REAL-TIME FEATURE DETECTION

6.111 Final Project

Somani Patnaik

Jon Losh

Dember Giraldez

12/10/2009

1. Introduction

Our group wanted to explore application of the FPGA for image processing. We also wanted to do image processing in real-time, so we chose to implement feature detection. We designed a platform that intended to do histogram equalization (add contrast), edge detection and corner detection. The reason we wanted to explore this application is that it the computations are very repetitive for each pixel, but demand a lot of resources, so the FPGA is a good platform for them. Software implementations required multiple “for-loops” and a lot of memory to carry out the computations, so we wanted to attempt it in hardware. We also liked that the platform is modular, with the camera input going to memory and then feeding into the different image processing modules, which then output to memory. Finally the display logic reads from memory and we get our desired image.

Initially, we wanted to attempt to also have an application of these image processing modules, and one of them is face detection. However, we found that face detection requires an enormous amount of training. Some of the methods we looked into required training with data sets of over 10,000 images of faces, so we decided it was beyond the scope of the class.

Table of Contents

1. Introduction 2
2. Summary 5
3. Scan logic, ZBT interface and display (Jon) 6
4. Histogram Equalizer (Somani) 9
5. Edge and Corner Detectors (Dember) 11
6. Difficulties 22
7. Appendix 23

List of Figures

1. Block diagram for feature detection 6
2. Scan logic 7
3. Block diagram for histogram 10
4. Pixel window and Gx and Gy kernels 13
5. Sobel Pixel Flow 15
6. 3 x 6 Local Window 15
7. Sliding Local Window 16
8. Original Image of Jon 16
9. After applying actual Sobel 17
10. After applying threshold Sobel 17
11. Harris Required Window 18
12. Harris Data Flow 19
13. Harris Corner Module 21
14. Harris Result 21

2. *Summary*

The primary problem that this project was trying to address is real time image processing. In order to do that we read an image from the NTSC Camera and use an NTSC decoder to parse the camera data to give the YCrCb values. Currently, we were processing video only in grayscale. For this only the Y value is passed on to the rest of the modules. As the Y values are passed on from the decoder, they are written to ZBT1. To store data into the 36 bits wide ZBTs, four pixel values are written at each address in the ZBT. Each of these pixels is 8-bits wide. the addressing in the ZBT is based on the location of the pixels, which helps us keep track of the position of the pixel in the image.

The pixels from the NTSC decoder are also read in parallel by the Histogram equalizer module to make the histogram for each frame. The second part of the histogram equalizer reads the data out from ZBT1, looks it up in the histogram equalized LUT and passes it on to the Sobel Edge detector and Harris Corner detector modules. The histogram equalizer follows a scan logic which gets the data from the ZBT1 in a way that the Sobel Edge Detector and Harris Corner detector can get useful data as soon as possible for their processing. Both these modules maintain their separate local windows of pixels, that they need to calculate of a given pixel is an edge or a corner.

The Sobel edge detector maintains a 3x6 window. It uses this window to calculate if the 4 pixels in the middle of the window are edges. It then outputs the different set of four pixels depending on whether the pixel was an edge or not. It also stores of the address of these pixels from the first ZBT and passes it along with the new pixel values. This keeps track of the image location through the processing. Thus the Sobel Edge has a 32-bit output for four pixel values and an address with the pixel values for the location.

The Harris corner detector maintains an 5x8 local window to calculate if the middle four pixels are corners or not. Unlike the Sobel edge detector, the Harris Corner detector uses the pixel values to find corners but outputs the same values for the pixels. It uses extra four bits to mark if each of these four pixel values is a corner. As in the Sobel Edge detector, it keeps track of the addressing of the pixels and passes it along with pixel output. Thus, the output from the Harris corner detector is 36 bits, containing the four original pixel values 8 bits each and 4 bits to mark the corners and an address for the pixels.

Both Sobel Edge Detector and Harris corner detector write their outputs to ZBT2 from which the display logic gets the pixel values for displaying. The display logic uses the horizontal and vertical locations on the screen to read the values from the ZBT2, parses it and displays it to on the screen. The block diagram below gives the data flow through the entire system.

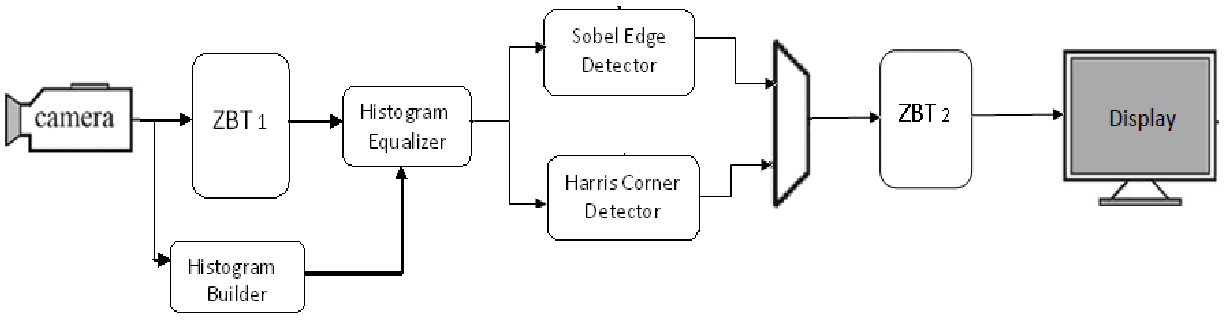


Figure 1. Block Diagram for the Feature Detection

3. Scan Logic, ZBT interfacing and Display Logic (Jon)

Scan logic:

The scanning logic is embedded inside of the histogram equalizer module. It uses two internal counters to keep track of where it's requesting data from in the image from the 1st ZBT, row_counter (9 bits) and column_counter (10 bits). It turns the values of these counters into a 19-bit address, zbt_addr, that it sends to the 1st ZBT to request data. To process a frame of video before the next one comes, the scan logic requests a read from the 1st ZBT every four clock cycles by incrementing a 2-bit counter, read_count, every clock cycle and requesting a read when it is zero. The module scans vertically in a two address wide column, incrementing row_counter every two reads and alternately incrementing and decrementing column_counter every read. At the bottom of a column, the scanner jumps back to the top, making the previous address in the right column the address in the left column. A picture of how the scanning proceeds is shown in figure 2:

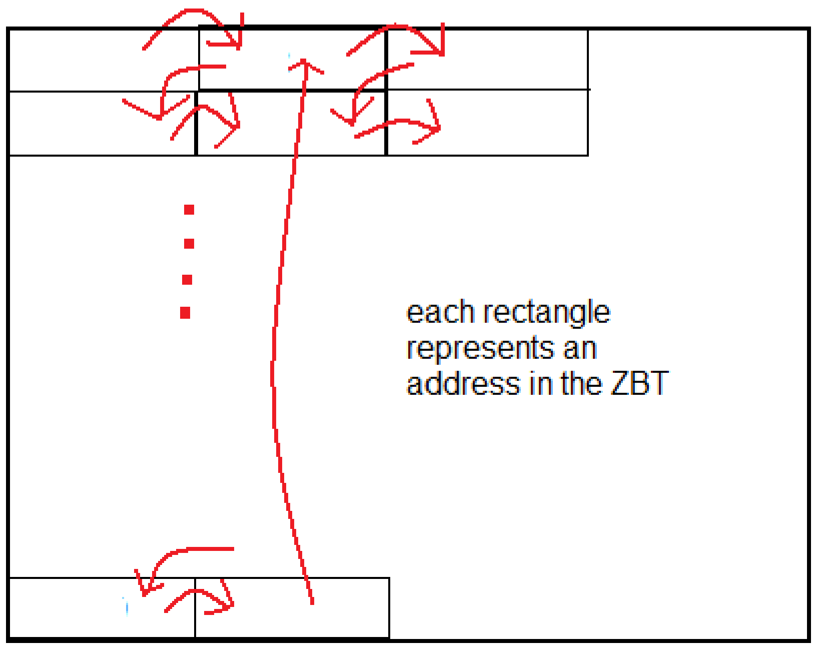


Figure 2. The scan logic starts in the upper-left hand corner and follows the red arrows. Note that consecutive columns overlap.

