# Border detection processing with FPGA

Contreras Vargas Andrés, Dzay Villanueva Luis, Llanes Pasos Diego, Huchin Matú Rodrigo, Basto Clavijo Julio and Vega Can Rafael
Autonomous University of Yucatan

***Abstract*—The objective of this project is to perform image processing capable of detecting edges through the Sobel operator, through the use of FPGA, which is responsible for carrying out the mathematical operations corresponding to the convolution. There are four points that will be discussed, speed, integration, modules and results.**

## I. INTRODUCTION

The project consists of an image processing block, the input signal consists of a 3x3 matrix with the value of the pixels surrounding a specific pixel of a grayscale image, the block is capable of convolving with a 3x3 Kernel of the Sobel operator, the operation of the project was developed in order to execute it on a Raspberry 1 with the software OCTAVE and its serial communication functions, OCTAVE is responsible for reading, communicating and reading final data; The FPGA on the other hand was developed in a NEXYS 3 card to which the modules programmed in Verilog were integrated with the following functions, communication via UART, convolution and image scaling. It is a sequential model which can only finish processing once all the pixels in the image were recovered. There is four

## II. PROCESSING SPEED

### Part 1. Implementation in computer

The first stage of the project was to program the algorithm in OCTAVE, this in order to visualize the operations executed in the process, which, as can be seen in Figure 1, is the convolution operation, executes a total of 9 sums and 9 multiplications, this for each pixel of the image except the edges, which were not taken into account in this project.

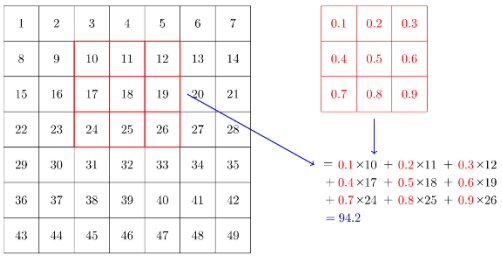


*Figure 1. Convolution process*

This process scans the image, obtaining the values of 9 pixels, the computation time in a computer is not considerable nor the time over which the comparison will be made since the application is to take it to an independent personal computer system, the compute time it's about ten seconds.

### Part 2. Implementation in Raspberry pi

*Figure 2. Raspberry pi 1*

In order to migrate the operations to an embedded system, the Raspberry pi 1 microprocessor was chosen, in which the OCTAVE software script was run, the computation time was considerably slower, a process about an hour and a half, they were analyzed the sections of code in which the process took longer, the reading of the image did not present a delay. Instead, the convolution section consists of 9 sums and 9 multiplications, which is why it is considered a problem in terms of computation time.

```
for i=2:m-1
    for j=2:n-1

     y(i,j)= b4(i-1,j-1)*kernel1(1,1)+b4(i,j-1)*kernel1(2,1) + b4(i+1,j-1)*kernel1(3,1)+...
             b4(i-1,j)*kernel1(1,2)+b3(i,j)*kernel1(2,2)+ b4(i+1,j)*kernel1(3,2)+...
             b4(i-1,j+1)*kernel1(1,3)+b4(i,j+1)*kernel1(2,3)+ b4(i+1,j+1)*kernel1(3,3);
    end
end
```

*Figure 3. Convolution code*

### Part 3. Implementation in FPGA and Raspberry

As a proposal for improvement, in terms of computing time, it was decided to use the FPGA to carry out the operations by means of an accumulator adder, whose process consists of collecting all the values corresponding to 9 pixels, multiplying them by the Sobel kernel and then adding all the values, so get the value corresponding to the pixel to evaluate.

CPUs perform operations in sequence, so the first operation must run on the entire image (3x3 pixel) before the second one can start. In this example, assume that each step in the algorithm takes 6 ms to run on the CPU; therefore, the total processing time is 24 ms. Now consider the same algorithm running on the FPGA.

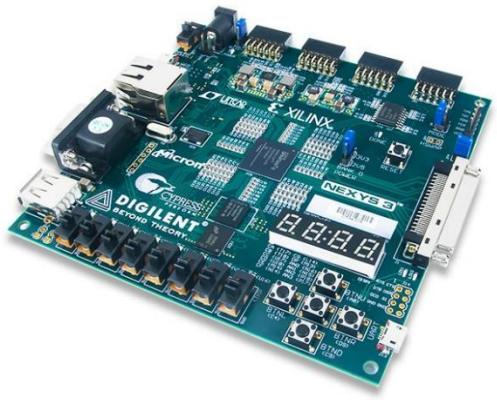

*Figure 4. Nexys 3 FPGA*

Since FPGAs are massively parallel in nature, each of the four operations in this algorithm can operate on different pixels in the image at the same time. This means the amount of time to receive the first processed pixel is just 2 ms and the amount of time to process the entire image is 4 ms, which results in a total processing time of 6 ms. This is significantly faster than the CPU implementation. Even if you use an FPGA co-processing architecture and transfer the image to and from the CPU, the overall processing time including the transfer time is still much shorter than using the CPU alone.

