Human Detection by Histogram of Oriented Gradients for Automatic Braking in Autonomous Cars

Christoporus Deo Putratama (13213008)  
Kevin Shidqi Prakoso (13213065)  
Bramantio Yuwono (13213128)

Date: 27/12/2016

EL4138- VLSI System Engineering

SEEI - ITB

1. Introduction

Recently, self-driving cars have been a trending topic in the development of the automotive industry. However, there have been several unfortunate events regarding the development of autonomous cars. There have been several accidents, and these events have had a negative effect on the public opinion. Therefore, more safety measures are in order if the autonomous car industry is to have a chance of thriving. One of the main concerns is aimed at the ability of the car to properly detect if an obstruction in ahead, and do the necessary manuevers to ensure minimum damage. The most pressing matter in this regard is of course, the ability to detect humans.

Human detection has been an important topic in image processing, and while many techniques have been developed, few have been able to produce results that are satisfactroy in terms of accuracy and speed. Naturally, the human detection process will need to be fast enough if it is to be effectively implemented in automatic braking. However, accurate algorithms usually dissapoint in terms of speed, and algorithms for automatic braking need to both accurate and fast, with perhaps a little more emphasis on the latter. One of the possible solutions is to develop a hardware-based system, in order to both have accuracy and speed.

  In this report, we aim to develop a circuit that implements a human detection algorithm by means of histograms of oriented gradients (HOG). HOG is a relatively new feature descriptor, introduced in 2005 by Dalal and Triggs [1]. This method is similar to edge orientation histograms, but this method differs in that it is computed on a grid of uniformly spaced cells which are consisted of pixels and uses local overlapping contrast normalization for accuracy.

The rest of this report is organized as follows. Section II explains fundamentals of the circuit and architecture, section III shows reduction techniques used in the circuit for efficient implementation of this algorithm as well as several other appeals, section IV explains the simulation results, while section V will give an overall evaluation of the implementation in an Altera DE2-115 FPGA.

1. Architecture description
   1. Overall architecture

The histograms of oriented gradients uses local histograms of oriented gradients of pixel luminance to characterize a given image. HOG description of an image roughly gives its structure and shape, enabling us to perform detection algorithms to rcognize human shapes. The process of HOG feature extraction is as follows:

1. Luminance is calculated from the color image, and the gradient strengths and angles are calculated from the luminance map.
2. Histograms of oriented gradients are calculated by summarizing the gradient magnitude and orientation for each group of pixels.
3. The histograms are normalized, and used to determine whether or not there is human in the image using SVM classifier.

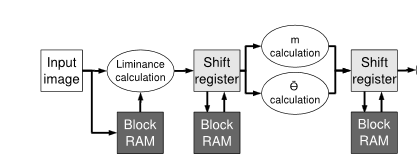


Figure 1: First half of architecture

Therefore, the design for the system is as follows. First, the input image needs to be converted to a luminance map, or in other words, converted to grayscale. Grayscaling is done by multiplying each color component of a pixel by a fraction, and then summing all the results. The final number is the luminance value of a pixel. To achieve this purpose, we use a simple adder combined with a constant multiplier consisting of shifters. The image is stored in a RAM block, as is the resulting grayscale image.

Second, we need to calculate the luminance gradient of the image, in directions x and y. The computations are done per block. Luminance gradients are easily computed by finding the difference of luminance between neighboring pixels, which is achieved using a simple subtractor. The luminance gradients are then used to calculate the gradient strengths and orientations. The results are used to fill the histogram of that particular block, with each pair of luminance gradients adding to a certain bar on the histogram depending on their orientation, by a value depending on their magnitude.

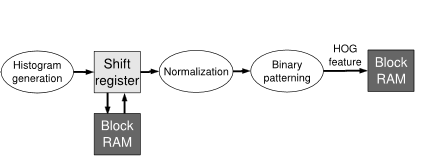


Figure 2: Second half of architecture

Third, the resulting histogram needs to be normalized and patterned. Histogram normalization causes the histogram to be as such, so that if all of its elements are summed, will produce unity as a result. Patterning involves combining several histograms from neighboring blocks. The resulting histogram is then saved in a RAM block, to be checked whether or not the histogram represents an image of a human.

Fourth, the detection process takes place. This involves multiplying a weighing matrix with the histogram, and adding it with a bias number. Systolic arrays are used to multiply matrices efficiently, and the resulting number determines whether or not a human is detected.If a human is indeed detected, a red box is drawn around the block which the histogram represents. As shown in the figures above, the process is fully pipelined.

* 1. NIOS II Processor and RAM

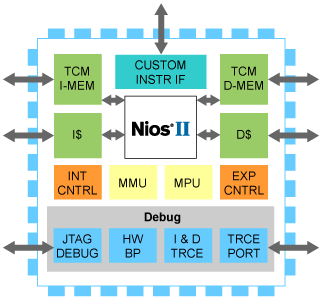


Figure 3: NIOS II Overview

The system is implemented on an Altera DE2-115 FPGA, which include 2MB of SRAM used in the architecture. To build the necessary components, the Altera Qsys is used. The SOPC built is as follows.

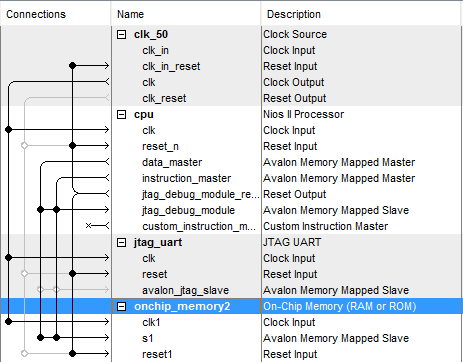


Figure 4: SOPC Overview

As a processor, NIOS II is too slow for our purpose, so its only responsibility is to manage RAM access and receive commands via the JTAG port, which enables switching of images, while all of the calculations are done on separate modules. The 2MB SRAM is more than enough to store the image, its grayscale counterpart, and any resulting luminance gradients as well as histograms. Several constants are also stored on the RAM, such as the weighing matrix.

* 1. Preprocessing

HOG is a feature descriptor to represents shapes, and as such, it ignores variations in color, which means an image needs to be converted to grayscale. A color image has three matrices, consists of R, G, and B matrix, so we must combine each of those matrices into a single matrix by using the following equation.

Y = 0.298912R + 0.586611G + 0.114478B

The resulting number is the luminance of a pixel, with a higher number representing a lighter shade of gray.

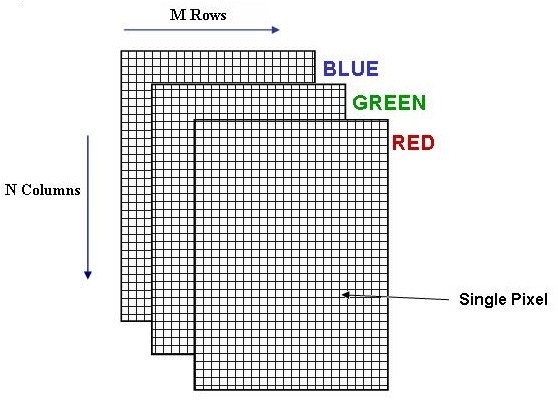


Figure 5:Representation of matrices from a color image

The implementation from this greyscaling process is to change all the three matrices into a 8-bit signal by the aforementioned equation. This involves multiplying and adding each component. However, because the multiplication factors are constant, the multiplication operation can be replaced by shifters, as follows.

0.58 is approximated by 0.100101, needing 3 shifters.

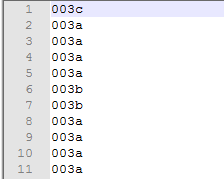
0.29 is approximated by 0.0100101 , needing 3 shifters.

0.11 is approximated by 0.000111 , needing 3 shifters.

The resulting grayscale image is then stored on the RAM. To speed up the process, we execute grayscaling in parallel with 120 modules.

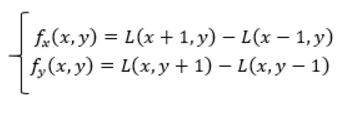


A sample of greyscaled image



Sample of raw image represented in hex numbers.

Gradient calculation is represented by the following expression:



Equation used to calculate gradients

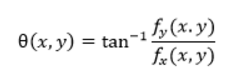
From the grayscaling process, the resulting image is then scanned to get the gradien from every pixel of the image. The gradient calculating process involves simple subtraction between neighbouring pixels. As before, calculation is done in parallel using 120 module, with each module continuing the work from a specific grayscaling module. As before, the results are then stored to the SRAM.

