col], buffer[1][mat_col+1] ,

                                                        buffer[1][mat_col-1], buffer[1][mat_col],
buffer[1][mat_col+1] ,

                                                        buffer[2][mat_col-1], buffer[2][mat_col],
buffer[2][mat_col+1]};

                {f_row, l_col}  : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                        data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                                {buffer[1][mat_col-1], buffer[1][mat_col], buffer[1][mat_col  ] ,

                                                        buffer[1][mat_col-1], buffer[1][mat_col],
buffer[1][mat_col  ] ,

                                                        buffer[2][mat_col-1], buffer[2][mat_col],
buffer[2][mat_col  ]};

                {f_row, uk_col} : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                        data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                                {buffer[2][mat_col-1], buffer[2][mat_col], buffer[2][mat_col+1] ,

                                                        buffer[2][mat_col-1], buffer[2][mat_col],
buffer[2][mat_col+1] ,

                                                        buffer[3][mat_col-1], buffer[3][mat_col],
buffer[3][mat_col+1]};

                {l_row, f_col}  : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                        data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                                {buffer[1][mat_col  ], buffer[1][mat_col], buffer[1][mat_col+1] ,

                                                        buffer[2][mat_col  ], buffer[2][mat_col],
buffer[2][mat_col+1] ,

                                                        buffer[2][mat_col  ], buffer[2][mat_col],
buffer[2][mat_col+1]};

    {l_row, e_col}  : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                        {buffer[0][mat_col-1], buffer[0][mat_col], buffer[0][mat_col+1] ,
```

```
                                                        buffer[1][mat_col-1], buffer[1][mat_col],
buffer[1][mat_col+1] ,

                                                        buffer[1][mat_col-1], buffer[1][mat_col],
buffer[1][mat_col+1]};

                        {l_row, l_col}  : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                                        data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                                                {buffer[0][mat_col-1], buffer[0][mat_col], buffer[0][mat_col  ] ,

                                                        buffer[1][mat_col-1], buffer[1][mat_col],
buffer[1][mat_col  ] ,

                                                        buffer[1][mat_col-1], buffer[1][mat_col],
buffer[1][mat_col  ]};

    {l_row, uk_col} : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                        {buffer[1][mat_col-1], buffer[1][mat_col], buffer[1][mat_col+1] ,

                                                        buffer[2][mat_col-1], buffer[2][mat_col],
buffer[2][mat_col+1] ,

                                                        buffer[2][mat_col-1], buffer[2][mat_col],
buffer[2][mat_col+1]};

    {uk_row, f_col} : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                        {buffer[1][mat_col  ], buffer[1][mat_col], buffer[1][mat_col+1] ,

                                                        buffer[2][mat_col  ], buffer[2][mat_col],
buffer[2][mat_col+1] ,

                                                        buffer[3][mat_col  ], buffer[3][mat_col],
buffer[3][mat_col+1]};

    {uk_row, e_col} : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                        {buffer[0][mat_col-1], buffer[0][mat_col], buffer[0][mat_col+1] ,

                                                        buffer[1][mat_col-1], buffer[1][mat_col],
buffer[1][mat_col+1] ,

                                                        buffer[2][mat_col-1], buffer[2][mat_col],
buffer[2][mat_col+1]};

                        {uk_row, l_col} : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                                        data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                                                {buffer[0][mat_col-1], buffer[0][mat_col], buffer[0][mat_col  ] ,

                                                        buffer[1][mat_col-1], buffer[1][mat_col],
buffer[1][mat_col  ] ,

                                                        buffer[2][mat_col-1], buffer[2][mat_col],
buffer[2][mat_col  ]};

                        default         : {data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4],
                                        data_mat[3], data_mat[2], data_mat[1], data_mat[0]} <=
                                                {buffer[1][mat_col-1], buffer[1][mat_col], buffer[1][mat_col+1] ,

                                                        buffer[2][mat_col-1], buffer[2][mat_col],
buffer[2][mat_col+1] ,

                                                        buffer[3][mat_col-1], buffer[3][mat_col],
buffer[3][mat_col+1]};
              endcase
            end
end

//*********************************************************************
//************** INSTANSIATION OF RAM IP ****************************
//*********************************************************************

ram         ram_inst (
                        .address_a ( address_port_a ),
                        .address_b ( address_port_b ),
                        .clock ( clk ),
                        .data_a ( 8'd0 ),
```

