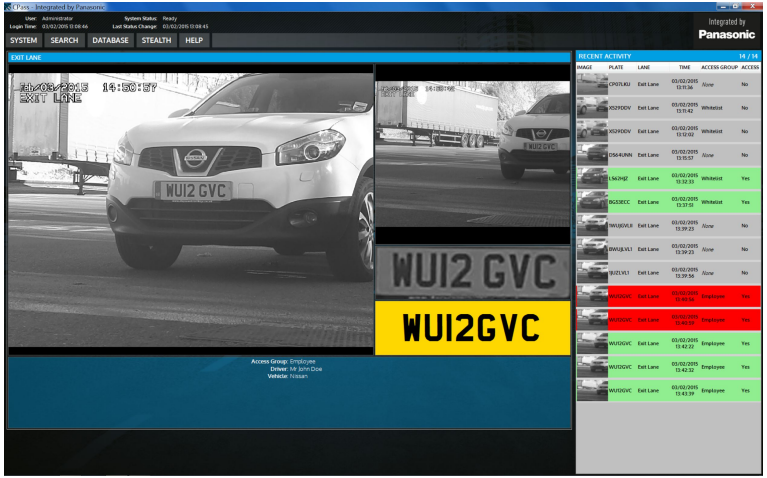
FPGA Vehicle Plate Reader

Team NSA: Darrin Ginoza, Jack Abernathy, Joe Squeri,
Patrick Murphy, Ravi Darbha

Project Motivation

The objective of this project is the development of a FPGA based vehicle plate recognition system. Currently vehicle plate recognition is an essential part of the infrastructure found in vehicle based systems concerning electronic payment, restricted area access, and the identification of criminal activities. FPGA based designs provide the ability for high performance recognition systems that are both low power and low latency.

Functionality - Overview

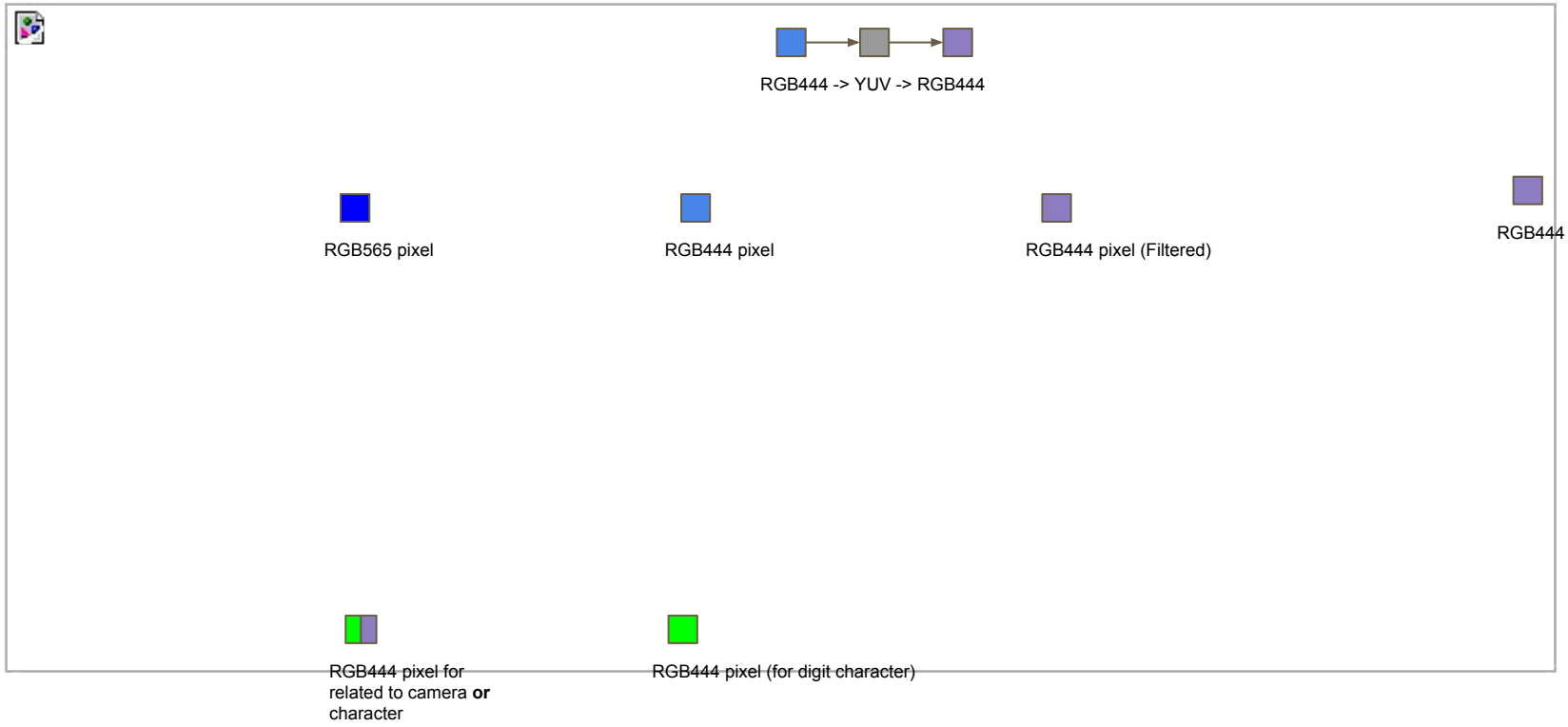


- Capture video from camera
- Process frames using edge detection
- Segment license plate characters
- Do character recognition on segments
- Display image processing output via VGA
- Switchable to display output of character recognition