* 1. HOG Calculation

After getting *dx* and *dy*, we can calculate the magnitudes of the gradients by using following expression:



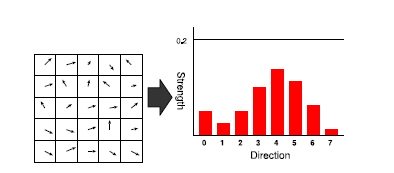
While the orientation of the gradients are calculated using the following formula



Equation used to calculate angle of gradien

However, implementing calculations such as square roots and trigonometric functions are not pratical in an FPGA. However, the exact value of the arctangent is not entirely necessary, because blocks in histograms represent a range of angles, not an exact number. Therefore, we approximate the arctangent calculation using shifter-multiplier and comparators. As for the magnitude calculations, approximations have been tried with dissapointing results. Therefore, we devised a new algorithm to calculate magnitudes efficiently. The details of the calculation are discussed in the following chapter.

Then the gradient strengths m are inserted into the histogram according to the corresponding quantized orientation label for every luminance value in a cell to make the histograms, as shown in the figure below.

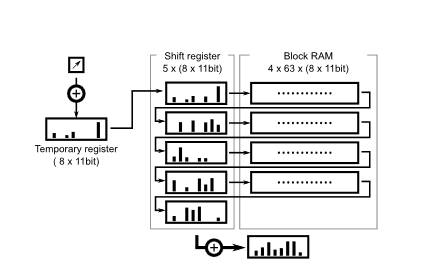


For example, when an angle of a certain pair of dx and dy is between Q1 and Q2, then the corresponding label becomes 2, and its magnitude is added onto bar with label 2. We take the range of angles from -π/2 to π/2, since HOG does not focus on gradient direction but orientation. The threshold values are as follows:



A histogram is generated for each cell, which contains 8x8 luminance values. Since we quantize the gradient orientation into eight labels, 8-dimension feature vectors are eventually generated. For a (w × h) input image, a total of w/8 × h/8 histograms are obtained since each cell consists of 8×8 luminance values.

Since quantizations have been made in the calculation steps of gradient orientations θd as mentioned in the previous parts, any other quantization process is not necessary in the histogram generation process. We have 640×480 values of luminance gradients(one for each pixel), which gives us a two-dimentional array with the width of 640/8 = 80 cells and the height of 480/8 = 60 cells. Since the maximum value of a gradient strength is 361, then the maximum size is 23,104 for a histogram of a single cell, data size for one cell becomes 14 bits ×8. Therefore, total histograms of luminance gradient for the entire single image are expressed with (14×8)×(80×60) = 537,600 bits.



We also take a stream process approach for histogram generation as shown in Figure . A specific histogram for a cell is obtained from a matirx of 64 (8×8) values of luminance gradients(both m and θd). A partial histogram of gradient orientations for 8 consecutive luminance values is made every 8 luminance values. Then the stream of the partial histograms goes through Block RAM line buffers so that partial histograms for eight lines are eventually summed up to make the full histogram for the cell.

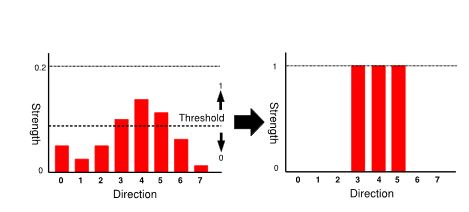
Since the maximum value for the gradient strength is 361, each orientation of a partial histogram of five luminance values can be expressed with 12 bits. Thus, the required capacity of the Block RAM buffers corresponds to 12 bits×8 orientations×63 cells×4 lines. Since the luminance values are streamedin every two clock cycles, full histograms are streamed out every 16 clock cycles.

* 1. Normalization and Binarization

The normalization process is carried out every block, which consists of 2x2 cells. With every histogram consisting of 9 parts, we have a two-dimensional array with a size of 36x36. Normalization invovles summing all 9 elements of a histogram, summing it with elements of other histograms, and calculating the root mean square value of the histograms. With a picture size of 640x480, we have a total of 78 x 58 x 72 dimensions from a single image data. Because of this complexity, as before we employ a simplification method to calculate normalization, as will be described in the next chapter. This process involves a maximum of three shifts.

Normalization will also use the streamed structure, similar to the one used for histogram generation. The process is carried out for a moving 2x2 window of cell histograms. For this process, three lines of 2-stage shift registers, along with 2 lines of 61-stage RAM blocks. Every time a new histogram is generated, 72 dimensions of histogram elements in the 2x2 cell are summed in two clock cycles. The next process, as mentioned, requires three shifts, and one conditional check, resulting in two cycles. Thus, the total amount of clock cycles required by the normalization process is 4 cycles.

The next part, binarization, strives to reduce the amount of memory required by the next processes. If we express the HOG as a 8-bit double precision floating point, it would require 78 x 58 x 72x 8-byte memory capacity, which is more than the SRAM can handle. In our scheme, each one dimension feature will be binarized using a threshold value so that every gradient can be represented as a single bit. Since the gradient orientations are quantized into 8 orientations, a feature amount obtained from a certain cell can be described by eight bits (or 0 to 255). An example can be seen in the figure below. Using this scheme, a 1/64 reduction can be achieved. However, this scheme will also reduce performance for several pictures, as will be shown in the last chapter.



* 1. Support Vectoring Machine

A support vectoring machine can be used when the data has exactly two classes. An SVM classifies the data by locating the best plane that separates the data points of one class from the other class. The *best* plane, in this sense, for an SVM is the one with the largest *margin* or euclidean distance between the two classes. Here, margin means the maximum width of the slab which is parallel to the plane that has no data points inside it. In other words, the best plane is the one whose distance from the nearest data point is the largest.

In this implementation, the SVM is represented by a weighing matrix and a biasing number. The histograms are multiplied with the weighing matrix, resulting in a single number, which is then added with the biasing number. If the result is a positive number, then the conclusion is that a human is detected.

As the SVM from the LSI-Contest website still has unpredictable flaws, we decided to train our own SVM classifier using two different datasets, namely MIT pedestrian database and INRIA database. A boosting algorithm is used with the SVM serving as the ‘weak’ classifier, based on [4]. The result of the training process is a new weighing kernel and a new bias number. These new components prove more effective at detecting humans in images outside of the one provided by the website.

* 1. VGA and Display Module

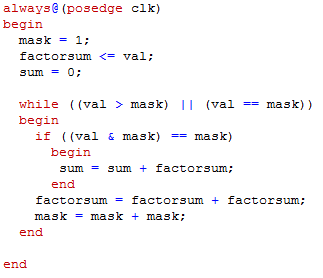
To display the image, we use a LG® Flatron L1742S monitor, using the ram as the screen buffer. The ram block in question is the original picture, which will be edited at the end of the detection process, should a human be detected. The editing process involves drawing a red box along the edges of the window being processed. This is achieved by simply changing the value of all the pixels at the edge of the box by a RGB value of FF,00,00. This means changing the value of the RAM block corresponding to the pixels at the edge of the box.

The Altera DE2-115 is equipped with a ADV7123 video DAC. Therefore, the ports that we must take into consideration are the Hsync, Vsync, and the RGB values, each with its own port.

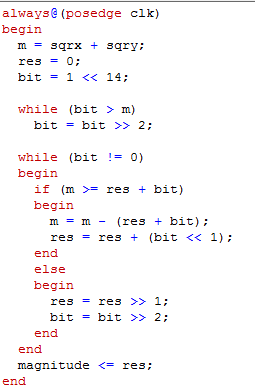
1. Simplification of the Calculation Process and Other Appeals

### 3.1 Magnitude Calculation

So, we will made an approximation to get these values of magnitudes and angles.



Verilog Algorithm to approximate a multiplier operation



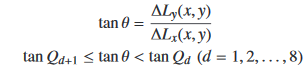
Verilog algorithm to approximate a root operation

### *3.2 Angle Calculation*

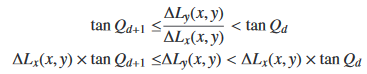
Angle calculation involves trigonometric functions such as the arctangent, which is fairly complicated to implement in an FPGA. One alternative is the CORDIC algorithm, however it takes many iterations to produce an accurate result. Combinatorial CORDIC circuits that can output a result in a single clock cycle exists, but they consume a lot of logic elements.

However, an accurate result is not needed because the histogram quantizes the angles into 9 ranges. Therefore, it is only needed to find out to which range the gradient belongs. Using another method, angle calculation can be achieved in two cycle using only shifters and conditional checks.