The reason we scan vertically is because the feature detection modules require square windows of pixels as inputs. Since the NTSC camera is outputting pixel data in a horizontal scan, it ends up getting stored in four-wide, one-tall blocks in the 1st ZBT. Scanning vertically allows the Sobel and Harris modules to keep internal circular buffers that reuse the maximum amount of pixel data, keeping the amount of data processed per ZBT access relatively high. Also, the reason that consecutive columns overlap is because the feature detection modules do not output data for the pixels on the edges of the windows they process, so we need to feed more than a frame's worth of data into them to process the whole frame.

This scanning logic is somewhat wasteful in that it reads from most of the addresses twice to get data for a single frame. The tradeoff is that it reduces the amount of internal memory that the Sobel edge detector and Harris corner detector need to have and also the number of individual feature detection modules needed inside of the top-level Sobel and Harris modules. This reduction means less gate area used, faster compile times and quicker development. Also, we already have more than enough time to process the image as is, so there is little to be gained from a change in scanning logic.

To account for the two cycle latency of the ZBT, the 36-bit data bus from the 1st ZBT, zbt_val, gets internally latched two cycles after a read request is made. The lower 32 bits of this word are four 8-bit greyscale pixel values from the image in a four-wide, one-tall block, with the highest eight bits corresponding to the pixel on the far left. These pixel values are passed on in a 32-bit bus, four_pixel, to the Sobel edge detctor and Harris corner detector modules. The row and column values denoting where these pixels are in the image are pipelined so that they show up at the output at the same time as the pixel data. These are also passed on to Sobel and Harris, as they use them to calculate where to write their outputs in the 2nd ZBT.

ZBT interfacing:

Although we started out working with stock code for interfacing with the 1st ZBT, we had to wire up a second one ourselves. We copied the approach that the writers of the NTSC camera code took to handling conflicts over reads and writes: when one module wants to write to a ZBT and another wants to read, the one that wants to read takes priority. What this results in is some pixels failing to be updated until the next frame, which often goes unnoticed. Giving writes priority resulted in unexpected data being passed to the reading module, creating excessive noise in the display.

A distinct improvement we made was setting up the ram clocks properly. In the stock code, the ZBT's are driven with an inverted system clock. This pushes back the ram clocks by half a clock cycle, giving them a bigger window to meet their setup times. However, inverting the clock results in an inherently glitchy signal in the FPGA since it is done with lookup tables, so this caused the ZBT's to fail unexpectedly at random times. We addressed this issue by hooking up the ram clocks to outputs from the digital clock manager, which generates a properly phase-shifted version of the system clock with no phase shifting or unintended skew.

The addressing convention for both ZBT's is identical. Given a pixel in a row and column in the image, it will be located at {2'b00, row[8:0], column[9:2]}, where row and column are expressed in binary. The bottom two bits of the column are ignored because pixels are stored in 4x1 groups at a given address. The data storage convention is slightly different between the two modules. For the 1st ZBT, the highest four bits are empty, and then the remaining 32 bits hold four 8-bit greyscale pixel values, with the highest bits corresponding to the leftmost pixel in the block. So a 36-bit word in the 1st ZBT contains this:

{4'b0, [8-bit grayscale pixel value], [8-bit grayscale pixel value], [8-bit grayscale pixel value], [8-bit grayscale pixel value]}

The second ZBT's data storage looks like this:

{[8-bit grayscale pixel value], c1, [8-bit grayscale pixel value], c2, [8-bit grayscale pixel value], c3, [8-bit grayscale pixel value], c4}

c1-c4 are one-bit values that indicate whether the pixel before them is a corner or not, as determined by the Harris corner detector module.

Display logic:

The display logic is largely the same as in the stock NTSC code. However, we made one modification that overlays red pixels on corners that the Harris corner detector module finds. If any pixel in the current block of four pixels the display is processing is marked as a corner, the module raises its corner output high. The corner wire controls a mux that passes through the grayscale value as normal when it is low and outputs a red pixel when it is high. The reasoning behind labeling blocks of four pixels as corners rather than individual pixels was that a single red pixel would be hard to see on the display.

4. *HISTOGRAM EQUALISER MODULE (Somani):*

This module takes a 640x480 image as input and outputs a histogram equalized image of the same size as output. This helps to increase the contrast in the image so that the system works well in different lighting conditions.

This module has 3 stages of operation:

- Histogram builder,
- Histogram equalizer
- Image builder

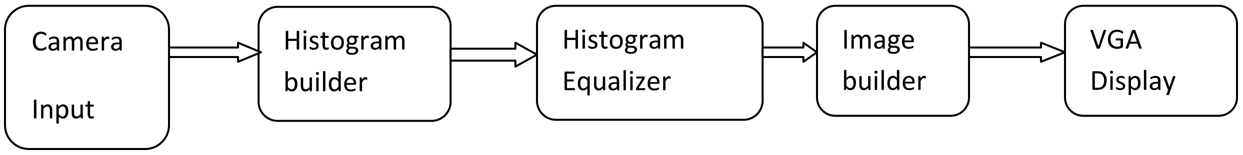


Figure 3. Block diagram for the Histogram Equalizer

Histogram builder:

In this stage the intensity histogram statistics for the image is calculated. The histogram builder counts the number of times a particular intensity appears in the image. There are 256 intensity values in the image. These values are classified into 256 bins one for each intensity value. Let this number be stored in counters n(k) where k is the intensity value of the pixel, k varies from 0 to 255. This is a sequential process which takes a long time.

Histogram Equalizer:

In this operation the n(k) values are normalized. This is done by calculating the cumulative density function (CDF) of the intensity values. The CDF value of n(k) (C(k)) is the summation of all the previous n(k) values and then it is normalized by scaling this with the number of pixels in the image.

$$\mathrm{C(k)} = \frac{(C(k) - \min val) * 255}{p * q - \min val}$$

where, p*q: 640*480 in this case, and
min_val : the number of pixels at the lowest intensity value that is in the frame

Image Builder:

Intensity of each pixel in the new image is the CDF of intensity value of the old image at that point scaled by 256. This increases the contrast in the image by having all the 256 gray scale values in the image.

$$\mathrm{K\_new(i,j)} = C(k) \;;$$

HARDWARE IMPLEMENTATION

The histogram builder is very slow as it is a sequential process that goes through every single pixel in the image to classify it. In order to do it faster and work with real time video, the trick used was to make the Look-Up-Table (LUT of intensity values) from one image and use it for the next one. This way we avoid waiting for the entire image before looking it up.

The module kept track of the frame parity (frame here refers to the entire 640x480 image, not one part of interlaced video). It maintains two arrays of 256 20-bit registers called incrementors. Depending on the frame parity one of the two arrays is used to count the incoming pixels. As the pixel values come from the camera, each time there is a valid input, the register containing the number of pixels at that greyscale value for that frame is incremented.