Specification Overview

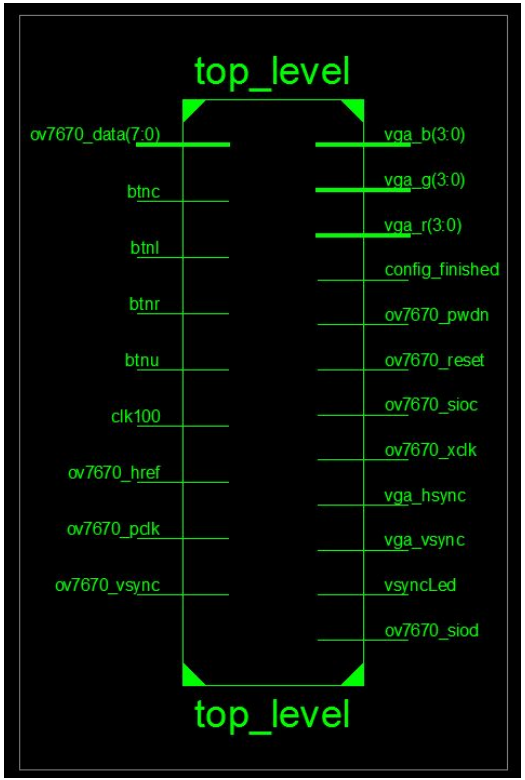
- We based our design on standard six-digit Massachusetts license plates
- Our segmentation implementation requires the plates to be approximately 18 inches in front of camera
- Character recognition is currently implemented on a numerical dictionary

