ISTANBUL TECHNICAL UNIVERSITY

Digital System Design Applications

Experiment VIII
IMAGE PROCESSING SYSTEM

2024

In this experiment, students will implement an image processing system. A 640x480 12-bit RGB input image will be loaded up to Block RAM. Using a given kernel, 2-D convolution will be applied onto the input image. Output image will be showed onto the VGA monitor. Output image will aslo be recorded to a text file, then will be examined in MATLAB.

Objectives

- Creating an image processing system from scratch.
- Creating testbenches to generate output text files.
- Showing output image onto the monitor by VGA interface.

Requirements

Students are expected to know;

- The general concept of 2-D convolution,
- How to design finite state machines.
- How to set and instantiate Xilinx Block RAM IPs.
- How to record design outputs to a text file using testbenches.

Experiment Report Checklist

1. Codes for newly created modules (do not include the modules written in previous experiments).
2. Clear explanations for designing the processes of submodules.
3. Show post-sytnhesis FPGA resource utilization of the full design (number of LUTs, flip flops and I/O).
4. Testbench code, if edited or re-written.
5. MATLAB screenshots of input and output images.

- **Projects and reports are to be done INDIVIDUALLY. High amounts of points will be deducted from similar works.**
- **Reports must be written in a proper manner. Divide your text to sections and subsections if needed, label your figures and connect your sections with proper explanations of your works. Reports filled with imprecisely placed tables and figures, with no verbal explanations in workflow, will not fare well.**
- **Check homeworks section in Ninova for submission dates.**

Preliminary

2D-Convolution

- An image is defined as a two-dimensional function, $f(x,y)$, where x **and** y are spatial (plane) coordinates and the **amplitude of** f at any pair of coordinates (x,y) is called the intensity of the image at that point. Each point refers to a pixel in an image. Images are represented in rows and columns by matrices:

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & f(0,2) & \dots & f(0,N-1) \\ f(1,0) & f(1,1) & f(1,2) & \dots & f(1,N-1) \\ . & . & . & & . \\ . & . & . & & . \\ . & . & . & & . \\ f(M-1,0) & f(M-1,1) & f(M-1,2) & \dots & f(M-1,N-1) \end{bmatrix}$$

Figure 1: A digital image

- Images can be classified as grayscale images and colored images. **Grayscale images** are formed by shades of gray. For example, in 8-bit grayscale images value of a pixel is represented by 8-bits resulting in a 256 different shades of gray. On the other hand, colored images are formed by color channels. **24-bit RGB images** has three channels in which 8-bit intensities corresponds to red, green and blue colors for a pixel.

- **Image filtering** is used in a broad spectrum of image processing applications such as blurring, sharpening, embossing, edge detection and denoising [1]. Image filtering can be done in spatial and frequency domains. In this experiment, spatial filtering will be applied. Spatial filters are typically 3x3, 5x5 and 7x7 odd number sized square matrices. These matrices are called **kernel**.

- A linear spatial filter performs a sum-of-products operation **(convolution)** between an **image** f and a **filter kernel** w. The **kernel** w is a matrix whose size is $m \times n$, where $m = 2a+1$, $n = 2b+1$. The **image** f is also a matrix whose size is $M \times N$. The size of the result is $(M+m-1) \times (N+n-1)$.

- The 2D-convolution formula is as follows [1]

$$g(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} w(i,j) f(x-i, y-j)$$

  where x and y are varied so that the center (origin) of the kernel visits every pixel in f once. For a fixed location of (x,y), the equation implements the sum of products and provides the new value at that location.

- Consider a 5×5 **image** and 3×3 **kernel**. The **convolution** of the image will be a **3 x 3 matrix**. It is the result of the sum of products for each **3 x 3 area** in the image. Visually, sliding the kernel window onto the image both horizontally and vertically corresponds to the area in which convolution applied and the center of the kernel corresponds to the output pixel location (see the **convolution.gif** in Ninova).

- An example of the **convolution** of a **5 x 5 matrix** image with a **3 x 3 kernel convolution_5x5.m** and an another example for a **512 x 512 image** with the same kernel **convolution_image.m** are shared in the Ninova. The **3 x 3 kernel** in these examples is

called **Laplacian Filter** which is used for edge detection. 3x3 **Laplacian Kernel** :

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

- Multiply and accumulate units **(MAC)** perform convolution operations by computing products of data and weights and adds these products to an accumulator. The structure of a MAC unit is shown in **Figure 2**. This unit perform 3x3 convolution operation in three cycles.