The method is as follows. The tangent values of the quantization threshold angles Qd(d = 1, 2, . . . , 8), can be precomputed since they are constant values. Instead of calculating the arctangent, we represent the tangent as dy/dx. Therefore, if the value of tan is known, we get a quantized gradient orientation by the conditional expression with tan Qd and tan Qd+1 as shown in the equations below.



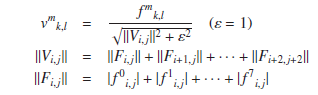
However, by multiplying dx with all the constant tangents, and by inserting the tan equation into the conditional statement, we get the following result.



Therefore, we need only to multiply dx with the constant tangent values which are precalculated, which can be done using shifters. Next, we need to evaluate the conditional statements to determine which bar in the histogram does the pixel in question belong to. Therefore, the angle calculation can be done in two clock cycles.

### 3.3 Normalization

The normalization process as used by MATLAB® and described in the equations below needs the calculation of a square root and division by fractional numbers, making FPGA implementation difficult.



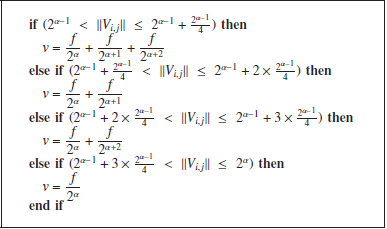
Therefore, we take an approximation approach as follows, which expands the method proposed in [5].If the denominator of the equation above (Sqr(||Vi, j||2 + ε2)) is approximated by 2α as 2α−1 <Sqr(||Vi, j||2 + ε2) ≤ 2α, the division for the normalization can be replaced by a shift operation. However, naive approximation to the nearest power-of-two value increases the normalization error. To mitigate the approximation error, we divided the interval between 2α−1 and 2α into four sub-intervals.

The range of 2α−1 to 2α can be divided into four intervals; (2α−1, 2α−1 + 2α−14 ], (2α−1 + (2^α−1)/4 , 2α−1 + 2 × 2^(α−1)/4 ], (2α−1 + 2 ×2^(α−1)/4 , 2α−1 + 3 × 2^(α−1)/4 ], and (2α−1 + 3 × 2^(α−1)/4 , 2α]. Here, ε = 1 and this is significantly smaller than ||Vi, j||2. Therefore, the conditional statements in the normalization equation can be derived when it is approximated that:

Sqr( ||Vi, j||^2 + ε^2) =||Vi, j||.

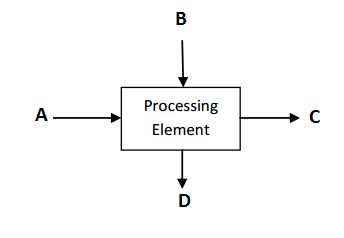
The figure below shows comparison results of approximation errors for our quadrisection approach and the naive power-of-two approach, in the case of f m k,l = 361. The results show that the normalization errors are effectively reduced with the relatively simple calculation process.

Since the maximum value of ||Vi, j|| is 81,225, the maximum number of shift operations for the division is 19. Thus, as a fraction part, 19 bits of 0s are appended to the LSB side of f m k,l in advance of shifting, and obtained vm k,l is also expressed with a fixed point arithmetic number with 14 + 19 = 33 bits.

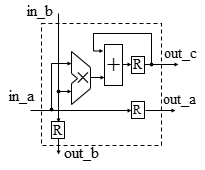


### 3.4 Matrix Calculation

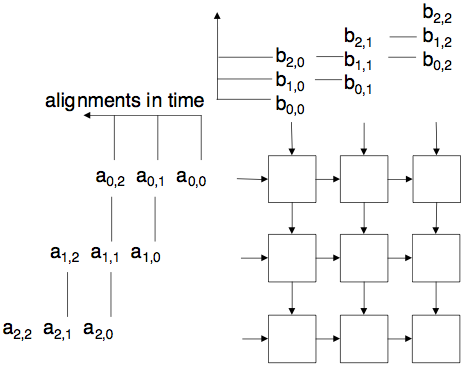
Matrix calculation is required for the SVM detection and the mask with magnitude calculation. To make matrix calculation more efficient, systolic arrays are used. Systolic architecture consists mainly of an two-dimentional array of uniform processing elements, in which data flows between the neighboring processing elements, synchronously, from four different directions. Processing elements take data from Top and Left and output the results to Right and Bottom.

[](http://1.bp.blogspot.com/-Vb9RAHN1sV8/VbulUV-2zBI/AAAAAAAAAu8/SRXsADo8ZCI/s1600/processing+element.PNG)

Here, each processing element, consisting of a multiplier and an adder, performs four operations, namely fetching, multiplication, shifting & addition**.** As the following figure shows, "in\_a", "in\_b" are inputs to the processing element and "out\_a", "out\_b" are outputs to the processing element. "out\_c" is to get the output result of each processing element, connected directly to the output of the module.

[](http://3.bp.blogspot.com/-gOyEfuAfNHk/Vbxt05BG_YI/AAAAAAAAAvQ/MRCKX4ULZzg/s1600/pe.PNG)

The processing elements are arranged in the form of a matrix. In this case we use, multiplication of 3x3 matrices, because by examining the weighing matrix, we find that there is a pattern in the array, in which 3 zeros are followed by 3 numbers. The following figure depicts how matrix A and B are fed into PE(processing element) array.

[](http://2.bp.blogspot.com/-XvlZz8VB6lE/VbyeTqj1TeI/AAAAAAAAAvg/oZysEUhCfO8/s1600/PE+array.png)

### 3.5 Scaling

Scaling is needed to detect humans with different sizes in images. The number of scaling processes depends on the size of the image. By following the algorithm in the MATLAB® file,we compare the image to a 134 x 70 pixels image and then we scale it up by 1.25n each time, until it reaches the full image size.

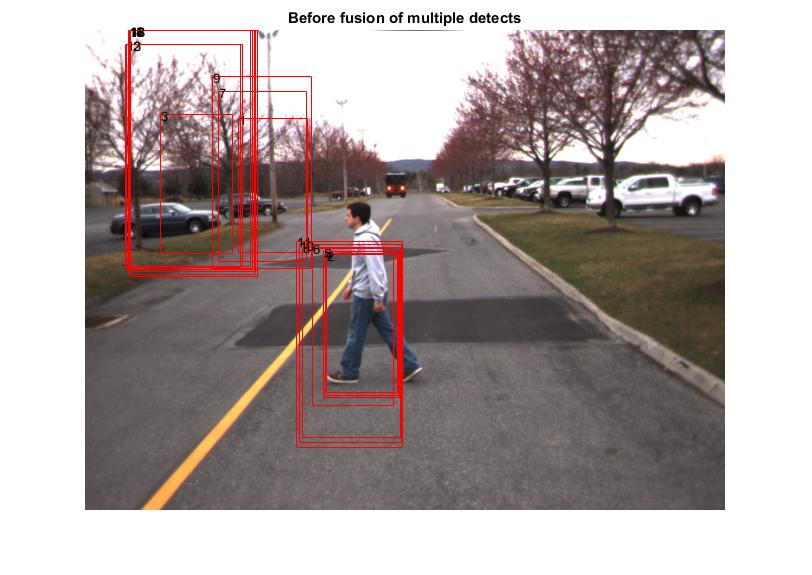
After we get the scale number, we will iterate the process of human detection up to n times. With each of iteration, the scaled image will have it’s matrix rescaled too. The new matrix’s index number are multiplier of the scaler itself.

The implementation of resizing the input image is similar with MATLAB algorithm reference provided by the lsi contest website, with some change such as in the Scaler module, instead using bilinear interpolation , we use iteration of scaling loop.

|  |
| --- |
| function pict = Scaler(img, scale)  [rows, cols ] = size(img);  row1 = floor(rows/scale);  col1 = floor(cols/scale);  pict= zeros(row1,col1,'uint8');    for i=1 : row1  for j=1 : col1  a = floor(scale\*i);  b = floor(scale\*j);  pict(i,j) = img(a,b);  end  end    end |

### 3.6 Adaboost

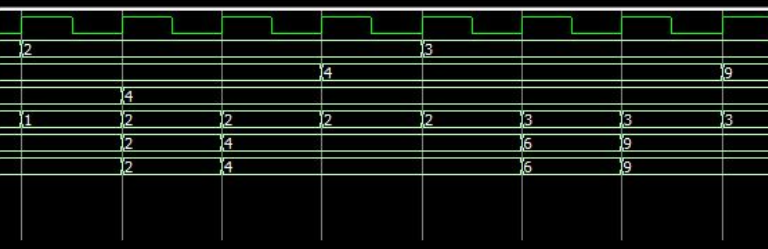
From experimenting with different images, we have found that the SVM provided by lsi contest website still has several flaws, as several pictures show that the SVM either makes a false positive or does not detect humans in a picture. An example is provided below.