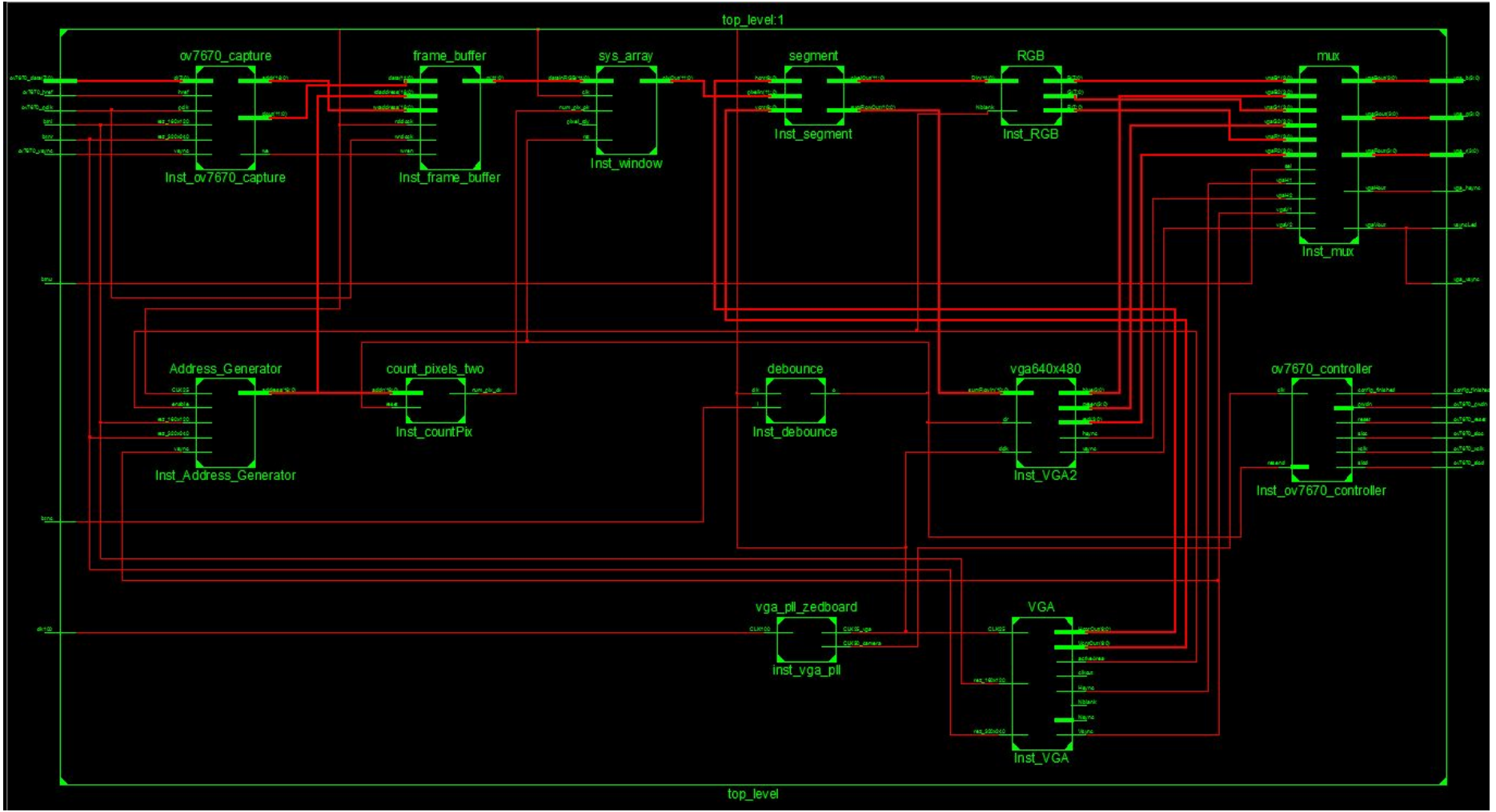
System Level Diagram



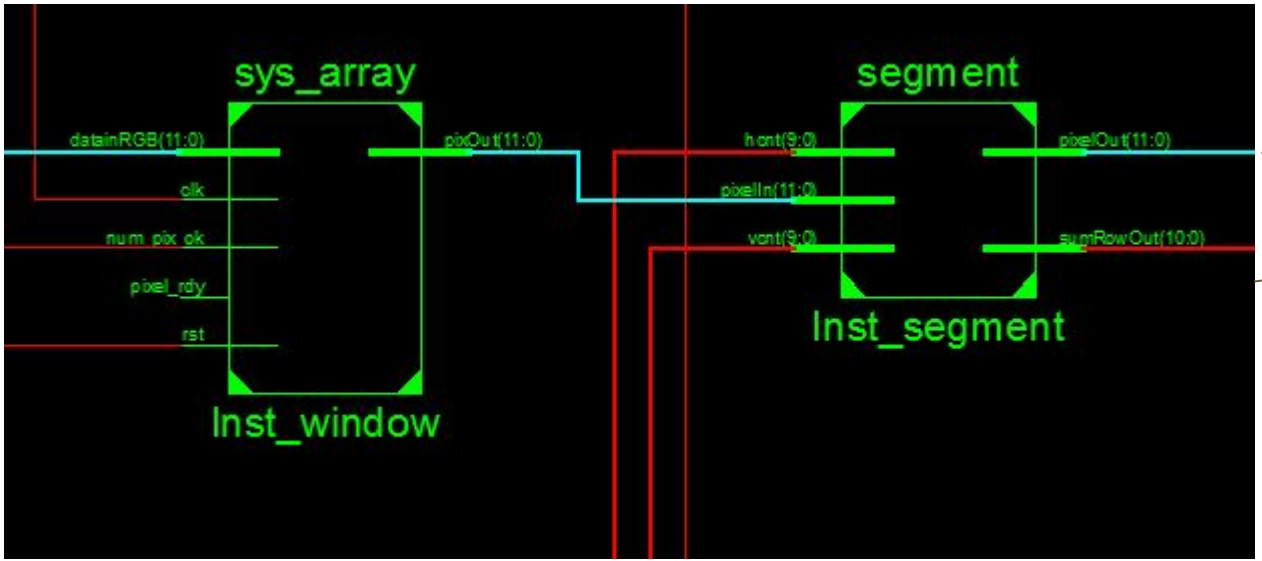
Images: https://en.wikipedia.org/wiki/Kernel_(image_processing)

RTL Overview




top_level:1
ov7670_capture
Inst_ov7670_capture
frame_buffer
Inst_frame_buffer
sys_array
Inst_window
segment
Inst_segment
RGB
Inst_RGB
mux
Inst_mux
Address_Generator
Inst_Address_Generator
count_pixels_two
Inst_countPix
debounce
Inst_debounce
vga640x480
Inst_VGA2
ov7670_controller
Inst_ov7670_controller
vga_pll_zedboard
inst_vga_pll
VGA
Inst_VGA
top_level