```
                .data_b ( w_data ),
                .wren_a ( 1'b0 ),
                .wren_b ( wren_b ),
                .q_a ( data_port_a ),
                .q_b ( data_port_b )
);

endmodule
```

Test-Bench Line Buffer 32x32

```
`timescale 1ns/1ps
module tb_buffer ();

        logic       clk             = 0   ;
        logic       busy            = 0   ;
        logic       reset           = 1   ;
        logic       buffermode      = 0   ;
        logic       wren_b          = 0   ;
        logic       hold_mode       = 0   ;
        logic [ 7:0] w_data         = 0   ;
        logic [14:0] addr_image     = 0   ;
        logic [14:0] addr_kernal    = 400;
        logic [14:0] addr_conv      = 0   ;
        logic [ 4:0] mat_row        = 0   ;
        logic [ 4:0] mat_col        = 0   ;
        logic [ 7:0] data_mat   [3*3-1:0]     ;
        logic [ 7:0] kernal     [3*3-1:0]     ;

        logic [7:0] buffer_image[31:0][0:31];
        logic [7:0] data_mat_exp[ 8:0]     ;

        integer matched     = 0;
        integer miss_matched = 0;

        //--------------------------------------------------------------------------
        //--------------------- REAL IMAGE DATA ------------------------------------
        //--------------------------------------------------------------------------

  assign buffer_image[ 0] = {8'b10000111, 8'b10000110, 8'b10001001, 8'b10001001, 8'b10001001, 8'b10001010,
8'b10001010, 8'b10001001 ,
                8'b10001010, 8'b10001001, 8'b10001001, 8'b10001011, 8'b10001001, 8'b10001011,
8'b10001011, 8'b10001010 ,
                8'b10001011, 8'b10001010, 8'b10001011, 8'b10001100, 8'b10001100, 8'b10001100,
8'b10001100, 8'b10001011 ,
                8'b10001010, 8'b10001011, 8'b10001011, 8'b10001011, 8'b10001010, 8'b10001010,
8'b10001001, 8'b10001000};
  assign buffer_image[ 1] = {8'b10001001, 8'b10001001, 8'b10001011, 8'b10001011, 8'b10001011, 8'b10001011,
8'b10001100, 8'b10001011 ,
                8'b10001100, 8'b10001101, 8'b10001100, 8'b10001101, 8'b10001100, 8'b10001101,
8'b10001101, 8'b10001101 ,
                8'b10001101, 8'b10001110, 8'b10001110, 8'b10001110, 8'b10001110, 8'b10001100,
8'b10001100, 8'b10001110 ,
                8'b10001110, 8'b10001110, 8'b10001100, 8'b10001100, 8'b10001101, 8'b10001101,
8'b10001100, 8'b10001010};
  assign buffer_image[ 2] = {8'b10011101, 8'b10011101, 8'b10011010, 8'b10011101, 8'b10011110, 8'b10011101,
8'b10011110, 8'b10011101 ,
                8'b10011101, 8'b10011110, 8'b10011101, 8'b10011101, 8'b10011100, 8'b10011111,
8'b10100000, 8'b10011111 ,
                8'b10011110, 8'b10011110, 8'b10011110, 8'b10011110, 8'b10011111, 8'b10011111,
8'b10011111, 8'b10011110 ,
                8'b10011111, 8'b10100000, 8'b10100001, 8'b10100000, 8'b10100000, 8'b10100001,
8'b10011110, 8'b10011100};
  assign buffer_image[ 3] = {8'b10100010, 8'b10011110, 8'b10011100, 8'b10100000, 8'b10100001, 8'b10100010,
8'b10100001, 8'b10100001 ,
                8'b10100001, 8'b10100001, 8'b10100001, 8'b10100010, 8'b10100011, 8'b10100011,
8'b10100110, 8'b10100111 ,
                8'b10100100, 8'b10100010, 8'b10100010, 8'b10100100, 8'b10100011, 8'b10100100,
8'b10100101, 8'b10100011 ,
                8'b10100011, 8'b10100100, 8'b10100110, 8'b10100101, 8'b10100011, 8'b10100001,
8'b10100100, 8'b10100010};
  assign buffer_image[ 4] = {8'b10101001, 8'b10101000, 8'b10100111, 8'b10101000, 8'b10101001, 8'b10101001,
8'b10101000, 8'b10101000 ,
                8'b10101010, 8'b10101010, 8'b10101001, 8'b10101001, 8'b10111111, 8'b10001110,
8'b01110110, 8'b10101110 ,
                8'b10110010, 8'b10101100, 8'b10101100, 8'b10101100, 8'b10101101, 8'b10101101,
8'b10101101, 8'b10101100 ,
                8'b10101010, 8'b10110010, 8'b10100010, 8'b10100010, 8'b10110000, 8'b10101110,
8'b10110001, 8'b10101011};
```