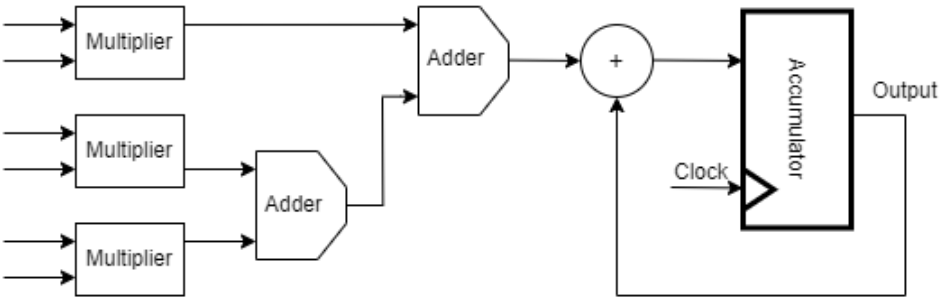

Figure 2: MAC Unit with 3 Parallel Multiplier

- A fully parallel architecture with nine multipliers can be seen below:

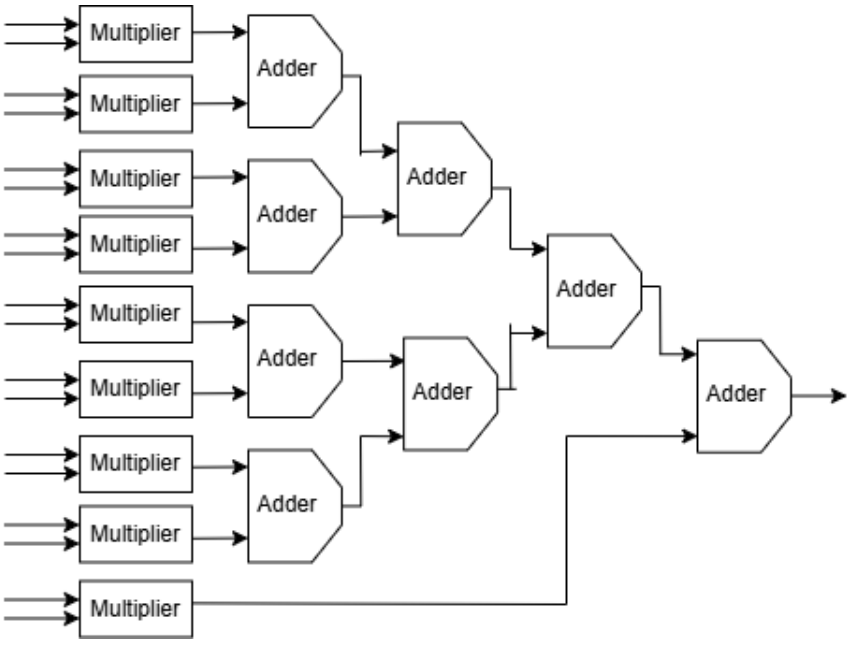

Figure 3: MAC Unit with 9 Parallel Multiplier

VGA Interface

- VGA monitors are CRTs (cathode ray tubes). Their most basic mode is 640 x 480 x 60Hz VGA; it consists of 640 columns by 480 lines of pixels, refreshed 60 times per second. It employs three colors (R = red, G = green, B = blue) per pixel, with the intensity of each color determined by an analog voltage in the 0V-to-0.7V range. The pixel signals are generated digitally, then converted to analog by the three DACs. These analog voltages constitute the image that will be displayed by the VGA monitor [2].
- Pin configurations of a VGA connector can be seen below:

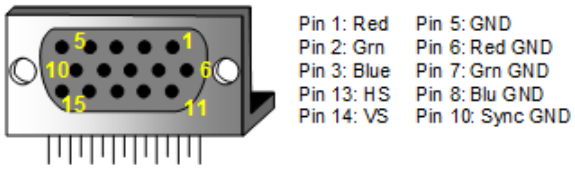

Figure 4: VGA

- The frequency for the default VGA is 25 MHz. Five control signals are included: Hsync (horizontal synchronism), Vsync (vertical synchronism), Hactive (portion of Hsync during which pixels are displayed), Vactive (portion of Vsync during which lines of pixels are displayed), and dena (display enable).
- Hsync and Vsync are responsible for determining when a new frame should start. Hactive and Vactive represent the time intervals during which the pixels of an image are driven on the screen. Finally, dena is responsible for turning the pixel signals ON&OFF. The control signals are Hsync, Vsync, Hactive, Vactive, and dena, and their timings define the VGA mode.
- The waveforms for Hsync and Hactive (based on pixel_clk), which consist of four parts (all measured in number of pixels; i.e., number of clock cycles), called Hpulse (width of the horizontal synchronization pulse), HBP (horizontal back porch), Hactive (active line display interval), and HFP (horizontal front porch).
- The vertical timing diagram is also consisting of four parts (all measured in number of lines or number of Hsync cycles), called Vpulse (width of the vertical synchronization pulse), VBP (vertical back porch), Vactive (active column display interval), and VFP (vertical front porch).