RTL Overview



Frame Buffer

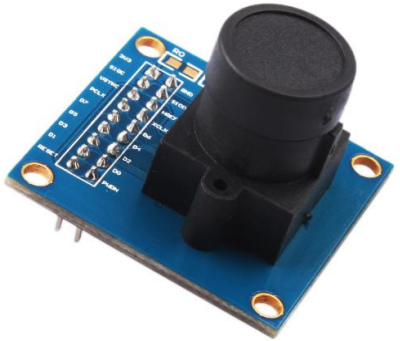
Data Counter

VGA_segment

VGA_OCR

Camera Controller

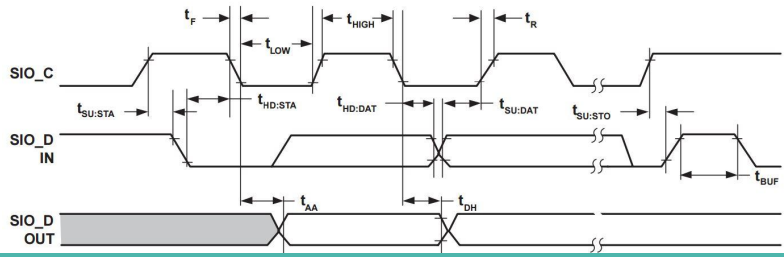
- OV7670 camera - low voltage CMOS image sensor capable of operating up to 30 frames per second.
- For our camera controller we modified existing code that interfaced the camera with the Zedboard found at:
  http://hamsterworks.co.nz/mediawiki/index.php/Zedboard_OV7670



Camera Controller - Capturing Pixels

- Camera provides full framed images (640 x 480 pixels) controlled through the Serial Camera Control Bus (SCCB) interface which is compatible with an I2C interface
- The camera transfers individual pixels in RGB565 format (Red: 5 bits, Green: 6 bits, Blue: 5 bits). Pixel data is streamed in row by row.
- Our camera controller captures the pixel data provided by the camera and truncates it to RGB444 format (Red: 4 bits, Green: 4 bits, Blue: 4 bits) before passing data onto the frame buffer.

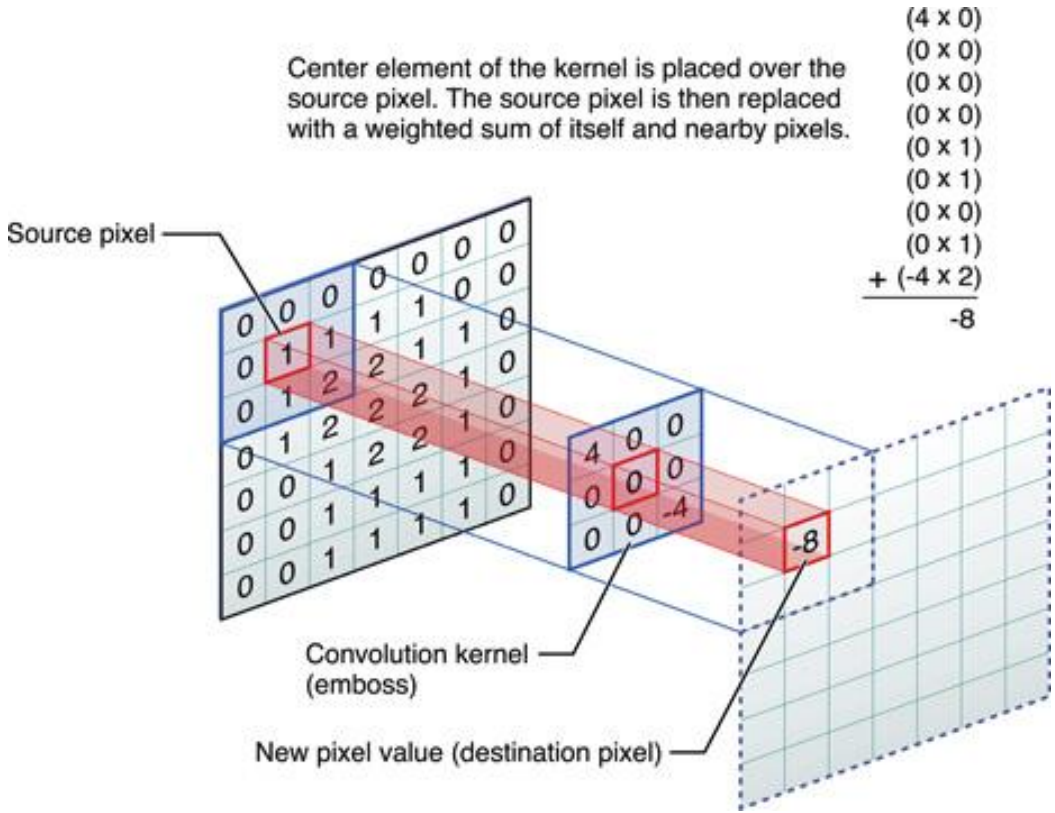
SCCB Timing Diagram



Camera Controller - Frame Buffer

- An IP core is used to buffer incoming frames. The frame buffer is capable of holding one entire frame (640 x 480 = 307,200 pixels) at once.
    - This is a total of 460,800 bytes as our pixels are 12 bits each.
- As a new frame is streamed in, it begins to overwrite the previous frame saved in the frame buffer