```
 assign buffer_image[ 5] = {8'b10110011, 8'b10110011, 8'b10110001, 8'b10110010, 8'b10110011, 8'b10110011,
8'b10110100, 8'b10110010 ,
                    8'b10110010, 8'b10110011, 8'b10110010, 8'b11000100, 8'b10111111, 8'b10100000,
8'b01010110, 8'b01100100 ,
                    8'b10101011, 8'b10111101, 8'b10110110, 8'b10110111, 8'b10110111, 8'b10110111,
8'b10110101, 8'b10111001 ,
                    8'b10111101, 8'b10110001, 8'b10100110, 8'b01101111, 8'b10011011, 8'b10111101,
8'b10111010, 8'b10110111};
 assign buffer_image[ 6] = {8'b11000010, 8'b11000001, 8'b11000000, 8'b11000010, 8'b11000010, 8'b11000011,
8'b11000101, 8'b11000011 ,
                    8'b11000100, 8'b11000100, 8'b11001101, 8'b10111000, 8'b10100101, 8'b10111100,
8'b01101011, 8'b01001100 ,
                    8'b01100010, 8'b10100101, 8'b11000101, 8'b11001011, 8'b11000111, 8'b11000110,
8'b11000101, 8'b10111111 ,
                    8'b10101101, 8'b10100111, 8'b10111010, 8'b01100001, 8'b01011010, 8'b10110010,
8'b11001010, 8'b11000111};
 assign buffer_image[ 7] = {8'b11010011, 8'b11010000, 8'b11010000, 8'b11010010, 8'b11010011, 8'b11010101,
8'b11010011, 8'b11010100 ,
                    8'b11010010, 8'b11010101, 8'b11000111, 8'b10100010, 8'b11000110, 8'b10011010,
8'b01011100, 8'b01010110 ,
                    8'b01001111, 8'b01010011, 8'b01110110, 8'b10110101, 8'b11010000, 8'b11010001,
8'b11010110, 8'b10011101 ,
                    8'b10101001, 8'b11001001, 8'b10011111, 8'b01101110, 8'b01001111, 8'b01011101,
8'b10110010, 8'b11000101};
 assign buffer_image[ 8] = {8'b11010110, 8'b11010100, 8'b11011100, 8'b11011010, 8'b11011100, 8'b11010110,
8'b11101000, 8'b11001100 ,
                    8'b11001011, 8'b11010000, 8'b10110011, 8'b11000001, 8'b10010100, 8'b01110110,
8'b01011111, 8'b01011011 ,
                    8'b01010100, 8'b01010100, 8'b01101001, 8'b01011111, 8'b01100011, 8'b10011000,
8'b10101001, 8'b10100000 ,
                    8'b11010111, 8'b10100010, 8'b10000011, 8'b01110000, 8'b01011101, 8'b10100010,
8'b10101001, 8'b01101010};
 assign buffer_image[ 9] = {8'b11010101, 8'b11011001, 8'b11011100, 8'b10010001, 8'b10000011, 8'b10000010,
8'b01101001, 8'b01000010 ,
                    8'b01110000, 8'b11001110, 8'b11001010, 8'b10100101, 8'b01110101, 8'b01111100,
8'b01011101, 8'b01011000 ,
                    8'b01010001, 8'b10001010, 8'b11011010, 8'b01111011, 8'b00100111, 8'b01011000,
8'b10011010, 8'b11010111 ,
                    8'b10101011, 8'b10000100, 8'b01111100, 8'b01110100, 8'b10110010, 8'b10110101,
8'b10000010, 8'b01101001};
 assign buffer_image[10] = {8'b10110101, 8'b10011101, 8'b11011010, 8'b01110111, 8'b00110111, 8'b10100111,
8'b01011110, 8'b00010111 ,
                    8'b00010100, 8'b10110111, 8'b11010000, 8'b01111110, 8'b01111101, 8'b01100011,
8'b01101000, 8'b01101111 ,
                    8'b10001010, 8'b11100000, 8'b11100010, 8'b10011110, 8'b00110100, 8'b00100011,
8'b11010011, 8'b11000000 ,
                    8'b10000001, 8'b10000110, 8'b10010001, 8'b11000110, 8'b10101001, 8'b10001100,
8'b01110100, 8'b10010010};
 assign buffer_image[11] = {8'b10011111, 8'b01010101, 8'b10001111, 8'b10111101, 8'b10011110, 8'b01011010,
8'b01001100, 8'b00110100 ,
                    8'b00011110, 8'b10000011, 8'b10110110, 8'b01111100, 8'b01100001, 8'b10001001,
8'b10111110, 8'b10011011 ,
                    8'b10110111, 8'b11101110, 8'b11010011, 8'b10010001, 8'b01010100, 8'b00001110,
8'b01100100, 8'b10011011 ,
                    8'b10010101, 8'b10101100, 8'b10101101, 8'b10100001, 8'b10010100, 8'b01110011,
8'b10001101, 8'b10110011};
 assign buffer_image[12] = {8'b01110111, 8'b01011001, 8'b01101001, 8'b01011111, 8'b10001110, 8'b10100100,
8'b01010100, 8'b10010100 ,
                    8'b00101101, 8'b01110010, 8'b10011101, 8'b01110111, 8'b10010011, 8'b11001001,
8'b01111010, 8'b01000000 ,
                    8'b10011001, 8'b11011111, 8'b11001001, 8'b10001111, 8'b01101011, 8'b00101011,
8'b00100101, 8'b00110001 ,
                    8'b10001001, 8'b11011111, 8'b10010011, 8'b10111001, 8'b10010100, 8'b10000001,
8'b10101100, 8'b10000010};
 assign buffer_image[13] = {8'b01101011, 8'b01011110, 8'b01101110, 8'b01010110, 8'b01010100, 8'b11000000,
8'b11001101, 8'b10001011 ,
                    8'b00011000, 8'b10000100, 8'b10000111, 8'b01111110, 8'b11100111, 8'b10011011,
8'b00111110, 8'b01111001 ,
                    8'b11101000, 8'b11011001, 8'b10111110, 8'b10110000, 8'b10000011, 8'b01110010,
8'b01110010, 8'b01110101 ,
                    8'b00101100, 8'b10001001, 8'b10111101, 8'b10100111, 8'b10100010, 8'b10111111,
8'b10011010, 8'b01011000};
 assign buffer_image[14] = {8'b01101010, 8'b01011100, 8'b01101101, 8'b01101100, 8'b10110110, 8'b11011010,
8'b10110101, 8'b01100011 ,
                    8'b00101101, 8'b01000010, 8'b01101001, 8'b10110010, 8'b11011011, 8'b01101101,
8'b01001010, 8'b10100100 ,
                    8'b11101101, 8'b11101001, 8'b11101010, 8'b11011000, 8'b10011101, 8'b01110000,
8'b01101001, 8'b10010101 ,
```