## III. INTEGRATION

Once the project was to be carried out with Raspberry and FPGA, the different communication protocols were evaluated, the Nexys 3 contains an FT232 module that allows serial communication, taking advantage of the fact that the OCTAVE has a special command package for serial port communication.

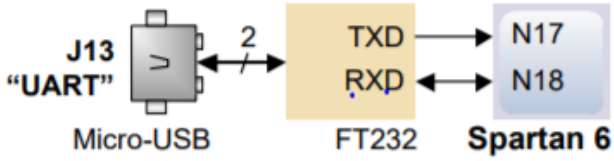


*Figure 5. UART pinout*

In order to remove CPU computing time, the mathematical operations of the convolution were removed to the FPGA, so that only the image size is needed to know the number of times the serial communicates data to the FPGA.

```
clc;clear;
s1 = serial("\\\\.\\COM10");
srl_flush(s1);
set(s1, "baudrate", 115200) # Change baudrate
set(s1, "bytesize", 8)    # Change byte size (config becomes 5-N-1)
set(s1, "parity", "N")    # Changes parity checking (config becomes 5-E-1),
set(s1, "stopbits", 2)    # Changes stop bits (config becomes 5-E-2), possible

b2=imread('Lena.jpg');
b1=rgb2gray(b2);
b2=bitand(254,b1);
b4=bitshift(b2,-1);
[n,m]=size(b1);
y=zeros(n,m);
for i=2:n-1
    for j=2:m-1
      for k=i-1:i+1
        for p=j-1:j+1
        srl_write(s1,uint8(b4(k,p)));
        end
      end
    y(i,j)=srl_read(s1,1);
  end
end
y1=y./max(max(y));
y2=uint8(y1*255);
y3=y2-((mean(mean(y2)))/2);
imshow(y3);
fclose(s1);
```

*Figure 6. OCTAVE code*

For every pixel in the image, nine bytes are sent, that is, 9 pixels; One issue to consider was the format to which the values reach the FPGA, since the eighth only allows to send values of type UINT8, for this, using a MATLAB tool, the function of the convolution was evaluated to determine an 8-bit data , how much of that information is composed of integers and floats. The result was four integers and floating, since considering that the values of an image are handled in floating point, the values of the image are divided between 255, being in a new range of 0 to 1; considering this and the matrix of the sobel operator used the maximum and minimum value that the image can adopt is (4, -4).

| | | |
|---|---|---|
| -1 | 0 | +1 |
| -2 | 0 | +2 |
| -1 | 0 | +1 |

Gx

| | | |
|---|---|---|
| +1 | +2 | +1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

Gy

*Figure 7 Operator sobel*

In the code shown in Figure 7, it is shown how the values were adapted to the new range; in previous tests it was carried out considering the input value to the FPGA, such as four integers and floats, which resulted in a range of 255 values normalized in a new one of sixteen different values, so that it was considered in a second stage to use only one integer at the entrance and seven floating, considering the output as the original four integers and floating. With this in consideration, the adjustment was made with an AND gate, and subsequently performing a one-bit shift; having a new real range of (.9921875-0).

## IV. MODULES

The development in FPGA, consists of eight modules, with code each, the schematic was used to connect the operation of all.

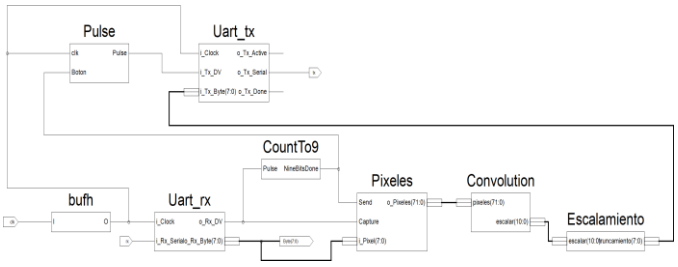


*Figure 8. FPGA Schematic*

In order to use only one clock, we use the resource BUFH in figure 9, the horizontal clock buffer (BUFH/BUFHCE) allows access to the global clock lines in a single clock region through the horizontal clock row.