At the same time the other array that has the histogram of the previous image is used to calculate the CDF. Iterating through this array, the values at each location is updated to be the sum of all the previous values in the array. This calculates the CDF for that frame.

The hard part in the implementation is the divider algorithm. Initially, a coregen divider module was used to do the division but a 29-bit wide divider along with the register arrays used for the incrementor and LUT had a huge compile time with Xilinx. To avoid this a lot of approximations were used to do the division. In general the *min_val*, for any real time image frame is a very small number, so in comparison to the total size of the image it can be ignored. Also the scaling, instead of going from 0-255, it was done for 0-256. This made the divisor a constant. It was 640*480/256 = 1200 in this case. Thus, the CDF value was just divided with 1200 to generate the look up table. Even this division was a challenge. So instead division by 1200 was approximated by to right shifting the dividend bits by 13 and subtracting it from right shifted dividend bits by 10. Only the top 8 bits were considered for the LUT.

This unit also used the scan window logic to read the set of pixel values from the ZBT and use the LUT and pass the new set of values to the Sobel edge detector and Harris corner modules.

Tricks in the design/ scope for improvement:

Using two BRAMs instead of the arrays for the histogram incrementor module would make the logic much faster and have less design constraint.

Using the LUT from the previous image for the values of the next image helps get around the timing constraints of real time videos.

Reading the data in straight from the camera, helps avoid the need of ZBT double buffers for the other processing units.

Any trickier way of implementing the division would also help improve the design and reduce the compile time.

This is also a good place to do clock domain crossing between the camera clock and the display clock as the input can be at the camera clock while the look up is at system clock.

5. Edge and Corner Detectors (Dember)

Sobel Edge Detector

A Sobel edge detector is used to find the edges in an image. It works by convolving a 3x3 window of pixels with two horizontal and vertical filters in order to find the sobel edge value of the center pixel in that 3x3 window. The result of applying the two filters is a gradient of image intensity at that point. By measuring the gradient, the objective is to measure change in the intensity of the pixel at that point. If the changes is little or non-existent, then we most likely do not have an edge. If the change is significant, then we most-likely have an edge.

Process plus Design Decisions:

The main challenge for running the edge detector came from the fact that it was in real time, giving us limited time to process a frame. Also, because edge detection for a pixel requires all the pixels surrounding it, we needed to store this information in memory first. This is why we decided to implement the local window. Initially, we had intended to have multiple instantiations calculating gradients for many pixels at a time, but the main restriction was that we could only load 4 pixels at a time from memory. Our initial design consisted of a 4 x 4 local window, in which we would process pixels (1,1),(1,2),(2,1) and (2,2), but we want into problems with our sliding window logic. When the local window got to the bottom, it would slide 4 columns to the right and then start processing the pixels. However, we weren't processing the outer pixels in the window, so in the end we were only processing 50% of all the pixels. The solution we came up with was to modify our window size to 3 x 6, and load the first four pixels in one read, followed by the remaining two. We still processed the middle four pixels at a time and when we got to the bottom, we still shifted our window 4 columns to the right. This time however, since our window was 6 columns wide, we did not skip any pixels

Another problem we had was getting the display module working. This was particularly challenging, for many reasons. Initially we were running the memory at a different clock than the rest of our modules, so this caused problems. Also, it was tricky to fine tune our addressing convention until it worked. When the sobel_top module got new data, it had to keep the row and column information associated with that data throughout the processing and then write a combination of the as the two as the memory address for the data in memory. It does not sound as complicated when I outline, but it was in practice.

Something I would have done differently is not counting on having a display for debugging. The display module or writing to the second memory (where my module reads from) didn't work until Sunday night, so we only had two days to integrate and debug the sobel_top and harris_top modules. If I had to do this project again, I would try to come up with a way to simulate camera input, so that I can go ahead and test my modules without the camera. We attempted to simulate data at one point, but trying to create the data that a video camera would output for a given image was very difficult.

Now I will describe some of the modules implemented.

Sobel Module

Input [7:0] p0,p1,p2,p3,p5,p6,p7,p8 //8-bit pixels values
Input sw //corresponds to switch[3] for determining sobel mode (see below)

Output: [7:0] out // sobel edge value

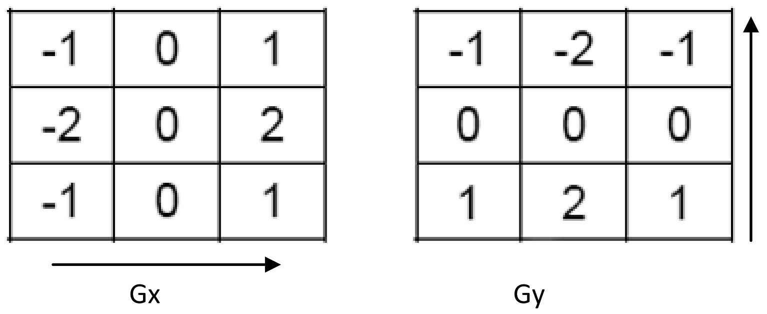
Clock: Pixel clock 25MHZ

The sobel module takes a 3 x 3 window of pixels and calculates the sobel edge value for the pixel at the center of the window. It convolves the window with the following two gradient kernels in the time domain:

P0	P1	P2
P3	P	P5
P6	P7	P8

3 x 3 pixel window

$$*$$



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

Gx

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

Gy

Figure 4: Pixel window and Gx and Gy kernels

The calculations to obtain the gradients are the following two:

Gx=((p2-p0)+((p5-p3)*2)+(p8-p6))

Gy=((p0-p6)+((p1-p7)*2)+(p2-p8))

After calculating Gx and Gy, the module takes the magnitude of these gradients. The output is truncated to 8 bits so it does not go above 255, which is the maximum value in our grayscale corresponding to the color white. To truncate it to 8 bits, the module OR's together bits 10, 9 and 8; if the result is 1, then it outputs 8'hFF, otherwise it outputs the actual value.

The following diagram shows the whole calculation. One thing to note is the bit shift, since it is equivalent to multiplying by two.