```
                8'b01100110, 8'b00100101, 8'b10010001, 8'b10111111, 8'b10110011, 8'b10101001,
8'b10011111, 8'b01100010};
  assign buffer_image[15] = {8'b01011011, 8'b01101011, 8'b10011111, 8'b10100101, 8'b10100110, 8'b10011011,
8'b01101100, 8'b10011111 ,
                8'b01010000, 8'b01011000, 8'b10010101, 8'b11001111, 8'b11010111, 8'b01100111,
8'b01100011, 8'b01010110 ,
                8'b11000011, 8'b11011111, 8'b11001100, 8'b10111001, 8'b10100000, 8'b01010110,
8'b00010011, 8'b01101001 ,
                8'b10100010, 8'b00101101, 8'b00100011, 8'b11000010, 8'b10011101, 8'b10110110,
8'b10011001, 8'b01100101};
  assign buffer_image[16] = {8'b10010000, 8'b10110100, 8'b10001110, 8'b01011111, 8'b01101001, 8'b01110000,
8'b01011000, 8'b10011000 ,
                8'b10011100, 8'b10111010, 8'b11000001, 8'b11000100, 8'b11010100, 8'b01101010,
8'b01001100, 8'b00011011 ,
                8'b01101001, 8'b10110101, 8'b10011101, 8'b01111010, 8'b01001101, 8'b00011011,
8'b00000101, 8'b01100100 ,
                8'b11001100, 8'b00111010, 8'b00101110, 8'b10010010, 8'b10101010, 8'b10011100,
8'b01111111, 8'b01101101};
  assign buffer_image[17] = {8'b10110100, 8'b10101101, 8'b10000000, 8'b01100110, 8'b01101000, 8'b01101110,
8'b01100101, 8'b01000111 ,
                8'b01101110, 8'b10100001, 8'b10011010, 8'b01100100, 8'b10111101, 8'b01011110,
8'b00110110, 8'b00010101 ,
                8'b00011011, 8'b01101001, 8'b01101010, 8'b00110110, 8'b00001110, 8'b00000110,
8'b00000101, 8'b01010111 ,
                8'b11011010, 8'b01001011, 8'b00101110, 8'b10001111, 8'b10110000, 8'b01110101,
8'b10000111, 8'b10101010};
  assign buffer_image[18] = {8'b10110111, 8'b10111000, 8'b01111011, 8'b01110101, 8'b01101000, 8'b01101011,
8'b01100110, 8'b10000100 ,
                8'b10000101, 8'b01010101, 8'b01000011, 8'b01000001, 8'b01100000, 8'b10010100,
8'b01100110, 8'b00110011 ,
                8'b00000110, 8'b00011101, 8'b00100000, 8'b00000110, 8'b00000111, 8'b01001011,
8'b00100110, 8'b00111011 ,
                8'b11101000, 8'b01010111, 8'b00111100, 8'b10000100, 8'b10000011, 8'b10000111,
8'b10101100, 8'b10110011};
  assign buffer_image[19] = {8'b10100011, 8'b10000001, 8'b10011000, 8'b01101100, 8'b01110011, 8'b01100111,
8'b01111101, 8'b10111010 ,
                8'b11000001, 8'b10000101, 8'b01110110, 8'b10110100, 8'b01001111, 8'b01101011,
8'b10100010, 8'b00010101 ,
                8'b00111001, 8'b00111101, 8'b01000110, 8'b01101010, 8'b10100000, 8'b10111000,
8'b00111001, 8'b00001110 ,
                8'b10100010, 8'b01101000, 8'b00101110, 8'b00110110, 8'b10000110, 8'b10100000,
8'b10110010, 8'b11000101};
  assign buffer_image[20] = {8'b01111000, 8'b01100101, 8'b10010000, 8'b01101111, 8'b01110000, 8'b10001011,
8'b10101100, 8'b10100101 ,
                8'b10010011, 8'b01111001, 8'b10110101, 8'b11010101, 8'b01101101, 8'b00111001,
8'b10001111, 8'b00110111 ,
                8'b10100011, 8'b10001111, 8'b11010101, 8'b11000100, 8'b10011001, 8'b10100101,
8'b01111110, 8'b00000110 ,
                8'b00101001, 8'b00111100, 8'b00101110, 8'b00110100, 8'b10010111, 8'b10111110,
8'b11000010, 8'b10010110};
  assign buffer_image[21] = {8'b01100101, 8'b01101100, 8'b10000100, 8'b01101010, 8'b10110001, 8'b10101010,
8'b10000001, 8'b10100011 ,
                8'b10000110, 8'b10110101, 8'b11001111, 8'b10001100, 8'b01110111, 8'b00100111,
8'b10000111, 8'b00111100 ,
                8'b01110111, 8'b11000001, 8'b10110110, 8'b10000001, 8'b10011111, 8'b11011001,
8'b11000001, 8'b00010011 ,
                8'b00100011, 8'b01101101, 8'b00111000, 8'b01100011, 8'b11000100, 8'b11001110,
8'b10000011, 8'b10000101};
  assign buffer_image[22] = {8'b01011111, 8'b01011110, 8'b10000111, 8'b11000111, 8'b10010111, 8'b01111000,
8'b10011011, 8'b10011110 ,
                8'b11000000, 8'b10110100, 8'b10000010, 8'b10001000, 8'b10000101, 8'b00111111,
8'b01110010, 8'b01011000 ,
                8'b11000001, 8'b11000010, 8'b10001110, 8'b10010010, 8'b11000100, 8'b10110011,
8'b10010101, 8'b00101010 ,
                8'b01000100, 8'b11010100, 8'b10011011, 8'b01000100, 8'b11001010, 8'b10001100,
8'b10000101, 8'b10100001};
  assign buffer_image[23] = {8'b01101110, 8'b10001111, 8'b11001110, 8'b10101000, 8'b01110001, 8'b10011000,
8'b10111100, 8'b11001011 ,
                8'b10100111, 8'b10000110, 8'b01101100, 8'b10001000, 8'b11000001, 8'b10000001,
8'b01110011, 8'b10000000 ,
                8'b11000100, 8'b10100011, 8'b10000111, 8'b10111110, 8'b10111100, 8'b10001101,
8'b10010010, 8'b00110110 ,
                8'b01001101, 8'b10110100, 8'b10101000, 8'b01010011, 8'b01101100, 8'b01101111,
8'b10100001, 8'b10111100};
  assign buffer_image[24] = {8'b11000000, 8'b11000010, 8'b10100110, 8'b10000101, 8'b10101101, 8'b11001001,
8'b10101100, 8'b01111111 ,
                8'b10000111, 8'b01111110, 8'b10010111, 8'b10111111, 8'b10110101, 8'b01111111,
8'b01111100, 8'b01101111 ,
```