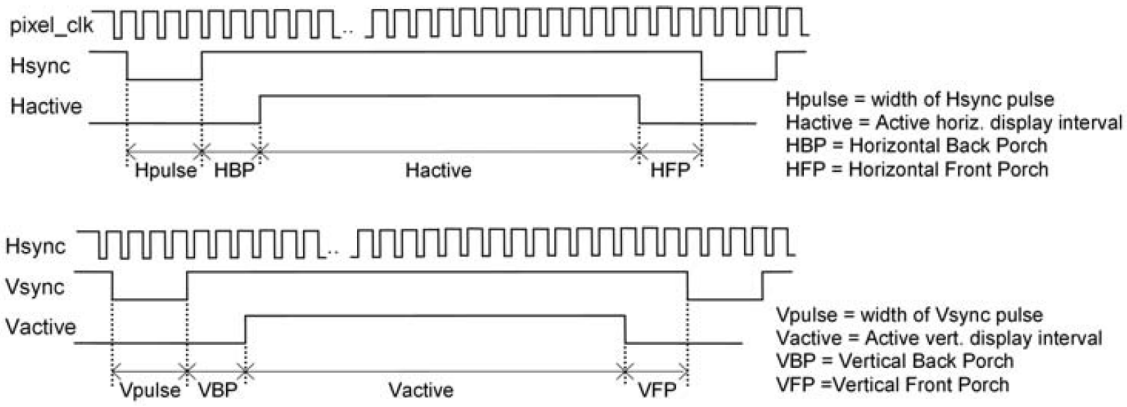

Figure 5: HSYNC adn VSYNC [2]

Implementation of an Image Processing System

- The image processing system with the top level schematic given at **Figure 6** will be realized in this experiment. Each submodule will be designed separately and connected together in a top module. Each submodule and top module will be simulated and verified. The output image will be recorded to a textfile using a testbench and verified with the MATLAB result. Finally, the system will be shown on a monitor by VGA interface.
- The system will apply Laplacian Filter to a 640 x 480 image (with zero padding 642x482). The clock frequency for the overall system will be 25MHz. The RGB pixel values will be 4 bits (12-bit RGB) due to the board limitations. The board allows 4-bit digital to analog converter (DAC) for the VGA connector.
- Nexsys A7-100T board will be used in this experiment. Create a project for Nexsys A7-100T board. Include **Nexys-A7-100T-Master.xdc** file which is provided in Ninova.