Therefore, a new SVM was trained, using both the INRIA database and the MIT pedestrian database. The new SVM is a “weakened” version of the SVM, but we also used boosting algorithms to improve performance of the SVM. the size of the weights assigned by Adaboost to the weak SVMs, serve as an indication of which data points are likely to become support vectors in the final model, and hence can be useful in the implementation of editing algorithms.

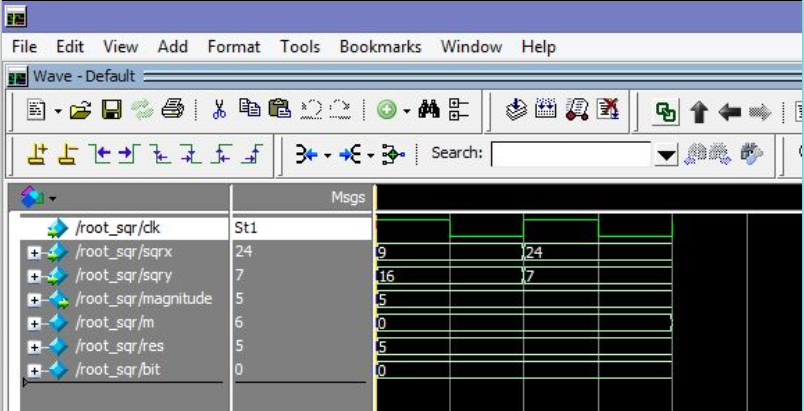
# 4. Simulation Results and Analysis

## 4.1 Magnitude Calculation



Square

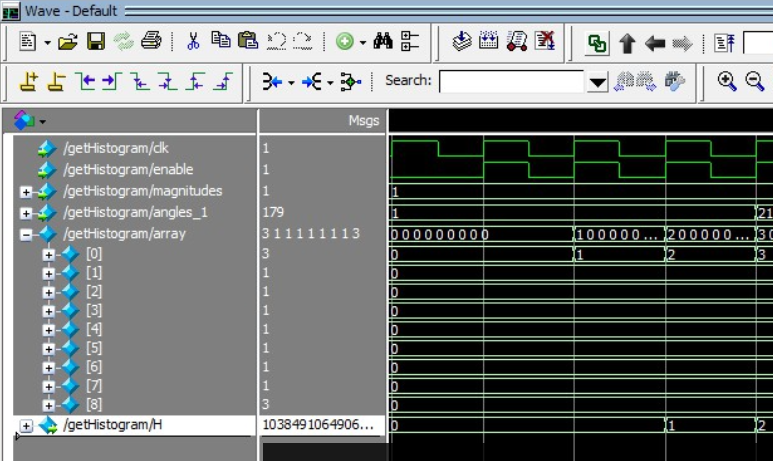
The simulation above shows the circuit for calculating the square of a number. The first row is the clock, while the second is the input number, while the third row is the result. As seen, a square of a number can be obtained in two clock cycles.



Square Root

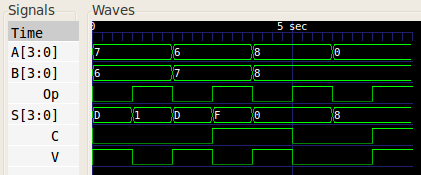
The simulation above shows the circuit for calculating the magnitude of a number. This circuit does the addition and the root operation. As seen, when we input 9 and 16, we obtain a result of 5.

## 4.2 Histogram Calculation



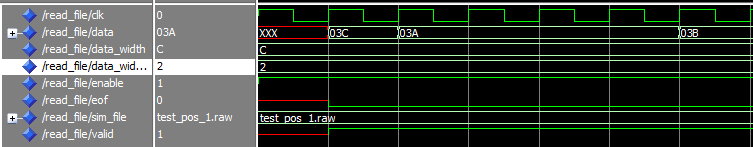
The function of this circuit is to generate a histogram based on the calculation of magnitude and angles. The histogram is made into an array of 9 numbers, which are then added by new inputs each clock cycle.

## 4.3 Preprocessing



Dx and Dy calculation

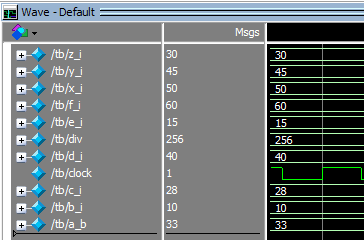
This circuit calculates the difference between neighboring pixels and creates a new array with the same size as the previous picture. As seen, this circuit is a simple subtractor, and outputs the result in a single clock cycle.



Signal created from grayscaling.

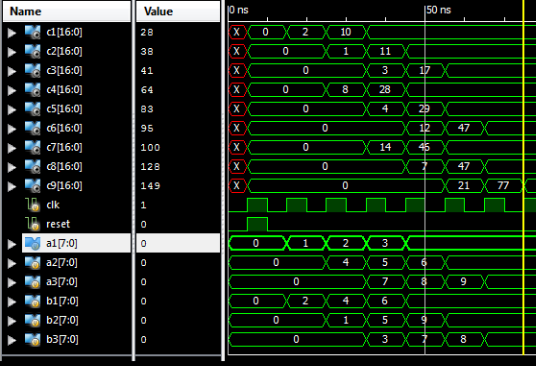
This circuit performs grayscaling. It takes the different components of a color picture, namely R,G, and B, and computes the luminance. It outputs a new luminance value every single clock cycle.

## 4.4 Histogram Normalization



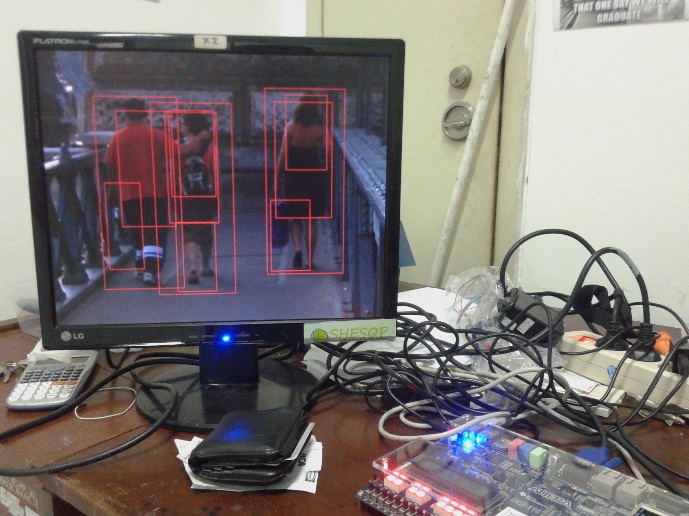
This module computes the necessary number to perform histogram equalization from the method described above. When all the inputs are summed, they produce a number around 300, which is closest to 256. The actual division is then done by shifters.

## 4.5 Matrix Multiplication



This circuit implements the systolic matrix multiplication mentioned before. Output results can be seen after6 clock periods from the time of data insertion. (time period between a1 gets 1 and the time of yellow marker). In usual case for 3x3 matrix multiplication altogether, it takes 3x3x3 = 27 times to do the iterations and calculations. But in this case it takes lesser time (x6 times), because systolic architecture is a class of parallel pipe-lined architecture.

## 4.6 Overall





The pictures above show the result of the overall implementation of our system. Two pictures are used, with the first one provided by the lsi-contest website, while the second one is from the MIT pedestrian database. These results show that human detection have been performed successfully, with a lack in the fusion of multiple detection.

## 4.7 Critical Path & Analysis

The summary of the implemenation is summarized in the table below

|  |  |  |  |
| --- | --- | --- | --- |
|  | Used | Available | % |
| Number of Logic Elements | 83,497 | 114,480 | 72.9 |
| Number of Registers | 17,383 | 28,800 | 60.4 |
| Number of fully used LUT-FF pairs | 2,070 | 23,109 | 9.0 |
| Number of Memory bits | 2,800,000 | 3,900,000 | 71.8 |
| MAX Frequency(MHz) | 50 |  |  |

Our implementation requires around 385,000 clock cycles to extract HOG data from an entire image with the appropriate scaling techniques, and subsequently perform detection using SVM. This means, at a maximum clock settings of 50 MHZ, we would be able to produce a frame rate of about 129 fps. In other words, we would be able to completely scan one image in about 0,007 seconds. With the average response time of a human driver being above 0,2 seconds, this means that our system should be able to improve the safety of both manual and autonomous cars significantly.