```
                8'b10010100, 8'b10001100, 8'b10110011, 8'b11001010, 8'b10011110, 8'b10011010,
8'b10111100, 8'b01010100 ,
                8'b00111011, 8'b10110010, 8'b11000110, 8'b01011000, 8'b01001111, 8'b10100111,
8'b11000100, 8'b10101010};
  assign buffer_image[25] = {8'b10101110, 8'b10010111, 8'b11000000, 8'b11001110, 8'b11001011, 8'b10011110,
8'b10001000, 8'b10001111 ,
                8'b10010110, 8'b10111001, 8'b11001001, 8'b10100100, 8'b10110011, 8'b10100000,
8'b01010000, 8'b01100000 ,
                8'b10010111, 8'b10111000, 8'b11001110, 8'b10100100, 8'b10110010, 8'b11001010,
8'b10111000, 8'b00101100 ,
                8'b10010000, 8'b11010000, 8'b10110010, 8'b01100111, 8'b10011011, 8'b11010101,
8'b10110101, 8'b10100110};
  assign buffer_image[26] = {8'b11100000, 8'b11100100, 8'b11100110, 8'b11100010, 8'b11011110, 8'b11100001,
8'b11100111, 8'b11100011 ,
                8'b11100101, 8'b11100111, 8'b11100000, 8'b11100010, 8'b11100101, 8'b11010000,
8'b01010100, 8'b10010001 ,
                8'b11100011, 8'b11100101, 8'b11100000, 8'b11100011, 8'b11100110, 8'b11101000,
8'b10100010, 8'b01001001 ,
                8'b11100011, 8'b11100011, 8'b11100111, 8'b10000000, 8'b11001100, 8'b11100011,
8'b11100000, 8'b11100010};
  assign buffer_image[27] = {8'b11010111, 8'b11010111, 8'b11011000, 8'b11011010, 8'b10110010, 8'b01110001,
8'b10111001, 8'b11011110 ,
                8'b10101110, 8'b11000010, 8'b11011011, 8'b11011001, 8'b11011011, 8'b11010001,
8'b01011011, 8'b10001011 ,
                8'b11011011, 8'b11011011, 8'b11011100, 8'b11011100, 8'b11011101, 8'b11100000,
8'b01010001, 8'b10101000 ,
                8'b11100001, 8'b11011101, 8'b11011110, 8'b10010100, 8'b11010110, 8'b11011110,
8'b11011111, 8'b11100000};
  assign buffer_image[28] = {8'b10111100, 8'b10111110, 8'b10111110, 8'b10110101, 8'b10011001, 8'b10001101,
8'b10011110, 8'b11000000 ,
                8'b10101110, 8'b10110000, 8'b11000100, 8'b11000011, 8'b11000011, 8'b10110000,
8'b01100111, 8'b01111000 ,
                8'b11000110, 8'b11000110, 8'b11000100, 8'b11000101, 8'b11001010, 8'b01111101,
8'b01001001, 8'b11000011 ,
                8'b10110101, 8'b10110000, 8'b10101001, 8'b01111100, 8'b10101001, 8'b10101011,
8'b10101011, 8'b10101101};
  assign buffer_image[29] = {8'b10111100, 8'b10100010, 8'b10111100, 8'b11001011, 8'b11100011, 8'b10100110,
8'b10010110, 8'b11001010 ,
                8'b11001000, 8'b11000110, 8'b11000011, 8'b11000111, 8'b10110011, 8'b01110110,
8'b01110010, 8'b01011100 ,
                8'b10000110, 8'b10000011, 8'b10001001, 8'b10000110, 8'b01110101, 8'b00110110,
8'b01010010, 8'b01011011 ,
                8'b01010110, 8'b01010000, 8'b01110101, 8'b00111111, 8'b00101100, 8'b00111111,
8'b00111110, 8'b00111111};
  assign buffer_image[30] = {8'b10110011, 8'b10110110, 8'b11001100, 8'b11010000, 8'b10110101, 8'b01000110,
8'b00011000, 8'b10001011 ,
                8'b10111001, 8'b11000000, 8'b11000010, 8'b11000100, 8'b11000000, 8'b10111110,
8'b10110011, 8'b10101010 ,
                8'b10101100, 8'b10101101, 8'b11000001, 8'b11000111, 8'b10101110, 8'b11000111,
8'b11000010, 8'b10111101 ,
                8'b11010001, 8'b10101100, 8'b10010000, 8'b10101111, 8'b10111011, 8'b11000110,
8'b10110111, 8'b10111011};
  assign buffer_image[31] = {8'b10101011, 8'b10111000, 8'b10100110, 8'b10001001, 8'b01101111, 8'b01010110,
8'b01011010, 8'b01011000 ,
                8'b01101011, 8'b10100010, 8'b11000001, 8'b11000010, 8'b11000100, 8'b10111011,
8'b10111000, 8'b10110100 ,
                8'b10111010, 8'b11000111, 8'b11011111, 8'b10101100, 8'b00111100, 8'b10000010,
8'b10110100, 8'b11001101 ,
                8'b11001100, 8'b10001001, 8'b01000011, 8'b01100100, 8'b10100010, 8'b11000111,
8'b10010011, 8'b10000110};