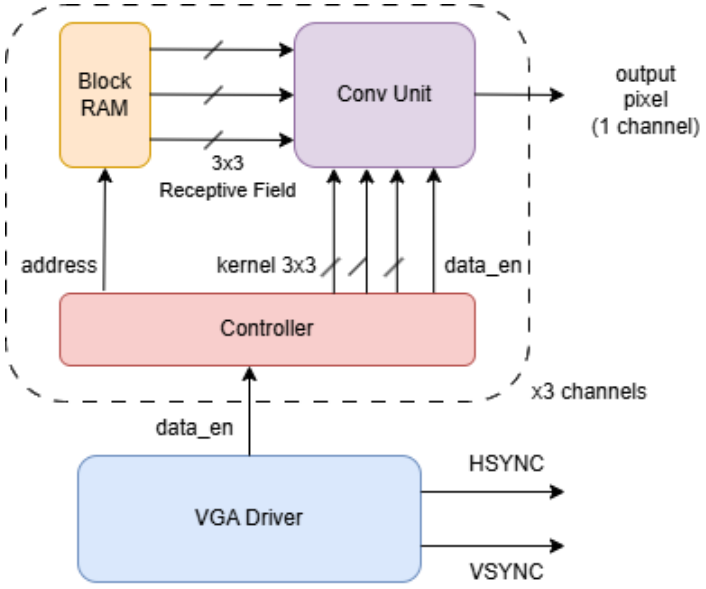

Figure 6: Block Diagram of the Image Processing System

Convolution Unit

1. Create a new module called **conv_unit**. This module should have 1-bit inputs **pixel_clk** which is 25MHz,**rst** and **enable** which corresponds to data_en in VGA Controller. 12-bit inputs **pixel1**, **pixel2**, **pixel3**, **kernel1**, **kernel2** and , **kernel3**; 4-bit output **pixel_out**.
2. 12-bit (3x4-bit) kernel inputs will be provided by the controller and 12-bit (3x4-bit) pixel inputs will be provided by the block RAM.
3. This module will perform **multiply and accumulation** operation parallely just like in the Figure 3. Use * and + operators to calculate sum of products. Consider the size of the products and summations. The size of a product is equal to size of the multiplicand plus size of the multiplier (c=a*b -> size(c) = size(a)+size(b)). Besides, the size of a summation is equal to size of the larger one of the operands plus 1 (c=a+b -> size(c) = max(size(a),size(b)) + 1).

4. Calculation of sum of products will be a signed operation. Therefore, use **signed** keyword for the signal corresponds to final summation (necessary for comparison). The kernel value **-1** will correspond to **4'b1111** (2's complement).

5. Write down an **always block** to drive output pixels properly. The **always block** should be triggered with **pixel_clk** and **rst** signals. If the **enable** input is low, then the **pixel_out** should be given as zero. Otherwise, calculated pixel values should be driven. Besides, the pixel values should not be more 15 ($2^4 - 1$) and less than 0. If the value is more than 15, then the pixel should be mapped to 15. If the value is less than 0, then the pixel should be mapped to 0.

6. Create a testbench to simulate the convolution unit. Show that it works properly.

Block RAM

1. **Block RAM** will contain the one channel of the **input image**. The image data will be loaded with a .txt file which is generated by **input_image.m** file given at Ninova.

2. Create a block BRAM IP with the 12 rows (3 x 4-bit) and 103148 (642*482/3) columns. Load up the **"red_input.txt"** as your BRAM's initialization file. Leave unspecified settings as default.

3. This file has 12-bit values in 103148 addresses. Each row includes 3 columns of the image and each element is 4-bit pixels. Consider that Block RAM allows an interface which drives 3 pixels at 25MHz frequency.

4. The original image size is actually 640×480. It's edges were padded with zeros to obtain the output as the same size with the original input. Thus the new input image size is 642×482 and output will be 640×480.

5. Block RAM will only be used for reading data. There will be three Block RAMs in the top module for each color channel with the initialization files: red_input, green_input and blue_input.

6. Create a testbench to simulate the Block RAM. Show that it works properly.

Controller

1. Create a new Verilog module named **controller**. This module should have 1-bit input **pixel_clk**, 1-bit input **rst**, 1-bit input **enable**, 12-bit input **data_in**; 1-bit output **done**, 17-bit output **address**, 12-bit outputs **kernel1**, **kernel2**, **kernel3**, **pixel1**, **pixel2**, **pixel3**. This module should provide the **ram addresses** for Block RAM; **kernel weights** and **pixel values** for Convolution Unit.

2. This module should be designed as a **Finite State Machine**. There should be two data buffers ([3:0]buffer1[641:0], [3:0]buffer2[641:0]) which stores two previous lines (rows) of the image. Besides, an 12-bit signal should be used to store previous data coming from Block RAM.

3. The states of the Finite State Machine should be: **FIRST_LINE, SECOND_LINE, PROC1, PROC2, PROC3 and DONE**. In FIRST_LINE state, Block RAM address should be incremented and the pixels of the first row should be stored to buffer1. In FIRST_LINE state, Block RAM address should also be incremented and the pixels of the second row should be stored to buffer2. Define a 10-bit counter as an index for the buffers.

4. If enable is high, then the states **PROC1**, **PROC2** and **PROC3** should operate. In PROC1 state, 3 pixel values should be taken from the Block RAM **(data_in)** and first 3x3 receptive field (area) of the image should be given as output. 3 pixels of the first row **(output pixel1)** from the buffer1, 3 pixels of the second row from the buffer2 **(output pixel2)** and 3 pixels coming from the Block RAM **(output pixel3)**. After that, 3 pixels of the first row from the buffer1 should be replaced with the 3 pixels of the second row from the buffer2 and 3 pixels of the second row from the buffer2 should be replaced with 3 pixels coming from the Block RAM. It is necessary for the next rows of the image. Besides, **data_in** should be stored as **previous_data** for the next states.