Window Operator - Kernel Convolution

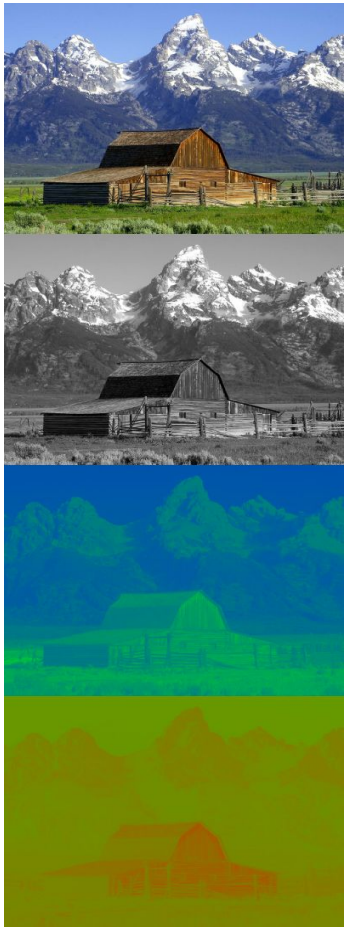


Window Module - Preprocessing

- Pixels are passed from the frame buffer into our window module.
- Before performing any windowing operation, the RGB pixel is converted to grayscale:

$$\begin{pmatrix} Y_r \\ U_r \\ V_r \end{pmatrix} = \begin{pmatrix} \left\lfloor \frac{R + 2G + B}{4} \right\rfloor \\ R - G \\ B - G \end{pmatrix}$$

Original



Y

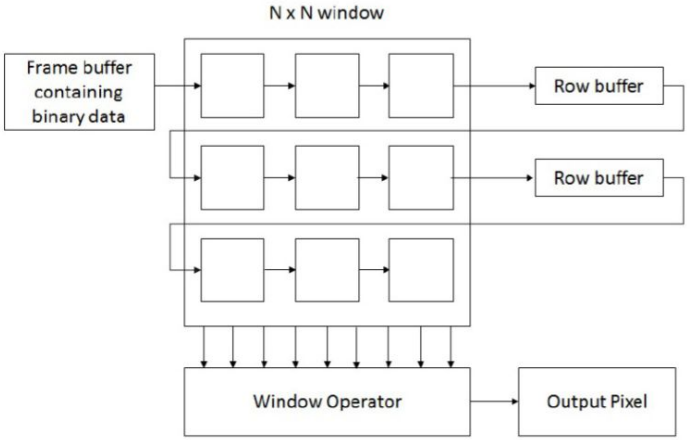
U

V

M. Po-Leen Ooi (2006) “Hardware Implementation for Face Detection on Xilinx Virtex-II FPGA using the Reversible Component Transformation Colour Space”v Electronic Design, Test and Applications, 2006. DELTA 2006. Third IEEE International Workshop on

Window Module - Design

- Our window module uses a 3x3 window to perform image processing.
- To ensure the proper pixels are used in our window operation, two FIFO queues (row buffers) are used in conjunction with our window. These are necessary as pixel data streams in row by row.



C. Supe, (2014) “Real Time Implementation of Spatial Filtering on FPGA”, Advances in Vision Computing, vol. 1, no. 4 December 2014.

Window Operators

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

- Experimented with a various window operations, as well as combining multiple operations:

Edge Detectors (Laplacian Spatial Filter):

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

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

Gaussian Blurring (Noise Reduction):

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Sharpening:

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

Sobel Filters:

-1	-2	-1
0	0	0
1	2	1

Horizontal

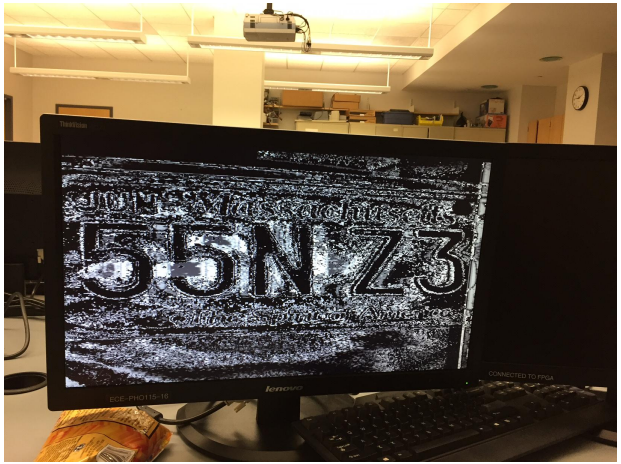
-1	0	1
-2	0	2
-1	0	1

Vertical

Window Operator - Edge Detection

- Ultimately found the below edge detection kernel provided the best results for purposes:

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



- However noise is still a significant issue

Verilog Example - Window Module

```
assign Y = (datainRGB[11:8] + 4'd2*datainRGB[7:4] + datainRGB[3:0])/4'd4; // Convert RGB pixel to grayscale, Y will be 4 bit value
assign Ur = datainRGB[11:8] - datainRGB[7:4];
assign Vr = datainRGB[3:0] - datainRGB[7:4];
assign datain = {Y,Ur,Vr};
// Setup our flip-flops to hold pixels in the window and fifo queues. 3x3 window operation used.
// Inputs to each flip-flop used for kernel convolution operation.
Dff dff1(prevPixels[0], datain, clk, reset);
Dff dff2(prevPixels[1], prevPixels[0], clk, reset);
Dff dff3(prevPixels[2], prevPixels[1], clk, reset);
// FIFOs of length 636 used (RowLength - 1). 636 used instead of 637 to correct clocking issues found in test benches
nFIFO #(.fifosize(636)) fifo1 (prevPixels[2], reset, clk, queue1out);
Dff dff4(prevPixels[3],queue1out, clk, reset);
Dff dff5(prevPixels[4], prevPixels[3], clk, reset);
Dff dff6(prevPixels[5], prevPixels[4], clk, reset);
nFIFO #(.fifosize(636)) fifo2(prevPixels[5], reset, clk, queue2out);
Dff dff7(prevPixels[6], queue2out, clk, reset);
Dff dff8(prevPixels[7],prevPixels[6], clk, reset);
Dff dff9(prevPixels[8], prevPixels[7], clk, reset);
```

Verilog Example - Window Module

```
always @ (posedge clk) begin
                if (rst) begin
                        tempOut0 <= 0;  // tempOut# holds a multiplication of a pixel with a weight in our kernel
                        tempOut1 <= 0;
                        tempOut2 <= 0;
                        tempOut3 <= 0;
                        tempOut4 <= 0;
                        tempOut5 <= 0;
                        tempOut6 <= 0;
                        tempOut7 <= 0;
                        tempOut8 <= 0;
                        weights[0] <= $signed(-1); // weights[] holds values used in kernel
                        weights[1] <= $signed(-1);
                        weights[2] <= $signed(-1);
                        weights[3] <= $signed(-1);
                        weights[4] <= $signed(8);
                        weights[5] <= $signed(-1);
                        weights[6] <= $signed(-1);
                        weights[7] <= $signed(-1);
                        weights[8] <= $signed(-1);
                end
```

Verilog Example - Window Module

```
else begin
        if (pixel_rdy) begin // if a pixel has been received
        // multiply pixel in window by appropriate weight. Note 0 added to start of each pixel to maintain sign of pixel
        tempOut0 <= $signed(weights[0])*$signed({1'b0,datain[11:8]});
        tempOut1 <= $signed(weights[1])*$signed({1'b0,prevPixels[0][11:8]});
        tempOut2 <= $signed(weights[2])*$signed({1'b0,prevPixels[1][11:8]});
        tempOut3 <= $signed(weights[3])*$signed({1'b0,queue1out[11:8]});
        tempOut4 <= $signed(weights[4])*$signed({1'b0,prevPixels[3][11:8]});
        tempOut5 <= $signed(weights[5])*$signed({1'b0,prevPixels[4][11:8]});
        tempOut6 <= $signed(weights[6])*$signed({1'b0,queue2out[11:8]});
        tempOut7 <= $signed(weights[7])*$signed({1'b0,prevPixels[6][11:8]});
        tempOut8 <= $signed(weights[8])*$signed({1'b0,prevPixels[7][11:8]});
        // tempOut holds the result of our window operation
        tempOut <= tempOut0 + tempOut1 + tempOut2 + tempOut3 + tempOut4 +
                    tempOut5 + tempOut6 + tempOut7 + tempOut8;
```

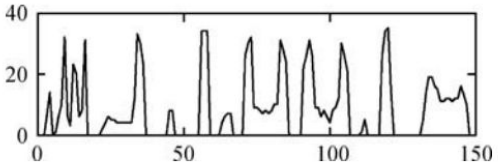
Verilog Example - Window Module

```
if(num_pix_ok)begin // if we've received enough pixels to run the convolution
// Saturation conditions:
        // tempOut is a 18 bit number. We need to return an 12 bit pixel to use in our processed image.
        // if tempOut is less than 0 we return a black pixel
        if($signed(tempOut) < 0)begin
                pixOut <= {4'b0000,4'b0000,4'b0000};
        end
        // if tempOut is greater than 15 we return a white pixel
        else if ($signed(tempOut) > 15) begin
                pixOut <= {4'b1111,4'b1111,4'b1111};
        end
        //otherwise (0 < pixel < 15) we return the value of the pixel. Note here we use the grayscale pixel value (Y),
        // which is 4 bits, for R, G, and B
        else begin
                pixOut <= {tempOut[3:0],tempOut[3:0],tempOut[3:0]};
        end
end
// otherwise output a black pixel (should result in black rows for first and last rows of processed image
else
        pixOut <= {4'b0000,4'b0000,4'b0000};
end
```

Character Segmentation - Dynamic Implementation

- For every frame being streamed in, a horizontal and vertical projection of the number of white pixels is counted
- Using the projection, a decision can be made on where to split the frame into segmented images
- OCR can then be implemented on each segment