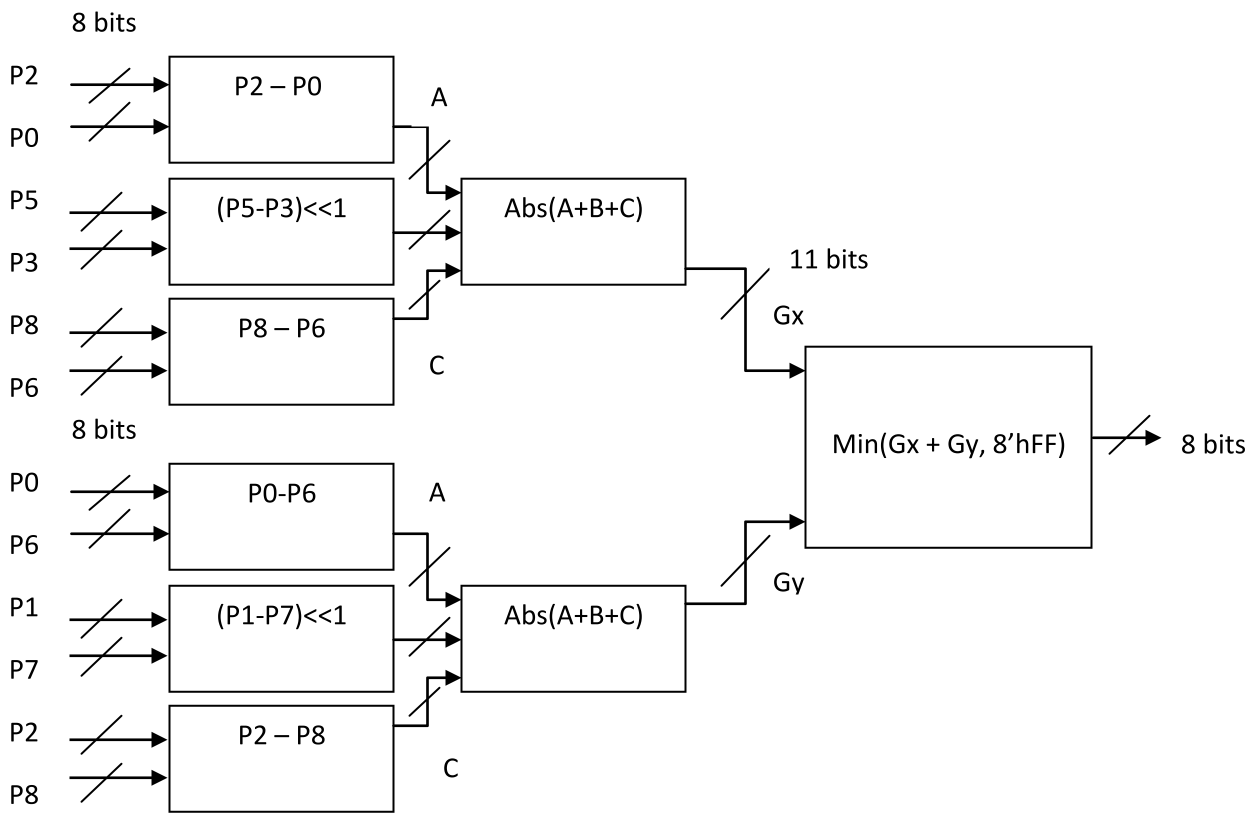


Figure 5. Sobel Pixel Flow

The output of the sobel has two modes: one corresponds to the sobel algorithm, which is the gradient intensity in grayscale; the other compares this value with a threshold, and outputs edges above the threshold in black and everything else in white, resulting in a "sketchpad" representation of the image. This feature is something we decided to add at the end, in order to vary the output and see what would happen, and we liked it. The two output modes are controlled by switch [3] on the FPGA.

Sobel_Top Module

The Sobel_Top Module manages the local 3 x 6 window of pixels that is loaded from memory.

It takes in the following inputs:

input [31:0] new_pixels

i**nput** new_data
input [8:0] row
input [9:0] column
input win
input sw
input bypass_sw

output reg [35:0] sobel_out,
output we
output [18:0] zbt_address

clock: pixel 25MHZ

The Soble_Top Module takes in 4 new 8-byte pixels in grayscale whenever the new_data signal goes high. It also takes the row and column information for the left-most pixel in the 4-pixel cluster. The row and column information will provide the location information for those four pixels, so they can later be displayed on the screen in the proper location.

On the new_data signal, the following 3 x 6 window of pixels is populated. Since the module can only take four pixels at a time, the win flag is used to determine whether the pixels correspond to columns 1-4 (win == 0) or columns 5,6 (win == 1). The window is stored in a register array and the reason it is 3 x 6 is that the module can output at most four results (since it can only write 36 bits to memory and each pixel needs 8 bits) and each of the sobel edges needs the surrounding pixels for gradient calculations. The four pixels that that have a sobel output for each window are shaded in gray.

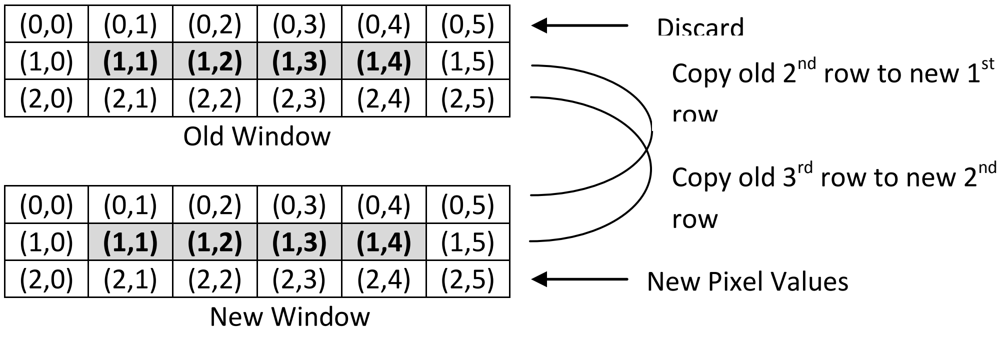
Win == 0 | win == 1

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	**(1,1)**	**(1,2)**	**(1,3)**	**(1,4)**	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)

Figure 6: 3 x 6 Local Window

When the window has been populated, a window_ready flag is used to signal that the 4-middle pixels can be processed. Window_ready along with new_data provide two flags for the module's synchronization. The expected throughput is that new_data will be provided every two cycles and that the module will output also every two cycles, on the cycle in which new_data is not being loaded. Also, address is 19-bits wide and the convention used is {2'b0, row, column[9:2]}. Also, since we won't be outputting on the next cycle, but in two-cycles, we latch both the window_ready signal, which we also use for the write enable (we) signal, and the memory address. This way we output two cycles after getting the necessary data.

When new_data arrives again, the window slides down in the following manner:



(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	**(1,1)**	**(1,2)**	**(1,3)**	**(1,4)**	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)

Old Window

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	**(1,1)**	**(1,2)**	**(1,3)**	**(1,4)**	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)

New Window

Discard
Copy old 2nd row to new 1st row
Copy old 3rd row to new 2nd row
New Pixel Values

Figure 7: Sliding local window

Finally, on top of the switch that is used to determine the sobel mode of operation (edge intensity or threshold), another switch allows the module to show the original image. When the bypass switch is on, the module simply outputs the unprocessed four pixels from the window, resulting in the original image being written to memory and being displayed.

These are some results from the Sobel Edge Detector:

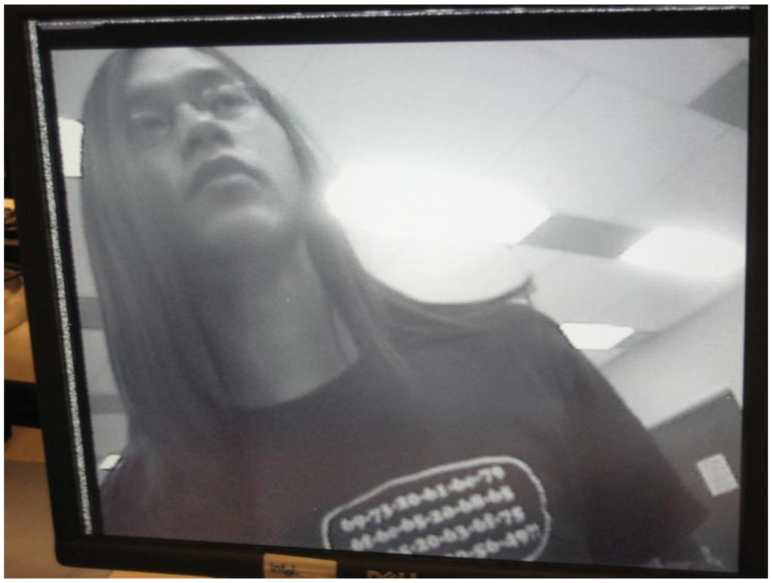


Figure 8: Original Image of Jon

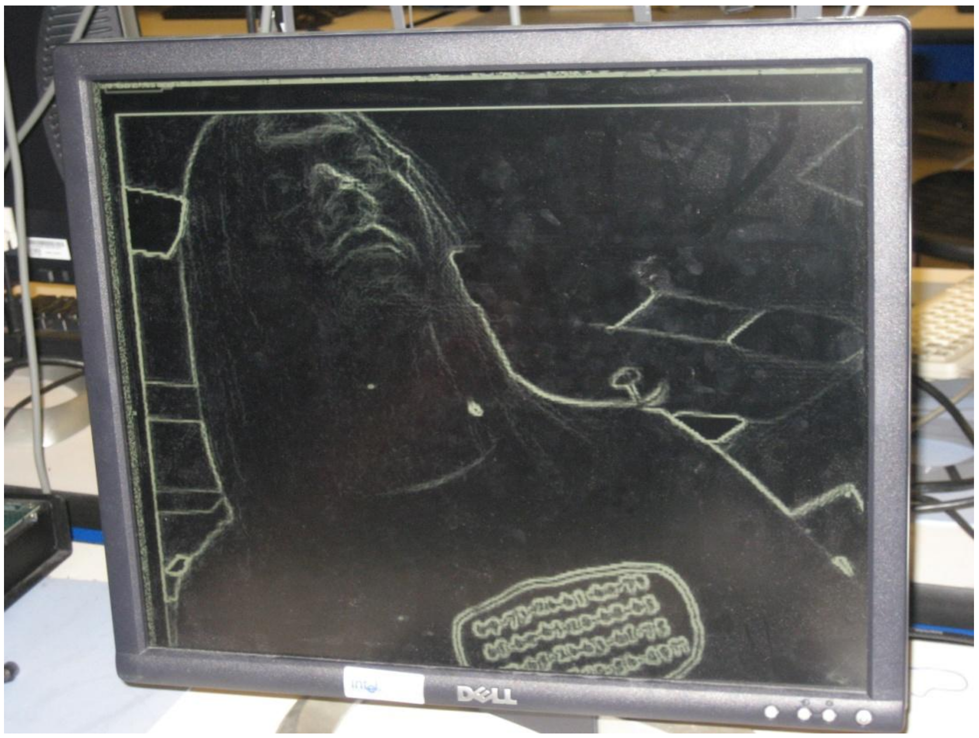


Figure 9: Image of Jon after applying Sobel filter that reflects that strength of the edge

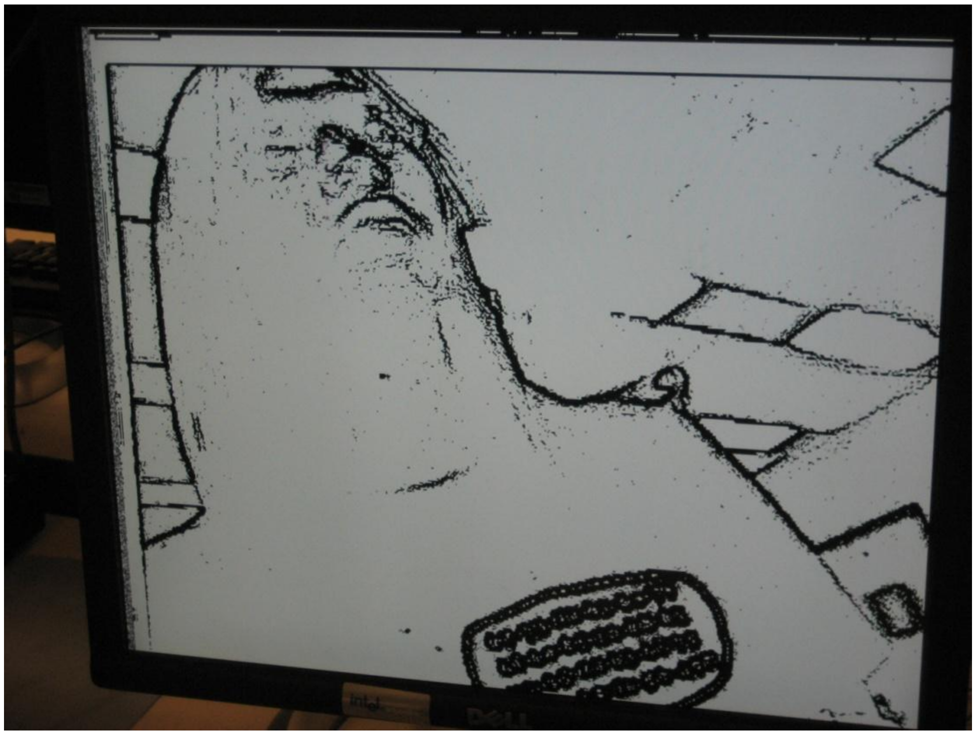


Figure 10: image of Jon that shows the Sobel filter that outputs white for a non-edge and black for an edge

Harris Corner Detection

Overview

The harris corner detector uses both x and y gradients in order to determine whether a pixel is a corner or not. To be a corner, then there has to be significant change in both directions. However, the harris corner algorithm calculates the likelihood that a pixel is a corner from summing all the surrounding gradients around a pixel and performing some calculations to determine its likelihood of being a corner.

In order to extract the corners, the harris corner detector needs a 5 x 5 window in order to compute a score for the pixel at location (2,2). The window size requirement comes from the fact that we need to calculate the surrounding gradients, and to calculate those gradients we need the pixels around those.

The following picture illustrates the need for a 5 x 5 window by specifying why each pixel is needed:

Req. for gradient	Req. for gradient	Req. for gradient	Req. for gradient	Req. for gradient
Req. for gradient	Gradient	Gradient	Gradient	Req. for gradient
Req. for gradient	Gradient	**Pixel of interest**	Gradient	Req. for gradient
Req. for gradient	Gradient	Gradient	Gradient	Req. for gradient
Req. for gradient	Req. for gradient	Req. for gradient	Req. for gradient	Req. for gradient

Figure 11: Harris required window

The following four steps are required to process the image:

1) Convolve image with horizontal and vertical differential operator to obtain gradients Gx and Gy.
2) Generate the three summations necessary from Ix and Iy to from the Harris 2x2 Window (see diagram).
3) Compute the determinant and trace to come up with a value for the likelihood that the current pixel is a corner.
4) Compare with a threshold to determine if it is a corner. If it is, output a “1” attached at the end of the 8-bit pixel value written to memory, so then an overlay can read this bit and mark that pixel as a corner.