        buffer #(
                .WIDTH      (32),
                .HEIGHT     (32),
                .K_WIDTH    (3 ),
                .K_HEIGHT   (3 ),
                .ADDR_W     (5 ),
                .ADDR_H     (5 ),
                .COLOUR_DEPTH(8 ),
                .MEMORY_ADDR (15)
        ) i_buffer (
                .clk       (clk       ),
                .busy      (busy      ),
                .reset     (reset     ),
                .buffermode (buffermode ),
                .wren_b    (wren_b    ),
                .hold_mode  (hold_mode  ),
                .w_data    (w_data    ),
                .addr_image (addr_image ),
```

```
                        .addr_kernal(addr_kernal),
                        .addr_conv  (addr_conv  ),
                        .mat_row    (mat_row    ),
                        .mat_col    (mat_col    ),
                        .data_mat   (data_mat   ),
                        .kernal     (kernal     )
                );

always #5 clk <= ~clk;


//-------------------------------------------------------------------------------
//------ THE COUNTER TO SELECT THE ROW AND COLUMN TO PERFORM CONVOLUTION ----------
//-------------------------------------------------------------------------------

always_ff @(posedge clk) begin
                if(wren_b) begin
                        if (mat_col == 5'b11111) begin
                                mat_col <= 0;
                                mat_row <= mat_row + 1;
                        end
                        else begin
                                mat_col <= mat_col + 1;
                        end
                end
                else begin
                        mat_col <= 0;
                        mat_row <= 0;
                end
end