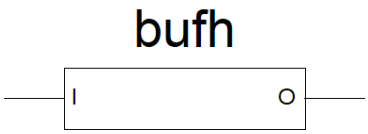


*Figure 9. BUFH*

It can also be used as a clock enable circuit (BUFHCE) to independently enable or disable clocks that span a single clock region.

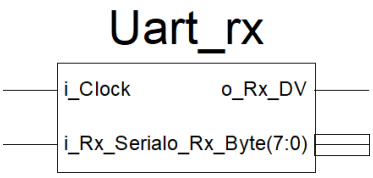


*Figure 10. UART RX module.*

In this module the data is received via serial, this module has two inputs, which are the clock and the input bytes; This is where the Baud rate is set, in the code shown below it was parameterized for a 115200 baud rate, since the 100 MHz clock is being used.

$$CLKS\ PER\ BIT = \frac{100Mhz}{115200} = 868$$

```
module Uart_rx
  #(parameter CLKS_PER_BIT=868)
  (
   input        i_Clock,
   input        i_Rx_Serial,
   output       o_Rx_DV,
   output [7:0] o_Rx_Byte
   );
```

The RX Dv output is connected to the Count 9 module, this output emits a pulse every time a byte is received, therefore it works as an indicator that the 9-byte packet was delivered.

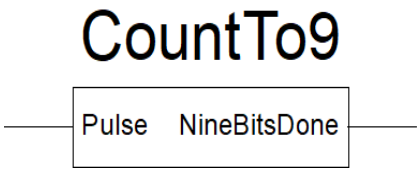


*Figure 11 CountTo9 module*

```
module CountTo9(
input Pulse,
output reg NineBitsDone = 0
    );

reg [3:0] count = 0;

always@(posedge Pulse)
   begin
      if(count<8)
         begin
            NineBitsDone <= 0;
            count <= count + 1;
         end
      else
         begin
            NineBitsDone <= 1;
            count <= 0;
         end
   end

endmodule
```

This module is a zero to nine counter, which is reset when it reaches nine, it serves to send a ready signal to the Tx module and as a signal to execute a shift in the bytes that reach the pixel module.

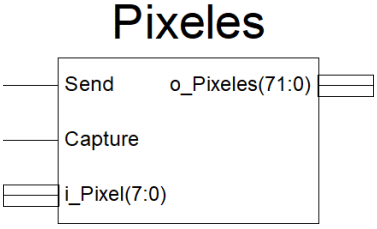


*Figure 12 Pixeles module*

```
module Pixeles(
input Send,
input Capture,
input [7:0] i_Pixel,
output reg [71:0] o_Pixeles = 0
    );

reg [71:0] r_Pixeles = 0;

always@(posedge Capture)
   begin
      r_Pixeles <= {i_Pixel, r_Pixeles[71:8]};
   end

always@(posedge Send)
   begin
      o_Pixeles[71:0] <= r_Pixeles[71:0];
   end


endmodule
```

This module works as a shift register, has 3 inputs, the "Send" input works as a trigger when the nine pixels are present, so that when the signal from CountTo9 is activated, the output register must be full, the "Capture" entry receives all the newly received values of the Rx pin, as can be seen in the code each time an entry is recorded in the "Capture" pin, the information is added in the pixel output variable, completing a total 72 bits in total.