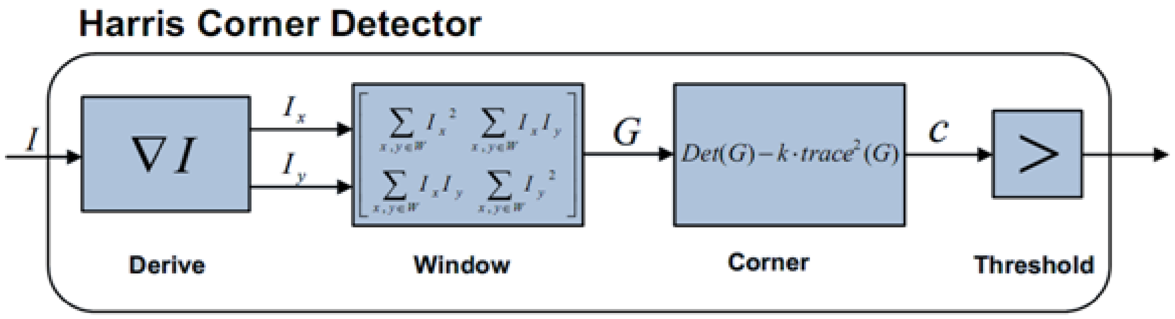

Harris Corner Detector
I
∇I
I_x
I_y
$\sum_{x,y\in W} I_x^2$ $\sum_{x,y\in W} I_x I_y$ $\sum_{x,y\in W} I_x I_y$ $\sum_{x,y\in W} I_y^2$
G
$Det(G)-k\cdot trace^2(G)$
c
$>$
Derive
Window
Corner
Threshold

Figure 12: Harris-Corner Data Flow

Module: Corner_Top

The Corner_Top module takes the following input (similar to Sobel_Top):

input [31:0] new_pixels
input new_data
input [8:0] row
input [9:0] column
input win

output reg [35:0] harris_out,
output we
output [18:0] zbt_address

Clock: Pixel

This module is very similar to Sobel_Top, but it uses a 5 x 8 local window, due to the specific requirement by the Harris Corner algorithm of a 5 x 5 window. 8 columns allow us to calculate corner values for 4 pixels per window. The Corner_Top module instantiates 18 Derive_pixel modules in order to calculate 36 gradients per window (18 in the x-direction (Gx) and 18 in the y-direction (Gy)). Window_ready is used for synchronization – all the derive pixel modules are wired, but the output is only read once the window_ready signal is high.

On the rising edge of the clock and once the windo_ready signal is high, it calculates:

$$\sum Ix^2$$
$$\sum Iy^2$$
$$\sum IxIy$$

For each one of the four pixels of interest to form the "Harris Window" as shown in the Harris diagram. Each sum is 20 bits wide, which represents their maximum possible value. These sums are all connected to the four corner modules, which will output the "corner" value for each Harris Matrix.

Finally, the output of the Corner_module is compared against a threshold and the result is appended in locations 27, 19, 8 and 0 of the harris_out, with the other pixels being the original image. 1 represets that the preceding pixel is a corner pixel, while zero represents it is not. This information is used to produce a red overlay for that pixel if it is a corner, so we are able to see corners on the screen. The output is assigned in the following manner on the rising edge of the clock (th is a threshold that we set experimentally for the harris corner detector:

```
harris_out <= {local_window[2][2], (c_out1 > th) ? 1'b1:1'b0, local_window[2][3], (c_out2  > th)
? 1'b1:1'b0, local_window[2][4], (c_out3  > th) ? 1'b1:1'b0, local_window[2][5], (c_out4  > th) ?
1'b1:1'b0};
```

Module: Derive Pixel

Input [7:0] p0,p1,p2,p3,p5,p6,p7,p8 //8-bit pixels values

Output [10:0] abs_gx
Output [10:0] abs_gy

This module computes the x and y gradient for a given pixel and requires the 8 surrounding pixels. It is very similar to the Sobel_Module, because they both calculate gradients. The difference is that Sobel_Module outputs an approximation of the magnitude of the gradients, whereas as this module outputs the absolute value of each of the x and y gradients.

Module: Corner

input [11:0] xx_sum
input [11:0] yy_sum
input [11:0] xy_sum

output [23:0] c_out

This module computes the determinant and trace of the Harris Matrix and outputs "the determinant – k * trace^2". K is a constant of 1/8 needed for this calculation. Using the all the sums, computation becomes the following:

$$c(x,y) = \sum_{x_k,y_k \in W} I_x^2 \sum_{x_k,y_k \in W} I_y^2 - \left(\sum_{x_k,y_k \in W} I_y I_y \right)^2 - k \left(\sum_{x_k,y_k \in W} I_x^2 + \sum_{x_k,y_k \in W} I_y^2 \right)^2$$

Figure: C value computation

Also, because this is the most computationally intensive calculation due in part to the width of each bus, the module carries out the calculation in a pipelined manner using 3 stages. The following diagram shows the pipelined calculation of C.

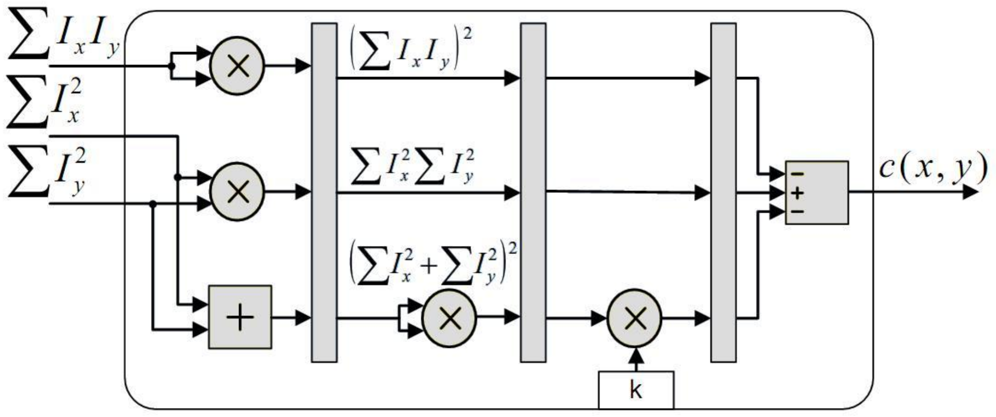


Figure 13: Pipelined Corner module

Harris Corner Result

The output from the harris corner wasn't what we expected. It was very sensitive, determining too many pixels to be corners. When we analyzed the intermediate values in the logic analyzer, we found that the sums of gradients squared were being computed correctly, but between then and the final C value something was not working out. The following is the display that the harris module was giving us:

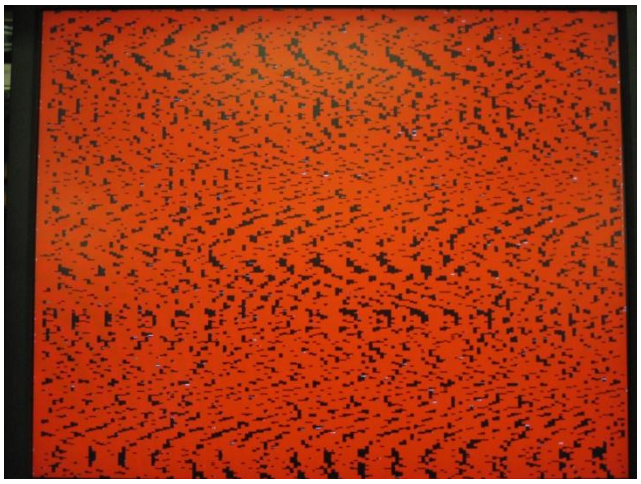


Figure 14: Harris output showing too many "corner" values (pixels in red)

6. Difficulties

It took us a long time for us to realize that crossing from the 27 MHz camera clock domain to the 25 MHz clock domain was best done at the 1st ZBT. The 2nd ZBT was being written to and read to very rapidly, and doing a clock domain change at the point made matters even worse.

We also had a very inefficient way of writing to the 2nd ZBT for a very long time that we kept in an attempt to keep the scanning logic neat. However, we found that making the scanning logic a little messier allowed us to store pixels in more or less the same way as they were stored in the 1st ZBT, which made tracking data and debugging with test patterns much, much easier.

Finding the problem with the ram clocks being inverted rather than properly set up with the digital clock manager took us a long time. Future students should be made aware that such a bug exists in the stock NTSC camera code.

There were many other small bugs along the way, but overcoming each of these problems resulted in huge leaps forward.