//-------------------------------------------------------------------------------
//-------------- THE EXPECTED OUTPUT GENERATION ---------------------------------------
//-------------------------------------------------------------------------------

always_ff @(posedge clk) begin
                if(wren_b)begin
                        casez ({mat_row, mat_col})
    10'b00000_00000 : {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6], data_mat_exp[5],
data_mat_exp[4], data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]}
                         <= {buffer_image[mat_row  ][mat_col  ], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col+1] ,
                                buffer_image[mat_row  ][mat_col  ], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col+1] ,
                                buffer_image[mat_row+1][mat_col  ], buffer_image[mat_row+1][mat_col],
buffer_image[mat_row+1][mat_col+1]};

    10'b00000_11111 : {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6], data_mat_exp[5],
data_mat_exp[4], data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]}
                         <= {buffer_image[mat_row  ][mat_col-1], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col  ] ,
                                buffer_image[mat_row  ][mat_col-1], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col  ] ,
                                buffer_image[mat_row+1][mat_col-1], buffer_image[mat_row+1][mat_col],
buffer_image[mat_row+1][mat_col  ]};

    10'b11111_00000: {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6], data_mat_exp[5],
data_mat_exp[4], data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]}
                         <= {buffer_image[mat_row-1][mat_col  ], buffer_image[mat_row-1][mat_col],
buffer_image[mat_row-1][mat_col+1] ,
                                buffer_image[mat_row  ][mat_col  ], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col+1] ,
                                buffer_image[mat_row  ][mat_col  ], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col+1]};

    10'b11111_11111 : {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6], data_mat_exp[5],
data_mat_exp[4], data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]}
                         <= {buffer_image[mat_row-1][mat_col-1], buffer_image[mat_row-1][mat_col],
buffer_image[mat_row-1][mat_col  ] ,
                                buffer_image[mat_row  ][mat_col-1], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col  ] ,
                                buffer_image[mat_row  ][mat_col-1], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col  ]};

    10'b00000_????? : {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6], data_mat_exp[5],
data_mat_exp[4], data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]}
                         <= {buffer_image[mat_row  ][mat_col-1], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col+1] ,
```

```
                        buffer_image[mat_row  ][mat_col-1], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col+1] ,
                        buffer_image[mat_row+1][mat_col-1], buffer_image[mat_row+1][mat_col],
buffer_image[mat_row+1][mat_col+1]};

    10'b11111_????? : {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6], data_mat_exp[5],
data_mat_exp[4], data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]}
                        <= {buffer_image[mat_row-1][mat_col-1], buffer_image[mat_row-1][mat_col],
buffer_image[mat_row-1][mat_col+1] ,
                        buffer_image[mat_row  ][mat_col-1], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col+1] ,
                        buffer_image[mat_row  ][mat_col-1], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col+1]};

    10'b?????_11111 : {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6], data_mat_exp[5],
data_mat_exp[4], data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]}
                        <= {buffer_image[mat_row-1][mat_col-1], buffer_image[mat_row-1][mat_col],
buffer_image[mat_row-1][mat_col  ] ,
                        buffer_image[mat_row  ][mat_col-1], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col  ] ,
                        buffer_image[mat_row+1][mat_col-1], buffer_image[mat_row+1][mat_col],
buffer_image[mat_row+1][mat_col  ]};

    10'b?????_00000 : {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6], data_mat_exp[5],
data_mat_exp[4], data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]}
                        <= {buffer_image[mat_row-1][mat_col  ], buffer_image[mat_row-1][mat_col],
buffer_image[mat_row-1][mat_col+1] ,
                        buffer_image[mat_row  ][mat_col  ], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col+1] ,
                        buffer_image[mat_row+1][mat_col  ], buffer_image[mat_row+1][mat_col],
buffer_image[mat_row+1][mat_col+1]};

                                default       : {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6],
data_mat_exp[5], data_mat_exp[4], data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]}
                                                <= {buffer_image[mat_row-1][mat_col-1], buffer_image[mat_row-
1][mat_col], buffer_image[mat_row-1][mat_col+1] ,
                        buffer_image[mat_row  ][mat_col-1], buffer_image[mat_row  ][mat_col],
buffer_image[mat_row  ][mat_col+1] ,
                        buffer_image[mat_row+1][mat_col-1], buffer_image[mat_row+1][mat_col],
buffer_image[mat_row+1][mat_col+1]};
                        endcase
            end
            else
                        {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6], data_mat_exp[5], data_mat_exp[4],
data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]} <= 0;
end