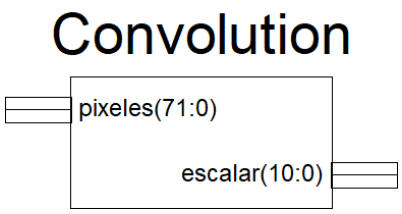


*Figure 13 Convolution module*

```
module Convolution(
input [71:0] pixeles,
output [10:0] escalar
    );
wire signed [21:0] escalar2;
wire signed [10:0] escalar3;
wire signed [10:0] escalar;
wire signed [10:0] elemento1 = {3'b000, pixeles[71:64]};
wire signed [10:0] elemento2 = {3'b000, pixeles[63:56]};
wire signed [10:0] elemento3 = {3'b000, pixeles[55:48]};
wire signed [10:0] elemento4 = {3'b000, pixeles[47:40]};
wire signed [10:0] elemento5 = {3'b000, pixeles[39:32]};
wire signed [10:0] elemento6 = {3'b000, pixeles[31:24]};
wire signed [10:0] elemento7 = {3'b000, pixeles[23:16]};
wire signed [10:0] elemento8 = {3'b000, pixeles[15:8]};
wire signed [10:0] elemento9 = {3'b000, pixeles[7:0]};
wire signed [21:0] multi1 = elemento1 * (11'b00010000000);
wire signed [21:0] multi2 = elemento2 * (11'b00000000000);
wire signed [21:0] multi3 = elemento3 * (11'b00010000000);
wire signed [21:0] multi4 = elemento4 * (11'b00100000000);
wire signed [21:0] multi5 = elemento5 * (11'b00000000000);
wire signed [21:0] multi6 = elemento6 * (11'b00100000000);
wire signed [21:0] multi7 = elemento7 * (11'b00010000000);
wire signed [21:0] multi8 = elemento8 * (11'b00000000000);
wire signed [21:0] multi9 = elemento9 * (11'b00010000000);
wire signed [21:0] multii3 = multi3 * (-1);
wire signed [21:0] multii6 = multi6 * (-1);
wire signed [21:0] multii9 = multi9 * (-1);
assign escalar2 = multi1+multi2+multii3+multi4+multi5+multii6+multi7+multi8+multii9;
assign escalar3 = escalar2[17:7];
assign escalar = escalar3;
endmodule
```

As we can see in the convolution code, this is the module responsible for performing the operations relevant to the image processing, its input parameter is a 72-bit register corresponding to the 9 pixels of the image, multiplication is performed with the Sobel operator with the respective complement to two in the operations with negative sign, the sums are made and the result is the value of the pixel that will be sent to the OCTAVE, the data is trimmed to obtain a value of 8 bits.

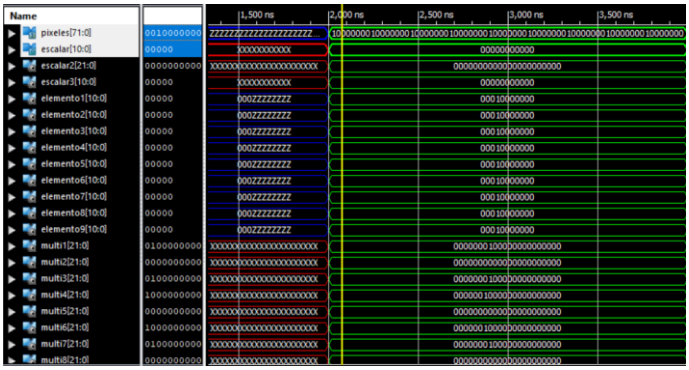


Figure 14 Testbench convolution

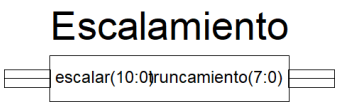


*Figure 15 Escalamiento module*

The scaling module is based on the formula presented in Figure 16, where once the values are within a maximum range of [4, -4] they need to be sent via serial to the Raspberry, as mentioned above, only values corresponding to unsigned integers can be used, therefore a sum of 4 is contemplated to all results to avoid having negative results.

$$I_N = (I - \text{Min})\frac{255}{\text{Max} - \text{Min}}$$

*Figure 16 equation to normalize*

The pulse module is the one in charge of the start bit, its input signal comes from the CountTo9 module, which each 9 bytes are received gives a pulse. Pulse module generates a rising edge that functions as input in the TX module

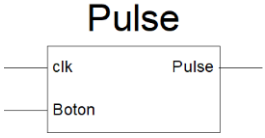


*Figure 16 equation to normalize*

The last module of the schematic is the UART TX, its input comes from the "Scaling" module which has the processed pixel value corresponding to the image, its output is that corresponding to the FT232 component that allows serial communication.

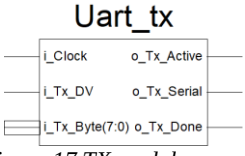


*Figure 17 TX module*

## V. RESULTS

*Figure 18 left: Input image right: output image*

The improvement is related to the computing time, using only the resources of the Raspberry, the processing took about an hour and a half to finish, this given that since its nature is sequential, 9 sums and multiplications were carried out online, unlike of the FPGA that can perform all nine sums at the same time. By integrating the FPGA to the Raspberry, the computing time decreased to 6 minutes for a 130 x 130 black and white image.

## REFERENCES

Viewpoint Systems. (2019). *FPGA Basics for Industrial Applications*. IEEE Communications Spectrum