7. Appendix

The following are some of our modules. We only included a few, because most are very difficult to read in word format, and are better read from the submission of the verilog files on the class website.

```
//////////////////////////////////////////////////////////////////////////////////
// The sobel module takes a 3 x 3 window of pixels and calculates the sobel edge
// value for the center pixel
//////////////////////////////////////////////////////////////////////////////////
module sobel( p0, p1, p2, p3, p5, p6, p7, p8, out, sw);

        input  [7:0] p0,p1,p2,p3,p5,p6,p7,p8;     // 8 bit pixels inputs
        input sw; //switch controlling threshold mode vs edge mode
        output [7:0] out;                                   // 8 bit output pixel

        wire signed [10:0] gx,gy;    //11 bits because max value of gx and gy is
        //255*4 and last bit for sign
        wire signed [10:0] abs_gx,abs_gy;         //it is used to find the absolute value of gx and gy
        wire [10:0] sum;                          //the max value is 255*8. here no sign bit needed.

        assign gx=((p2-p0)+((p5-p3)<<1)+(p8-p6));//sobel mask for gradient in horiz. direction
        assign gy=((p0-p6)+((p1-p7)<<1)+(p2-p8));//sobel mask for gradient in vertical direction

        assign abs_gx = (gx[10]? ~gx+1 : gx);      // to find the absolute value of gx.
        assign abs_gy = (gy[10]? ~gy+1 : gy);      // to find the absolute value of gy.

        assign sum = (abs_gx+abs_gy);                       // finding the sum
        assign out = sw ? ((sum > 50) ? 0 : 8'hff):
                                        ((|sum[10:8])?8'hff : sum[7:0]);
        //assign out = (|sum[10:8])?8'hff : sum[7:0];       // to limit the max value to 255

Endmodule

//////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////
// sobel_top: Top Level Module for sobel operator. The sobel operator takes the 3x6 local window of
pixels and calculates sobel edge values
// for the four pixels at locations (1,1), (1,2), (1,3) and (1,4).
//////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////
Module sobel_stop(
        input clk,
        input reset,
        input [31:0] new_pixels, //32-bit input consisting of the current 4 8-bit pixels (in grayscale)
        input new_data, //flag signaling 4 new pixels are ready
```

```
	input [8:0] row, //row for current pixel
	input [9:0] column, //column for current pixel
	input win, //flag used to determine weather we are populating the left most half of the window
(1st four pixels when win == 0), or the right half otherwise (pixels 4 and 5)
	input sw, //switch controls threshold mode vs edge intensity mode for sobel output
	input bypass_sw, //switch that causes sobel to just pass through pixel values w/o edge detection

	output reg [35:0] sobel_out, //36 bit output consisting of 4 pixel bytes and 4 1-bit placeholders -
> format: { pixel[35:28], 1'b0, pixel[26:19], 1'b0, pixel[17:10]. 1'b0, pixel[8:1], 1'b0}
	output we, //low for one cycle after we output data
	output reg [18:0] zbt_address //address where the output will stored in the second ZBT.
Convention: {2'b0, row, column[9:2]};
   );

	//register array corresponding to 3x6 local window, (row, column) convention
	// reg [WIDTH-1 : 0] rf [DEPTH-1 : 0];
	reg [7:0] local_window [2:0][5:0];

	//window_ready flag that indicates our local window has been loaded with the data provided by
"new_pixels". The second one is for pipelining since we write to memory every two cycles
	reg window_ready = 0;
	reg window_ready2 = 0;

	//address in memory corresponding to the current 4 pixels. Pixel location (1,1) is used as a
reference.
	reg [18:0] win_address;

	//four sobel outputs since we are calculating values for four pixels at a time
	wire [7:0] c1_wire;
	wire [7:0] c2_wire;
	wire [7:0] c3_wire;
	wire [7:0] c4_wire;


	//four instantiations of sobel modules. Each one requires the 8 pixels around the pixel of
interest, so that the horizontal and vertical gradients can be calculated
	//module sobel( p0, p1, p2, p3, p5, p6, p7, p8, out);
	sobel s1(.p0(local_window[0][0]),.p1(local_window[0][1]),.p2(local_window[0][2]),
				.p3(local_window[1][0]),.p5(local_window[1][2]),

	.p6(local_window[2][0]),.p7(local_window[2][1]),.p8(local_window[2][2]),.out(c1_wire),
.sw(sw));

	sobel s2(.p0(local_window[0][1]),.p1(local_window[0][2]),.p2(local_window[0][3]),
				.p3(local_window[1][1]),.p5(local_window[1][3]),

	.p6(local_window[2][1]),.p7(local_window[2][2]),.p8(local_window[2][3]),.out(c2_wire), .sw(sw)
);
```

```
	sobel s3(.p0(local_window[0][2]),.p1(local_window[0][3]),.p2(local_window[0][4]),
				.p3(local_window[1][2]),.p5(local_window[1][4]),

	.p6(local_window[2][2]),.p7(local_window[2][3]),.p8(local_window[2][4]),.out(c3_wire), .sw(sw)
);

	sobel s4(.p0(local_window[0][3]),.p1(local_window[0][4]),.p2(local_window[0][5]),
				.p3(local_window[1][3]),.p5(local_window[1][5]),

	.p6(local_window[2][3]),.p7(local_window[2][4]),.p8(local_window[2][5]),.out(c4_wire), .sw(sw)
);

		always @ (posedge clk)begin

			if (reset) begin
				window_ready <= 0;
				window_ready2 <= 0;
				end

			//latch output and memory address whenever the local window is ready after
loading a new set of pixels
			if(window_ready)begin
				sobel_out <= bypass_sw ? {local_window[1][1], 1'b0,
local_window[1][2], 1'b0, local_window[1][3], 1'b0, local_window[1][4], 1'b0}:
							{c1_wire, 1'b0, c2_wire, 1'b0, c3_wire,
1'b0, c4_wire, 1'b0};
				zbt_address <= win_address;
				window_ready <= 0;
				end



			//Logic for sliding a 3x6 window of pixels down. On new_data, new pixels are
added at the bottom, the other two rows are shifted, and the top one is discarded
			//Load new data from input onto local window
			if (new_data)begin
				//if !win, then we are loading the leftmost four pixels of 6 the pixel wide
window
				if (!win) begin
					win_address <= {2'b0, row, column[9:2]};
					{local_window[2][0], local_window[2][1], local_window[2][2],
local_window[2][3]} <= new_pixels;
					{local_window[1][0], local_window[1][1], local_window[1][2],
local_window[1][3]} <= {local_window[2][0], local_window[2][1], local_window[2][2],
local_window[2][3]};
					{local_window[0][0], local_window[0][1], local_window[0][2],
local_window[0][3]} <= {local_window[1][0], local_window[1][1], local_window[1][2],
local_window[1][3]};
```

```
                                        end
                                //if win, then we are loading the rightmost two pixels of the 6 pixel wide
window. new_pixels[15:0] aren't saved at this point
                                if (win) begin
                                        {local_window[2][4], local_window[2][5]} <= new_pixels[31:16];
                                        {local_window[1][4], local_window[1][5]} <=
{local_window[2][4], local_window[2][5]};
                                        {local_window[0][4], local_window[0][5]} <=
{local_window[1][4], local_window[1][5]};
                                        window_ready <= 1;
                                        end
                                end
                        //delay window_ready