5. In **PROC2** state, 3 new pixel values should be taken from the Block RAM. 2 pixels from the previous data, and 1 pixel from the new data should be given as output pixel3. Other outputs should be given from the buffer1 and buffer2. Buffer1 and Buffer2 should be updated for the next rows.

6. In **PROC3** state. 1 pixel from the previous data, and 2 pixels from the new data should be given as output pixel3. Other outputs should be given from the buffer1 and buffer2. Buffer1 and Buffer2 should be updated for the next rows. After that, turn back to the **PROC1** state and give the pixel values of the entire image to the Convolution Unit.

7. When the data transfer is completed. Change the state to DONE.

8. Output **kernel weights** should have fixed values. Input kernel will be 3x3 Laplacian filter:
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

9. Create a testbench to simulate the controller. Show that it works properly.

VGA Driver

1. RTL code for the VGA Driver **vga_controller.v** will be given in Ninova. The VGA Driver module has 3 outputs which are **VGA_HS** for horizontal synchronization, **VGA_VS** for vertical synchronization and **data_en**. **VGA_HS** and **VGA_VS** will be directly connected to VGA interface pins in the constraint file. **data_en** signal will be used in controller and conv_unit modules. When data_en is high, the system should operate and drive the output pixels to the VGA. Otherwise, the system should be idle and the output pixels should be assigned to zero.

2. Create a testbench for the **vga_controller.v** and show the intervals in which **HSYNC**, **VSYNC** and **data_en** signals are high.

Forming the Top Module

1. Create a new Verilog module named **top_module.v**. This module should have 1-bit input **clk** (100MHz), 1-bit input **rst**; 1-bit output **VGA_HS**, 1-bit output **VGA_VS** and 4-bit outputs **VGA_R**, **VGA_G**, **VGA_B**. **rst** signal should be connected to a button and output ports should be connected to VGA pins.

2. In the top module, generate a 25MHz clock from the 100MHz external clock for the overall image processing system.

3. When all submodule designs are completed, the system must be gathered up in a top level module as seen at Figure 6. Instantiate all the blocks seen at Figure 6.

4. Make all module connections properly.
5. Create a testbench for the **top_module.v** that writes the pixel values into a text file. Download the MATLAB script, **test_image.m** and compare the pixel values with your text file.
6. After completing and verifying the top module. Instantiate Convolution Unit, Block RAM and Controller for other channels (Green and Blue). There should be 3 instantiations for Convolution Unit, Block RAM and Controller corresponds to red, green and blue color channels.
7. Rewrite your testbench for the **top_module.v** that writes the pixel values of each channel into text files and verify your design.
8. Synthesize and implement your design.
9. Connect the VGA cable from FPGA board to the VGA monitor and show that your design works as expected.

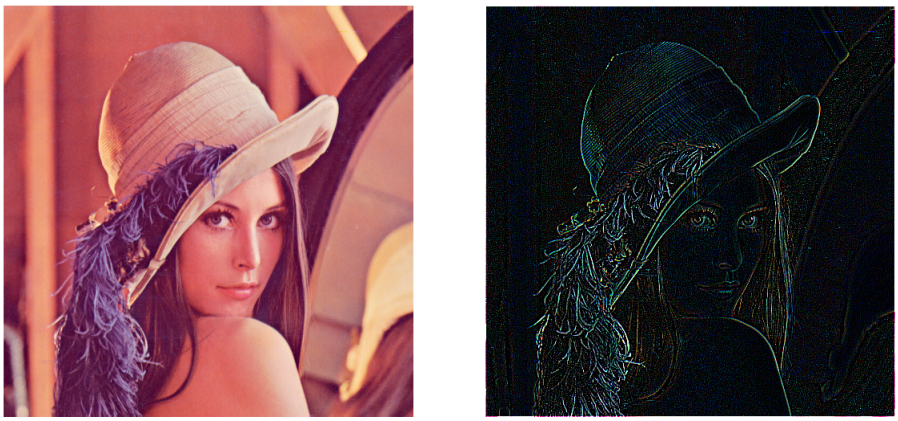

Figure 7: Edge Detection using Laplacian Filter

References

[1] R. C. Gonzalez and R. E. Woods, Digital image processing. New York: Pearson, 2018.

[2] V. A. Pedroni, Circuit Design with VHDL. The Mit Press, 2019.

[3] Stephen D. Brown and Zvonko G Vranesic, Fundamentals of Digital Logic with Verilog Design, McGraw-Hill, 2002.