(a) Image 1



(b) Vertical projection

(c) Projection segmentation

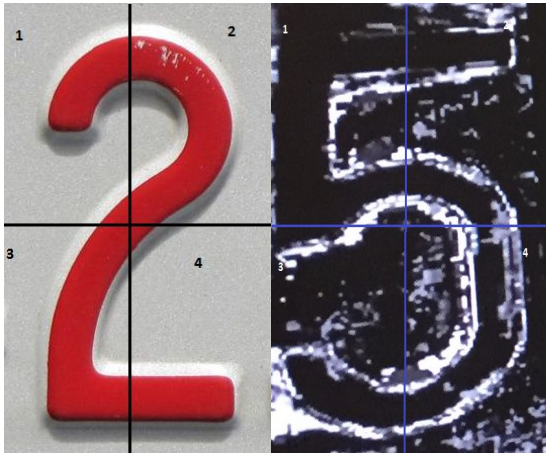
Character Segmentation - Simplified Implementation

- Due to time constraints we implemented a simplified character segmentation:
    - The license plate was assumed to be a fixed size and distance from the camera.
    - Pixels inside of the segments were retained while pixels outside the segments were made black.

Optical Character Recognition (OCR)

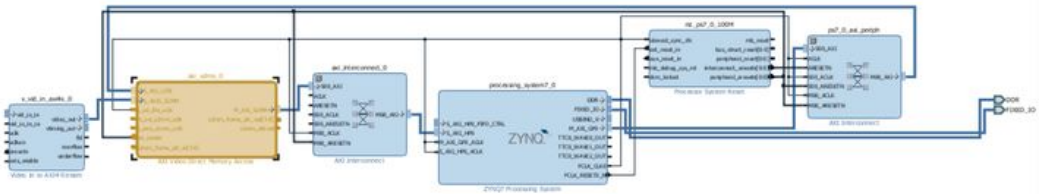
- Currently implemented a quadrant system
  - Add up number of white pixels in each quadrant, and compare them to the pixel count we expect in each quadrant for a specific number.
  - Working...not so well. See video.

$$r = \frac{\sum_m \sum_n \left(A_{mn} - \bar{A}\right)\left(B_{mn} - \bar{B}\right)}{\sqrt{\left(\sum_m \sum_n \left(A_{mn} - \bar{A}\right)^2\right)\left(\sum_m \sum_n \left(B_{mn} - \bar{B}\right)^2\right)}}$$



Continuing Efforts

- Improving OCR for the demonstration on 12/12.
  - Piping segmented frame to CPU for recognition.
- Get second filter working



Design Tradeoffs

- Some loss of complexity
  - Real time segmentations -> hard coded segmentation
- Loss of generalization
  - Plate needs to be of certain size and distance from camera, only numbers recognizable
- Gain of basic functionality of design, proof of concept

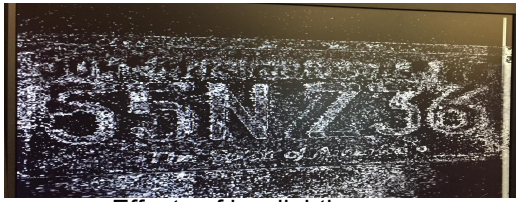
Characterization (Timing, Area)

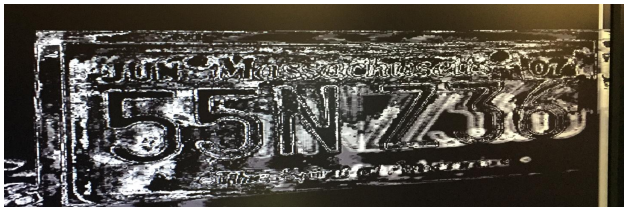
- Minimum period: 18.322ns (Max Freq. 53.579MHz)
- 944 Registers used (~1% utilization)
- 1,373 LUTs used (~2% utilization)
- # RAMB36E1(block ram)/FIFO36E1s used 103 (~73%)

Challenges

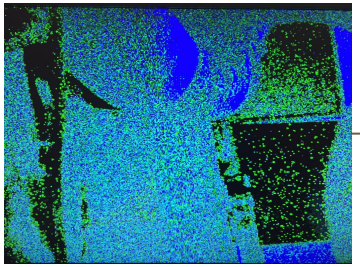
- **Noise**
  - Lighting has a large impact on noise in windowed frames.
  - Noise makes comparison in OCR difficult due to discrepancies between sampled data and reference data
- **Timing** - making sure everything synced correctly with VGA
  - Hsync/Vsync/Active Area
- **Time**
  - Would like to expand on certain features (segmentation, OCR), but difficult to implement a complex design in time allowed
  - Board breakdown

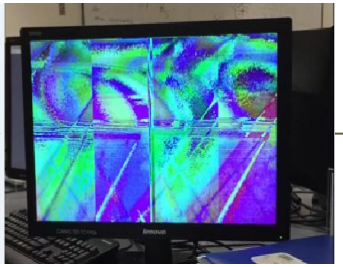
Ideal Matlab edge detection (Laplacian Kernel)