5. Conclusion

In this report, compact FPGA implementation of real-time human detection using the HOG feature extraction and SVM has been presented. As a result of empirical evaluation with an experimental setup equipped with an FPGA and camera device, the throughput of 62.5 fps was achieved without

using any external memory modules. If a high-speed camera device was available, the maximum throughput of 112 fps was expected to be accomplished. While some simplification was made to alleviate hardware complexity

References

1. http://www.lsi-contest.com/shiyou\_1e.html, accessed in 27 December 2016
2. https://www.mathworks.com/help/fixedpoint/ug/convert-cartesian-to-polar-using-cordic-vectoring-kernel.html, accessed in 27 December 2016Adiono, Trio. *Perancangan Sistem VLSI,* Pusat Mikroelektronika ITB, Bandung, 2012.
3. Ciletti, M.D. *Advanced Digital Design With the Verilog HDL,* Publishing House of Electonics Industry, Beijing, 2010.
4. R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura, “Hardware Architecture for HOG Feature Extraction,” in Proceedings of the 2009 International Conference on Intelligent Information Hiding and Multimedia Signal Processing. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1330–1333.
5. Elkin García, Fernando Lozano, ” Boosting Support Vector Machines”, Departamento de Ingeniería Eléctrica y Electrónica,Universidad de los Andes, Bogotá, Colombia

Appendix: Verilog Code

(Note: to conserve length, not all of the codes are added here)

1. Array code
2. module array8 (
3. input clk ,
4. input reset,
5. input en,
6. input wr\_en,
7. input [15:0] wr\_data,
8. input [2:0] wr\_idx,
9. input rd\_en,
10. input [2:0] rd\_idx,
11. output reg [15:0] rd\_data
12. );
13. reg [15:0] buffer [0:7];
14. reg state;
15. reg i;
16. localparam buff\_0 = 236;
17. localparam buff\_1 = 175;
18. localparam buff\_2 = 85;
19. localparam buff\_3 = 13;
20. localparam buff\_4 = 120;
21. localparam buff\_5 = 46;
22. localparam buff\_6 = 13;
23. localparam buff\_7 = 99;
24. localparam STATE\_IDLE = 1;
25. localparam STATE\_ACTIVE = 0;
26. always @(posedge clk)
27. begin
28. if(reset == 1 )
29. begin
30. rd\_data <= 0;
31. state <= STATE\_IDLE;
32. end
33. else
34. begin
35. case(state)
36. STATE\_IDLE :
37. begin
38. if(en == 1 && reset == 0)
39. begin
40. buffer[0] <= buff\_0[7:0];
41. buffer[1] <= buff\_1[7:0];
42. buffer[2] <= buff\_2[7:0];
43. buffer[3] <= buff\_3[7:0];
44. buffer[4] <= buff\_4[7:0];
45. buffer[5] <= buff\_5[7:0];
46. buffer[6] <= buff\_6[7:0];
47. buffer[7] <= buff\_7[7:0];
48. state <= STATE\_ACTIVE;
49. end
50. end
51. STATE\_ACTIVE:
52. begin
53. if(wr\_en == 1 && rd\_en == 0)
54. begin
55. rd\_data <= 0;
56. buffer[wr\_idx] <= wr\_data;
57. end
58. else if(wr\_en == 1 && rd\_en == 1)
59. begin
60. rd\_data <= buffer[rd\_idx];
61. buffer[wr\_idx] <= wr\_data;
62. end
63. else if(wr\_en == 0 && rd\_en == 1)
64. begin
65. rd\_data <= buffer[rd\_idx];
66. end
67. else if(wr\_en == 0 && rd\_en == 0)
68. begin
69. rd\_data <= 0;
70. end
71. end
72. endcase
73. end
74. end
75. endmodule
76. Scaling
77. `define ROW\_DIV 134
78. `define COL\_DIV 70
79. module calculateScaleStep(
80. input wire [11:0] rows,
81. input wire [11:0] cols,
82. output reg [4:0] step
83. );
84. reg [5:0] a;
85. reg [5:0] b;
86. reg [5:0] min;
87. always@ (rows or cols)
88. begin
89. a = rows/`ROW\_DIV;
90. b = cols/`COL\_DIV;
91. if (a < b)
92. min = a;
93. else
94. min = b;
96. if (min>=1 && min <1.25)
97. step <= 1;
98. else if (min>=1.25 && min<1.56)
99. step <= 2;
100. else if (min>=1.56 && min<1.95)
101. step <= 3;
102. else if (min>=1.95 && min<2.44)
103. step <= 4;
104. else if (min>2.44 && min<3.05)
105. step <= 5;
106. else if (min>3.05 && min<3.81)
107. step <= 6;
108. else if (min>3.81 && min<4.77)
109. step <= 7;
110. else if (min>4.77 && min<5.96)
111. step <= 8;
112. else if (min>5.96 && min<7.46)
113. step <= 9;
114. else if (min>7.46 && min<9.33)
115. step <= 10;
116. else if (min>9.33 && min<11.66)
117. step <= 11;
118. else if (min>11.66 && min<14.58)
119. step <= 12;
120. else if (min>14.58 && min<18.23)
121. step <= 13;
122. else if (min>18.23 && min<22.79)
123. step <= 14;
124. else if (min>22.79 && min<28.5)
125. step <= 15;
126. else if (min>28.5 && min<35.63)
127. step <= 16;
128. else
129. step <= 17;
130. end
131. endmodule
132. Square root
133. module isqrt(
134. input wire clk,
135. input wire [7:0] dx,
136. input wire [7:0] dy,
137. output reg [7:0] magnitude
138. )
139. reg [15:0] m;
140. reg [15:0] res;
141. reg [15:0] bit;
142. reg [15:0] temp;
143. always@(posedge clk)
144. begin
145. m = (dx\*dx) + (dy\*dy);
146. res = 0;
147. bit = 1 << 14;
149. while (bit > num)
150. bit = bit >> 2;
152. while (bit != 0)
153. begin
154. if (num >= res + bit)
155. begin
156. num = num - res + bit;
157. res = (res >> 1) + bit;
158. end
159. else
160. begin
161. res = res >> 1;
162. bit = bit >> 2;
163. end
164. end
165. magnitude <= m;
166. end
167. endmodule

1. GetHistogram
2. module getHistogram
3. (
4. input wire clk,
5. input wire [15:0] magnitudes, // ufix14\_En7
6. input wire [13:0] angles\_1, // sfix14\_En1 output ce\_out;
7. input wire enable,
8. output reg [135:0] H // sfix14\_E1
9. );
11. reg [13:0] array [0:8];
13. always @(posedge clk)
14. begin
15. if (enable == 1'b1)
16. begin
17. if (angles\_1 == 4'b0000)
18. begin
19. array[0] <= array[0] + magnitudes;
20. end
21. else if (angles\_1 == 4'b0001)
22. begin
23. array[1] <= array[1] + magnitudes;
24. end
25. else if (angles\_1 == 4'b0010)
26. begin
27. array[2] <= array[2] + magnitudes;
28. end
29. else if (angles\_1 == 4'b0011)
30. begin
31. array[3] <= array[3] + magnitudes;
32. end
33. else if (angles\_1 == 4'b0100)
34. begin
35. array[4] <= array[4] + magnitudes;
36. end
37. else if (angles\_1 == 4'b0101)
38. begin
39. array[5] <= array[5] + magnitudes;
40. end
41. else if (angles\_1 == 4'b0110)
42. begin
43. array[6] <= array[6] + magnitudes;
44. end
45. else if (angles\_1 == 4'b0111)
46. begin
47. array[7] <= array[7] + magnitudes;
48. end
49. else if (angles\_1 == 4'b1000)
50. begin
51. array[8] <= array[8] + magnitudes;
52. end
53. H <= {array[8], array[7], array[6], array[5], array[4], array[3], array[2], array[1], array[0]};
54. end
55. else
56. begin
57. H <= 135'b0;
58. array[0] <= 13'b0;
59. array[1] <= 13'b0;
60. array[2] <= 13'b0;
61. array[3] <= 13'b0;
62. array[4] <= 13'b0;
63. array[5] <= 13'b0;
64. array[6] <= 13'b0;
65. array[7] <= 13'b0;
66. array[8] <= 13'b0;
67. end