//-------------------------------------------------------------------------------------------
//------ THE INPUTS GIVEN IN A PARTICULAR ORDER AS SHOULD BE SEND BY THE TOP MODULE ----------
//-------------------------------------------------------------------------------------------

initial begin
            #10  reset     = 0     ;
                addr_image = 0     ;
                addr_kernal= 15'h400;
                addr_conv  = 15'h400;
            #10  busy      = 1     ;
                buffermode = 1     ;
            #90  buffermode = 0     ;
                hold_mode  = 1     ;
            #275 hold_mode  = 0     ;
                wren_b     = 1     ;
            repeat (1024) @(posedge clk);
            $display("matched = %d", matched);
            $display("miss_matched = %d",miss_matched);
            $stop;
end

//-------------------------------------------------------------------------------
//----------- COMPARING THE OUTPUT WITH THE EXPECTED OUTPUT ----------------------
//-------------------------------------------------------------------------------

always_ff @(negedge clk) begin
            if (wren_b) begin
                        if ({data_mat[8], data_mat[7], data_mat[6], data_mat[5], data_mat[4], data_mat[3],
data_mat[2], data_mat[1], data_mat[0]} ==
                                    {data_mat_exp[8], data_mat_exp[7], data_mat_exp[6], data_mat_exp[5],
data_mat_exp[4], data_mat_exp[3], data_mat_exp[2], data_mat_exp[1], data_mat_exp[0]})
                                matched <= matched + 1;
```

```
                        else
                                miss_matched <= miss_matched + 1;
 end
end

endmodule
```

Convolution Process

```
module conv_proc #(
        parameter KERNAL_WIDTH  = 3,
        parameter KERNAL_HEIGHT = 3,
        parameter COLOUR_DEPTH  = 8
) (
        input                   clk                             ,
        input                   reset                           ,
        input  logic [COLOUR_DEPTH-1:0] data_mat [KERNAL_WIDTH*KERNAL_HEIGHT-1:0],
        input  logic [COLOUR_DEPTH-1:0] kernal   [KERNAL_WIDTH*KERNAL_HEIGHT-1:0],
        output logic [COLOUR_DEPTH-1:0] w_data
);

        logic   [2*COLOUR_DEPTH-1:0] partial_product[KERNAL_HEIGHT*KERNAL_WIDTH-1:0];
        logic   [2*COLOUR_DEPTH-1:0] partial_add                        ;
        integer                 i                               ;
        integer                 j                               ;
        logic   [7:0] answer_product;

//-------------------------------------------------------------
//------ PRODUCT OF THE DATA MATRIX WITH KERNAL --------------
//-------------------------------------------------------------

always_comb begin
        for (i = 0; i < (KERNAL_WIDTH*KERNAL_HEIGHT) ; i++) begin
                partial_product[i] = data_mat[i] * kernal[i];
        end
end

//-------------------------------------------------------------
//---------- ADDING THE PARTIAL PRODUCT ----------------------
//-------------------------------------------------------------

always_comb begin
        partial_add = 0;
        for (j = 0; j < (KERNAL_WIDTH*KERNAL_HEIGHT) ; j++) begin
                partial_add = partial_add + partial_product[j];
        end
end

//-------------------------------------------------------------
//------ TRUNCATING THE 16 BIT OUTPUT TO 8 BIT _--------------
//-------------------------------------------------------------

assign answer_product = data_mat[4];
assign w_data = {answer_product[7],partial_add[11:5]};

Endmodule
```

Test-Bench Convolution Process

```
module tb_conv_proc ;

// Testbench signals
logic      clk             = 0;
logic      reset           = 1;
logic [7:0] data_mat[3*3-1:0]    ;
logic [7:0] kernal  [3*3-1:0]    ;
logic [7:0] w_data              ;

// Expected output
logic [7:0] expected;

integer matched = 0;
integer mismatched = 0;
logic [15:0] wiring;

conv_proc #(
        .KERNAL_WIDTH (3),
        .KERNAL_HEIGHT(3),
```

```
            .COLOUR_DEPTH (8)
) process_1 (
            .clk    (clk    ),
            .reset  (reset  ),
            .data_mat(data_mat),
            .kernal (kernal ),
            .w_data (w_data )
);

always #5 clk <= ~clk;

always_ff @(posedge clk) begin
            for (int i = 0; i < 9; i++) begin
                        data_mat[i] <= $urandom_range(0,127);
                        kernal  [i] <= $urandom_range(0,10 );
            end
end

always_comb begin : proc_
  wiring = 0;
            for (int k = 0; k < 9; k++) begin
              wiring = wiring + data_mat[k]*kernal[k];
            end
  expected = {data_mat[4][7:6], wiring[5:0]};
end

initial begin
            #5 reset = 0;
  #1000;
  $display("mismatched = %d",mismatched);
  $display("matched = %d",matched);
  $stop;
end

always_ff @(negedge clk) begin
            if(w_data == expected)
                        matched <= matched + 1;
            else
                        mismatched <= mismatched + 1;
end

endmodule
```