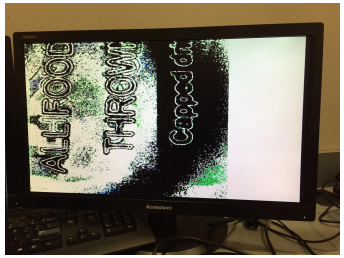

Effects of low lighting


Effects of direct lighting

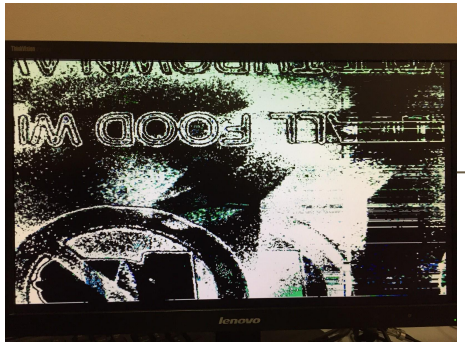
Successes

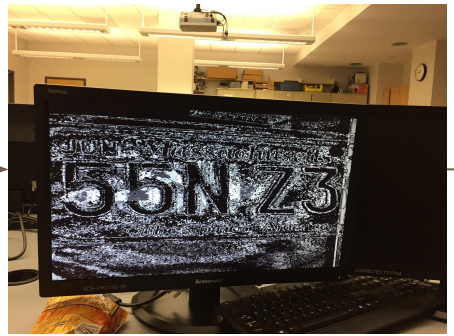


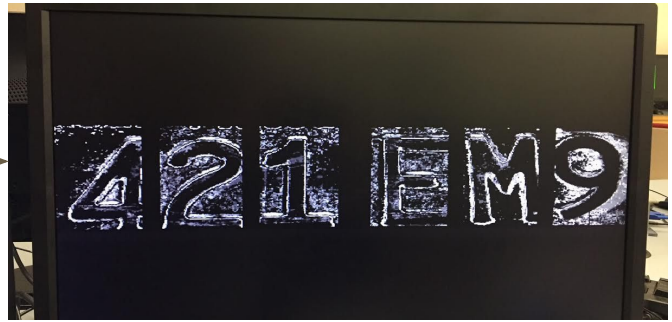




- Frame data streamed from camera and displayed on VGA.
  - Fixed most timing issues
- Window module implementing edge detection.
  - Experimented with a variety of filters to find best results for our implementation.
- Basic segmentation
- Recognition infrastructure

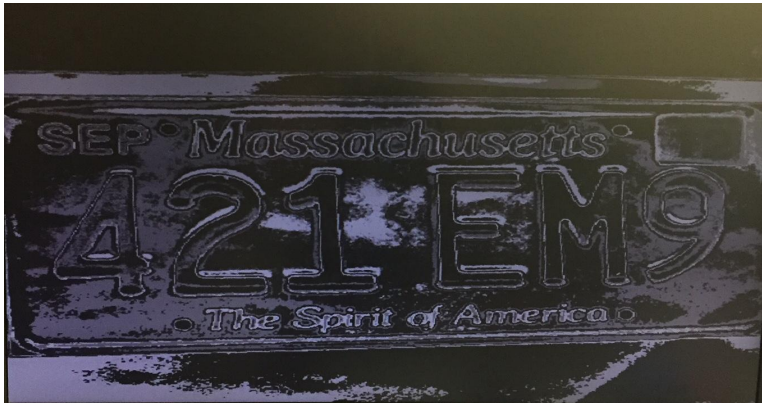




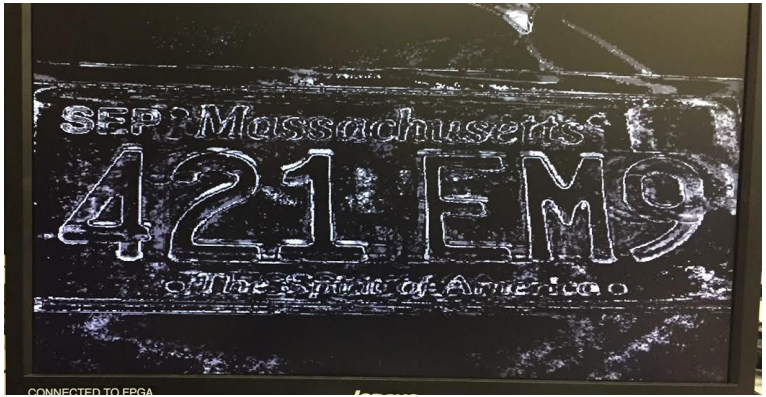


Failures

- Edge detected image very noisy.
- Dynamic character segmentation.
- Working OCR on FPGA.



Grayscale identity output



Edge detection output

Video Demonstration

https://drive.google.com/drive/folders/0B0UC3LLTtkG2cVFLYTJyVndhSnc

Work Split

- Camera to VGA - Joe, Patrick
- Window Module - Darin, Jack, Patrick
  - UART Testing
- Segmenting Module - Ravi, Joe, Patrick, Darin, Jack
  - MATLAB testing
- OCR Attempt - Joe, Darin, Ravi
- Memory Integration Attempt - Jack
  - Block and DDR
- CPU Integration Attempt - Ravi

Overall, we felt the workload was split evenly.

Thank you!