70. end

endmodule

1. Magnitude and angle
2. `define tan20 0.3640
3. `define tan40 0.8391
4. `define tan60 1.7321
5. `define tan80 5.6713
6. module magtan (
7. input clk,
8. input rst,
9. input wire [15:0] dx,
10. input wire [15:0] dy,
11. output reg [15:0] magnitude,
12. output reg [3:0] tan
13. );
14. reg [15:0] mag;
15. reg [15:0] dx1, dy1;
16. reg [3:0] neg;
17. always @(posedge clk)
18. begin
19. if (rst == 1)
20. begin
21. magnitude <= 0;
22. tan <=0;
23. end
24. else
25. begin
26. if(dx[15] == 1'b1)
27. dx1 = ~dx;
28. else
29. dx1 = dx;
30. if(dy[15]== 1'b1)
31. dy1 = ~dy;
32. else
33. dy1 = dy;
34. mag = dx1 + dy1;
35. end
36. end
37. always @\*
38. begin
39. if(dx[15]==dy[15])
40. neg <= 0;
41. else
42. neg <= 0;
43. end
44. always @(posedge clk)
45. begin
46. if (neg == 0)
47. begin
48. if (dy < `tan20\*dx)
49. tan = 4'b0000;
50. else
51. if ((dy > `tan20\*dx) && (dy < `tan40\*dx))
52. tan = 4'b0001;
53. else
54. if ((dy > `tan40\*dx) && (dy < `tan60\*dx))
55. tan = 4'b0010;
56. else
57. if ((dy > `tan60\*dx) && (dy < `tan80\*dx))
58. tan = 4'b0011;
59. else tan = 4'b1000;
60. end
61. else
62. begin
63. if (dy < `tan20\*dx)
64. tan = 4'b0100;
65. else
66. if ((dy > `tan20\*dx) & (dy < `tan40\*dx))
67. tan = 4'b0101;
68. else
69. if ((dy > `tan40\*dx) & (dy < `tan60\*dx))
70. tan = 4'b0110;
71. else
72. if ((dy > `tan60\*dx) & (dy < `tan80\*dx))
73. tan = 4'b0111;
74. else tan = 4'b1000;
75. end
76. end
77. endmodule
78. Normalization
79. module root\_norm(
80. input wire clk,
81. input wire [16:0] sqra,
82. input wire [16:0] sqrb,
83. input wire [16:0] sqrc,
84. input wire [16:0] sqrd,
85. input wire [16:0] sqre,
86. input wire [16:0] sqrf,
87. input wire [16:0] sqrg,
88. input wire [16:0] sqrh,
89. input wire [16:0] sqri,
90. output reg [15:0] magnitude
91. );
92. reg [16:0] m;
93. reg [16:0] res;
94. reg [16:0] bit;
95. always@(posedge clk)
96. begin
97. m = sqra + sqrb + sqrc + sqrd + sqre + sqrf + sqrg + sqrh + sqri;
98. res = 0;
99. bit = 1 << 14;
101. while (bit > m)
102. bit = bit >> 2;
103. while (bit != 0)
104. begin
105. if (m >= res + bit)
106. begin
107. m = m - (res + bit);
108. res = res + (bit << 1);
109. end
110. else
111. begin
112. res = res >> 1;
113. bit = bit >> 2;
114. end
115. end
116. magnitude <= res;
117. end
118. endmodule




124. Matrix multiplication
125. module top(clk,reset,ain1,ain2,ain3,ain4,ain5,ain6,ain7,ain8,bin1,bin2,bin3,bin4,bin5,bin6,bin7,bin8
126. ,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16,c17,c18,c19,c20,c21,c22,c23,c24,c25,c26,c27,c28,c29,
127. c30,c31,c32,c33,c34,c35,c36,c37,c38,c39,c40,c41,c42,c43,c44,c45,c46,c47,c48,c49,c50,c51,c52,c53,c54,c55,c56,c57,c58,c59,
128. c60,c61,c62,c63,c64);
129. parameter data\_size=3;
130. input wire clk,reset;
131. input wire [data\_size-1:0] ain1,ain2,ain3,ain4,ain5,ain6,ain7,ain8,bin1,bin2,bin3,bin4,bin5,bin6,bin7,bin8;
132. output wire [2\*data\_size:0] c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16,c17,c18,c19,c20,c21,c22,c23,c24,c25,c26,c27,c28,c29;
133. output wire [2\*data\_size:0] c30,c31,c32,c33,c34,c35,c36,c37,c38,c39,c40,c41,c42,c43,c44,c45,c46,c47,c48,c49,c50,c51,c52,c53,c54,c55,c56,c57,c58,c59;
134. output wire [2\*data\_size:0] c60,c61,c62,c63,c64;

137. wire [data\_size-1:0] a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16,a17,a18,a19,a20,a21,a22,a23,a24,a25,a26,a27,a28,a29;
138. wire [data\_size-1:0] a30,a31,a32,a33,a34,a35,a36,a37,a38,a39,a40,a41,a42,a43,a44,a45,a46,a47,a48,a49,a50,a51,a52,a53,a54,a55,a56,a57,a58,a59;
139. wire [data\_size-1:0] a60,a61,a62,a63,a64;
141. wire [data\_size-1:0] b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19,b20,b21,b22,b23,b24,b25,b26,b27,b28,b29;
142. wire [data\_size-1:0] b30,b31,b32,b33,b34,b35,b36,b37,b38,b39,b40,b41,b42,b43,b44,b45,b46,b47,b48,b49,b50,b51,b52,b53,b54,b55,b56,b57,b58,b59;
143. wire [data\_size-1:0] b60,b61,b62,b63,b64;
145. pe pe1 (.clk(clk), .reset(reset), .in\_a(ain1), .in\_b(bin1), .out\_a(a1), .out\_b(b1), .out\_c(c1));
146. pe pe2 (.clk(clk), .reset(reset), .in\_a(a1), .in\_b(bin2), .out\_a(a2), .out\_b(b2), .out\_c(c2));
147. pe pe3 (.clk(clk), .reset(reset), .in\_a(a2), .in\_b(bin3), .out\_a(a3), .out\_b(b3), .out\_c(c3));
148. pe pe4 (.clk(clk), .reset(reset), .in\_a(a3), .in\_b(bin4), .out\_a(a4), .out\_b(b4), .out\_c(c4));
149. pe pe5 (.clk(clk), .reset(reset), .in\_a(a4), .in\_b(bin5), .out\_a(a5), .out\_b(b5), .out\_c(c5));
150. pe pe6 (.clk(clk), .reset(reset), .in\_a(a5), .in\_b(bin6), .out\_a(a6), .out\_b(b6), .out\_c(c6));
151. pe pe7 (.clk(clk), .reset(reset), .in\_a(a6), .in\_b(bin7), .out\_a(a7), .out\_b(b7), .out\_c(c7));
152. pe pe8 (.clk(clk), .reset(reset), .in\_a(a7), .in\_b(bin8), .out\_a(a8), .out\_b(b8), .out\_c(c8));
153. pe pe9 (.clk(clk), .reset(reset), .in\_a(ain2), .in\_b(b1), .out\_a(a9), .out\_b(b9), .out\_c(c9));
154. pe pe10 (.clk(clk), .reset(reset), .in\_a(a9), .in\_b(b2), .out\_a(a10), .out\_b(b10), .out\_c(c10));
155. pe pe11 (.clk(clk), .reset(reset), .in\_a(a10), .in\_b(b3), .out\_a(a11), .out\_b(b11), .out\_c(c11));
156. pe pe12 (.clk(clk), .reset(reset), .in\_a(a11), .in\_b(b4), .out\_a(a12), .out\_b(b12), .out\_c(c12));
157. pe pe13 (.clk(clk), .reset(reset), .in\_a(a12), .in\_b(b5), .out\_a(a13), .out\_b(b13), .out\_c(c13));
158. pe pe14 (.clk(clk), .reset(reset), .in\_a(a13), .in\_b(b6), .out\_a(a14), .out\_b(b14), .out\_c(c14));
159. pe pe15 (.clk(clk), .reset(reset), .in\_a(a14), .in\_b(b7), .out\_a(a15), .out\_b(b15), .out\_c(c15));
160. pe pe16 (.clk(clk), .reset(reset), .in\_a(a15), .in\_b(b8), .out\_a(a16), .out\_b(b16), .out\_c(c16));
161. pe pe17 (.clk(clk), .reset(reset), .in\_a(ain3), .in\_b(b9), .out\_a(a17), .out\_b(b17), .out\_c(c17));
162. pe pe18 (.clk(clk), .reset(reset), .in\_a(a17), .in\_b(b10), .out\_a(a18), .out\_b(b18), .out\_c(c18));
163. pe pe19 (.clk(clk), .reset(reset), .in\_a(a18), .in\_b(b11), .out\_a(a19), .out\_b(b19), .out\_c(c19));
164. pe pe20 (.clk(clk), .reset(reset), .in\_a(a19), .in\_b(b12), .out\_a(a20), .out\_b(b20), .out\_c(c20));
165. pe pe21 (.clk(clk), .reset(reset), .in\_a(a20), .in\_b(b13), .out\_a(a21), .out\_b(b21), .out\_c(c21));
166. pe pe22 (.clk(clk), .reset(reset), .in\_a(a21), .in\_b(b14), .out\_a(a22), .out\_b(b22), .out\_c(c22));
167. pe pe23 (.clk(clk), .reset(reset), .in\_a(a22), .in\_b(b15), .out\_a(a23), .out\_b(b23), .out\_c(c23));
168. pe pe24 (.clk(clk), .reset(reset), .in\_a(a23), .in\_b(b16), .out\_a(a24), .out\_b(b24), .out\_c(c24));
169. pe pe25 (.clk(clk), .reset(reset), .in\_a(ain4), .in\_b(b17), .out\_a(a25), .out\_b(b25), .out\_c(c25));
170. pe pe26 (.clk(clk), .reset(reset), .in\_a(a25), .in\_b(b18), .out\_a(a26), .out\_b(b26), .out\_c(c26));
171. pe pe27 (.clk(clk), .reset(reset), .in\_a(a26), .in\_b(b19), .out\_a(a27), .out\_b(b27), .out\_c(c27));
172. pe pe28 (.clk(clk), .reset(reset), .in\_a(a27), .in\_b(b20), .out\_a(a28), .out\_b(b28), .out\_c(c28));
173. pe pe29 (.clk(clk), .reset(reset), .in\_a(a28), .in\_b(b21), .out\_a(a29), .out\_b(b29), .out\_c(c29));
174. pe pe30 (.clk(clk), .reset(reset), .in\_a(a29), .in\_b(b22), .out\_a(a30), .out\_b(b30), .out\_c(c30));
175. pe pe31 (.clk(clk), .reset(reset), .in\_a(a30), .in\_b(b23), .out\_a(a31), .out\_b(b31), .out\_c(c31));
176. pe pe32 (.clk(clk), .reset(reset), .in\_a(a31), .in\_b(b24), .out\_a(a32), .out\_b(b32), .out\_c(c32));
177. pe pe33 (.clk(clk), .reset(reset), .in\_a(ain5), .in\_b(b25), .out\_a(a33), .out\_b(b33), .out\_c(c33));
178. pe pe34 (.clk(clk), .reset(reset), .in\_a(a33), .in\_b(b26), .out\_a(a34), .out\_b(b34), .out\_c(c34));
179. pe pe35 (.clk(clk), .reset(reset), .in\_a(a34), .in\_b(b27), .out\_a(a35), .out\_b(b35), .out\_c(c35));
180. pe pe36 (.clk(clk), .reset(reset), .in\_a(a35), .in\_b(b28), .out\_a(a36), .out\_b(b36), .out\_c(c36));
181. pe pe37 (.clk(clk), .reset(reset), .in\_a(a36), .in\_b(b29), .out\_a(a37), .out\_b(b37), .out\_c(c37));
182. pe pe38 (.clk(clk), .reset(reset), .in\_a(a37), .in\_b(b30), .out\_a(a38), .out\_b(b38), .out\_c(c38));
183. pe pe39 (.clk(clk), .reset(reset), .in\_a(a38), .in\_b(b31), .out\_a(a39), .out\_b(b39), .out\_c(c39));
184. pe pe40 (.clk(clk), .reset(reset), .in\_a(a39), .in\_b(b32), .out\_a(a40), .out\_b(b40), .out\_c(c40));
185. pe pe41 (.clk(clk), .reset(reset), .in\_a(ain6), .in\_b(b33), .out\_a(a41), .out\_b(b41), .out\_c(c41));
186. pe pe42 (.clk(clk), .reset(reset), .in\_a(a41), .in\_b(b34), .out\_a(a42), .out\_b(b42), .out\_c(c42));
187. pe pe43 (.clk(clk), .reset(reset), .in\_a(a42), .in\_b(b35), .out\_a(a43), .out\_b(b43), .out\_c(c43));
188. pe pe44 (.clk(clk), .reset(reset), .in\_a(a43), .in\_b(b36), .out\_a(a44), .out\_b(b44), .out\_c(c44));
189. pe pe45 (.clk(clk), .reset(reset), .in\_a(a44), .in\_b(b37), .out\_a(a45), .out\_b(b45), .out\_c(c45));
190. pe pe46 (.clk(clk), .reset(reset), .in\_a(a45), .in\_b(b38), .out\_a(a46), .out\_b(b46), .out\_c(c46));
191. pe pe47 (.clk(clk), .reset(reset), .in\_a(a46), .in\_b(b39), .out\_a(a47), .out\_b(b47), .out\_c(c47));
192. pe pe48 (.clk(clk), .reset(reset), .in\_a(a47), .in\_b(b40), .out\_a(a48), .out\_b(b48), .out\_c(c48));
193. pe pe49 (.clk(clk), .reset(reset), .in\_a(ain7), .in\_b(b41), .out\_a(a49), .out\_b(b49), .out\_c(c49));
194. pe pe50 (.clk(clk), .reset(reset), .in\_a(a49), .in\_b(b42), .out\_a(a50), .out\_b(b50), .out\_c(c50));
195. pe pe51 (.clk(clk), .reset(reset), .in\_a(a50), .in\_b(b43), .out\_a(a51), .out\_b(b51), .out\_c(c51));
196. pe pe52 (.clk(clk), .reset(reset), .in\_a(a51), .in\_b(b44), .out\_a(a52), .out\_b(b52), .out\_c(c52));
197. pe pe53 (.clk(clk), .reset(reset), .in\_a(a52), .in\_b(b45), .out\_a(a53), .out\_b(b53), .out\_c(c53));
198. pe pe54 (.clk(clk), .reset(reset), .in\_a(a53), .in\_b(b46), .out\_a(a54), .out\_b(b54), .out\_c(c54));
199. pe pe55 (.clk(clk), .reset(reset), .in\_a(a54), .in\_b(b47), .out\_a(a55), .out\_b(b55), .out\_c(c55));
200. pe pe56 (.clk(clk), .reset(reset), .in\_a(a55), .in\_b(b48), .out\_a(a56), .out\_b(b56), .out\_c(c56));
201. pe pe57 (.clk(clk), .reset(reset), .in\_a(ain8), .in\_b(b49), .out\_a(a57), .out\_b(b57), .out\_c(c57));
202. pe pe58 (.clk(clk), .reset(reset), .in\_a(a57), .in\_b(b50), .out\_a(a58), .out\_b(b58), .out\_c(c58));
203. pe pe59 (.clk(clk), .reset(reset), .in\_a(a58), .in\_b(b51), .out\_a(a59), .out\_b(b59), .out\_c(c59));
204. pe pe60 (.clk(clk), .reset(reset), .in\_a(a59), .in\_b(b52), .out\_a(a60), .out\_b(b60), .out\_c(c60));
205. pe pe61 (.clk(clk), .reset(reset), .in\_a(a60), .in\_b(b53), .out\_a(a61), .out\_b(b61), .out\_c(c61));
206. pe pe62 (.clk(clk), .reset(reset), .in\_a(a61), .in\_b(b54), .out\_a(a62), .out\_b(b62), .out\_c(c62));
207. pe pe63 (.clk(clk), .reset(reset), .in\_a(a62), .in\_b(b55), .out\_a(a63), .out\_b(b63), .out\_c(c63));
208. pe pe64 (.clk(clk), .reset(reset), .in\_a(a63), .in\_b(b56), .out\_a(a64), .out\_b(b64), .out\_c(c64));

211. endmodule
212. module pe(clk,reset,in\_a,in\_b,out\_a,out\_b,out\_c);
213. parameter data\_size=8;
214. input wire reset,clk;
215. input wire [data\_size-1:0] in\_a,in\_b;
216. output reg [2\*data\_size:0] out\_c;
217. output reg [data\_size-1:0] out\_a,out\_b;
218. always @(posedge clk)begin
219. if(reset) begin
220. out\_a=0;
221. out\_b=0;
222. out\_c=0;
223. end
224. else begin
225. out\_c=out\_c+in\_a\*in\_b;
226. out\_a=in\_a;
227. out\_b=in\_b;
228. end
229. end

endmodule

1. VGA
2. module VGA\_Controller( // Host Side
3. iCursor\_RGB\_EN,
4. iCursor\_X,
5. iCursor\_Y,
6. iCursor\_R,
7. iCursor\_G,
8. iCursor\_B,
9. iRed,
10. iGreen,
11. iBlue,
12. oAddress,
13. oCoord\_X,
14. oCoord\_Y,
15. // VGA Side
16. oVGA\_R,
17. oVGA\_G,
18. oVGA\_B,
19. oVGA\_H\_SYNC,
20. oVGA\_V\_SYNC,
21. oVGA\_SYNC,
22. oVGA\_BLANK,
23. oVGA\_CLOCK,
24. // Control Signal
25. iCLK\_25,
26. iRST\_N );
27. // Horizontal Parameter ( Pixel )
28. parameter H\_SYNC\_CYC = 96;
29. parameter H\_SYNC\_BACK = 45+3;
30. parameter H\_SYNC\_ACT = 640; // 646
31. parameter H\_SYNC\_FRONT= 13+3;
32. parameter H\_SYNC\_TOTAL= 800;
33. // Virtical Parameter ( Line )
34. parameter V\_SYNC\_CYC = 2;
35. parameter V\_SYNC\_BACK = 30+2;
36. parameter V\_SYNC\_ACT = 480; // 484
37. parameter V\_SYNC\_FRONT= 9+2;
38. parameter V\_SYNC\_TOTAL= 525;
39. // Start Offset
40. parameter X\_START = H\_SYNC\_CYC+H\_SYNC\_BACK+4;
41. parameter Y\_START = V\_SYNC\_CYC+V\_SYNC\_BACK;
42. // Host Side
43. output reg [19:0] oAddress;
44. output reg [9:0] oCoord\_X;
45. output reg [9:0] oCoord\_Y;
46. input [3:0] iCursor\_RGB\_EN;
47. input [9:0] iCursor\_X;
48. input [9:0] iCursor\_Y;
49. input [9:0] iCursor\_R;
50. input [9:0] iCursor\_G;
51. input [9:0] iCursor\_B;
52. input [9:0] iRed;
53. input [9:0] iGreen;
54. input [9:0] iBlue;
55. // VGA Side
56. output [9:0] oVGA\_R;
57. output [9:0] oVGA\_G;
58. output [9:0] oVGA\_B;
59. output reg oVGA\_H\_SYNC;
60. output reg oVGA\_V\_SYNC;
61. output oVGA\_SYNC;
62. output oVGA\_BLANK;
63. output oVGA\_CLOCK;
64. // Control Signal
65. input iCLK\_25;
66. input iRST\_N;
67. // Internal Registers and Wires
68. reg [9:0] H\_Cont;
69. reg [9:0] V\_Cont;
70. reg [9:0] Cur\_Color\_R;
71. reg [9:0] Cur\_Color\_G;
72. reg [9:0] Cur\_Color\_B;
73. wire mCLK;
74. wire mCursor\_EN;
75. wire mRed\_EN;
76. wire mGreen\_EN;
77. wire mBlue\_EN;
78. assign oVGA\_BLANK = oVGA\_H\_SYNC & oVGA\_V\_SYNC;
79. assign oVGA\_SYNC = 1'b0;
80. assign oVGA\_CLOCK = ~iCLK\_25;
81. assign mCursor\_EN = iCursor\_RGB\_EN[3];
82. assign mRed\_EN = iCursor\_RGB\_EN[2];
83. assign mGreen\_EN = iCursor\_RGB\_EN[1];
84. assign mBlue\_EN = iCursor\_RGB\_EN[0];
85. assign mCLK = iCLK\_25;
86. assign oVGA\_R = ( H\_Cont>=X\_START+9 && H\_Cont<X\_START+H\_SYNC\_ACT+9 &&
87. V\_Cont>=Y\_START && V\_Cont<Y\_START+V\_SYNC\_ACT )
88. ? (mRed\_EN ? Cur\_Color\_R : 0) : 0;
89. assign oVGA\_G = ( H\_Cont>=X\_START+9 && H\_Cont<X\_START+H\_SYNC\_ACT+9 &&
90. V\_Cont>=Y\_START && V\_Cont<Y\_START+V\_SYNC\_ACT )
91. ? (mGreen\_EN ? Cur\_Color\_G : 0) : 0;
92. assign oVGA\_B = ( H\_Cont>=X\_START+9 && H\_Cont<X\_START+H\_SYNC\_ACT+9 &&
93. V\_Cont>=Y\_START && V\_Cont<Y\_START+V\_SYNC\_ACT )
94. ? (mBlue\_EN ? Cur\_Color\_B : 0) : 0;
95. // Pixel LUT Address Generator
96. always@(posedge mCLK or negedge iRST\_N)
97. begin
98. if(!iRST\_N)
99. begin
100. oCoord\_X <= 0;
101. oCoord\_Y <= 0;
102. oAddress <= 0;
103. end
104. else
105. begin
106. if( H\_Cont>=X\_START && H\_Cont<X\_START+H\_SYNC\_ACT &&
107. V\_Cont>=Y\_START && V\_Cont<Y\_START+V\_SYNC\_ACT )
108. begin
109. oCoord\_X <= H\_Cont-X\_START;
110. oCoord\_Y <= V\_Cont-Y\_START;
111. oAddress <= oCoord\_Y\*H\_SYNC\_ACT+oCoord\_X-3;
112. end
113. end
114. end
115. // Cursor Generator
116. always@(posedge mCLK or negedge iRST\_N)
117. begin
118. if(!iRST\_N)
119. begin
120. Cur\_Color\_R <= 0;
121. Cur\_Color\_G <= 0;
122. Cur\_Color\_B <= 0;
123. end
124. else
125. begin
126. if( H\_Cont>=X\_START+8 && H\_Cont<X\_START+H\_SYNC\_ACT+8 &&
127. V\_Cont>=Y\_START && V\_Cont<Y\_START+V\_SYNC\_ACT )
128. begin
129. if( ( (H\_Cont==X\_START + 8 + iCursor\_X) ||
130. (H\_Cont==X\_START + 8 + iCursor\_X+1) ||
131. (H\_Cont==X\_START + 8 + iCursor\_X-1) ||
132. (V\_Cont==Y\_START + iCursor\_Y) ||
133. (V\_Cont==Y\_START + iCursor\_Y+1) ||
134. (V\_Cont==Y\_START + iCursor\_Y-1) )
135. && mCursor\_EN )
136. begin
137. Cur\_Color\_R <= iCursor\_R;
138. Cur\_Color\_G <= iCursor\_G;
139. Cur\_Color\_B <= iCursor\_B;
140. end
141. else
142. begin
143. Cur\_Color\_R <= iRed;
144. Cur\_Color\_G <= iGreen;
145. Cur\_Color\_B <= iBlue;
146. end
147. end
148. else
149. begin
150. Cur\_Color\_R <= iRed;
151. Cur\_Color\_G <= iGreen;
152. Cur\_Color\_B <= iBlue;
153. end
154. end
155. end
156. // H\_Sync Generator, Ref. 25 MHz Clock
157. always@(posedge mCLK or negedge iRST\_N)
158. begin
159. if(!iRST\_N)
160. begin
161. H\_Cont <= 0;
162. oVGA\_H\_SYNC <= 0;
163. end
164. else
165. begin
166. // H\_Sync Counter
167. if( H\_Cont < H\_SYNC\_TOTAL )
168. H\_Cont <= H\_Cont+1;
169. else
170. H\_Cont <= 0;
171. // H\_Sync Generator
172. if( H\_Cont < H\_SYNC\_CYC )
173. oVGA\_H\_SYNC <= 0;
174. else
175. oVGA\_H\_SYNC <= 1;
176. end
177. end
178. // V\_Sync Generator, Ref. H\_Sync
179. always@(posedge mCLK or negedge iRST\_N)
180. begin
181. if(!iRST\_N)
182. begin
183. V\_Cont <= 0;
184. oVGA\_V\_SYNC <= 0;
185. end
186. else
187. begin
188. // When H\_Sync Re-start
189. if(H\_Cont==0)
190. begin
191. // V\_Sync Counter
192. if( V\_Cont < V\_SYNC\_TOTAL )
193. V\_Cont <= V\_Cont+1;
194. else
195. V\_Cont <= 0;
196. // V\_Sync Generator
197. if( V\_Cont < V\_SYNC\_CYC )
198. oVGA\_V\_SYNC <= 0;
199. else
200. oVGA\_V\_SYNC <= 1;
201. end
202. end
203. end

endmodule