                        window_ready2 <= window_ready;
                end

                //we is high for one cycle one cycle after the window is ready
                assign we = window_ready2;
endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Modules computes the gradient of a pixel in the x and y directions using the 8
// surrounding pixel values
////////////////////////////////////////////////////////////////////////////////

module derive_pixel(p0, p1, p2, p3, p5, p6, p7, p8, abs_gx, abs_gy);
   input [7:0]p0;
   input [7:0]p1;
   input [7:0]p2;
   input [7:0]p3;
   input [7:0]p5;
   input [7:0]p6;
   input [7:0]p7;
        input [7:0]p8;

        output [7:0] abs_gx;                            // output 8-bit gx value
        output [7:0] abs_gy;                            // output 8-bit gy value

        wire [10:0] abs_gx2;                            // output 8-bit gx value
        wire [10:0] abs_gy2;                            // output 8-bit gy value

        wire signed [10:0]gx;
        wire signed [10:0]gy;

        assign gx = ((p2-p0) + ((p5-p3)<<1) + (p8-p6));//calculation for gradient in x-direction
   assign gy = ((p0-p6) + ((p1-p7)<<1) + (p2-p8));//calculation for gradient in y-direction
```

```
            assign abs_gx2 = (gx[10]? ~gx+1 : gx);    // to find the absolute value of gx.
            assign abs_gy2 = (gy[10]? ~gy+1 : gy);    // to find the absolute value of gy.

            assign abs_gx = (|gx[10:8]) ? 8'hff : gx[7:0];
            assign abs_gy = (|gy[10:8]) ? 8'hff : gx[7:0];


endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Corner module: calculates the value of C
//////////////////////////////////////////////////////////////////////////////////
module corner(
            input clk,
            input reset,
   input [11:0] xx_sum,
   input [11:0] yy_sum,
   input [11:0] xy_sum,
   output [23:0] c_out
   );

            //reg sum_xy[23:0] = xx_sum*yy_sum;

            //initial registers
            reg [23:0]sum_xy_sq = 0;
            reg [23:0]sum_xxyy = 0;
            reg [23:0]sum_Ix_plus_Iy = 0;

            //second stage registers
            reg [23:0]sum_xy_sq_st2 = 0;
            reg [23:0]sum_xxyy_st2 = 0;
            reg [23:0]sum_Ix_plus_Iy_sq = 0;

            //third stage registers
            reg [23:0]sum_xy_sq_st3 = 0;
            reg [23:0]sum_xxyy_st3 = 0;
            reg [23:0]sum_Ix_plus_Iy_sq_st3 = 0;

            //shift three to the right to divide by 2^3
            //reg [23:0]k_sum = 0;

            reg [23:0]c = 0;

            always @ (posedge clk) begin

                        if(reset)begin
                                  sum_xy_sq = 0;
```

```
                                sum_xxyy = 0;
                                sum_Ix_plus_Iy = 0;

                        //second stage registers
                                sum_xy_sq_st2 = 0;
                                sum_xxyy_st2 = 0;
                                sum_Ix_plus_Iy_sq = 0;

                                //third stage registers
                                sum_xy_sq_st3 = 0;
                                sum_xxyy_st3 = 0;
                                sum_Ix_plus_Iy_sq_st3 = 0;
                                end

                                //back to initial values on reset
                                sum_xy_sq = xy_sum*xy_sum;
                                sum_xxyy = xx_sum*yy_sum;
                                sum_Ix_plus_Iy = xx_sum + yy_sum;

                                //transition from frist stage to second stage
                                sum_xy_sq_st2 <= sum_xy_sq;
                                sum_xxyy_st2 <= sum_xxyy;
                                sum_Ix_plus_Iy_sq <= sum_Ix_plus_Iy*sum_Ix_plus_Iy;

                                //transition from second stage to third stage
                                sum_xy_sq_st3 <= sum_xy_sq_st2;
                                sum_xxyy_st3 <= sum_xxyy_st2;
                                //since k = 1/8, shift three to the right to divide by 8 or 2^3
                                sum_Ix_plus_Iy_sq_st3 <= {3'b0, sum_Ix_plus_Iy_sq[23:3]};

                        //final stage
                        c <= sum_xxyy_st3 - sum_xy_sq_st3 - sum_Ix_plus_Iy_sq_st3;
                        end

        //assign output
        //assign c_out = (|c[23:16]) ? 15'hFFF : c[15:0];
        assign c_out = c[23:0];

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Derive module takes reads
//////////////////////////////////////////////////////////////////////////////////
module corner_top
(
        input clk,
        input reset,
        input [31:0] new_pixels, //32-bit input consisting of 4 8-bit pixels
        input new_data, //flag signaling 4 new pixels are ready
        input [8:0] row,
        input [9:0] column,
        input win,

        output reg [35:0] harris_out,
        output we, //low for one cycle after we output data
        output reg [18:0] zbt_address,
        output [23:0]c_out42,
        output [19:0]xx_sum1_out
        );

        //threshold for corner value
        localparam th = 64000;
        //localparam th = 20'hFFFFF;
        //localparam th = 6'b000011;
        //localparam th = 12'b0;

        //register array corresponding to 5x8 window, row column convention
        // reg [WIDTH-1 : 0] rf [DEPTH-1 : 0];
        reg [7:0] local_window [4:0][7:0];

        reg window_ready = 0;
        reg window_ready2 = 0;
        reg [18:0] win_address;

        //gx and gy calculatioin for window pixel(1,1)
        wire[7:0]gx11;
        wire[7:0]gy11;
        //gx and gy calculation for pixel(1,2)
        wire[7:0]gx12;
        wire[7:0]gy12;
        //gx and gy calculation for pixel(1,3)
        wire[7:0]gx13;
        wire[7:0]gy13;
        //gx and gy calculation for pixel(1,4)
        wire[7:0]gx14;
        wire[7:0]gy14;
        //gx and gy calculation for pixel(1,5)
        wire[7:0]gx15;
        wire[7:0]gy15;
        //gx and gy calculation for pixel(1,6)
        wire[7:0]gx16;
        wire[7:0]gy16;
        //gx and gy calculation for pixel(2,1)
        wire[7:0]gx21;
        wire[7:0]gy21;
```

```
	//gx and gy calculation for pixel(2,2)
	wire[7:0]gx22;
	wire[7:0]gy22;
	//gx and gy calculation for pixel(2,3)
	wire[7:0]gx23;
	wire[7:0]gy23;
	//gx and gy calculation for pixel(2,4)
	wire[7:0]gx24;
	wire[7:0]gy24;
	//gx and gy calculation for pixel(2,5)
	wire[7:0]gx25;
	wire[7:0]gy25;
	//gx and gy calculation for pixel(2,6)
	wire[7:0]gx26;
	wire[7:0]gy26;
	//gx and gy calculation for pixel(3,1)
	wire[7:0]gx31;
	wire[7:0]gy31;
	//gx and gy calculation for pixel(3,2)
	wire[7:0]gx32;
	wire[7:0]gy32;
	//gx and gy calculation for pixel(3,3)
	wire[7:0]gx33;
	wire[7:0]gy33;
	//gx and gy calculation for pixel(3,4)
	wire[7:0]gx34;
	wire[7:0]gy34;
	//gx and gy calculation for pixel(3,5)
	wire[7:0]gx35;
	wire[7:0]gy35;
	//gx and gy calculation for pixel(3,6)
	wire[7:0]gx36;
	wire[7:0]gy36;

	//registers for sums of xx gradients, yy gradients, and xy gradients required for calculation in
corner module
	//one for each of the four corners of interest
	//20 bits to store the maximum possible value of 255 * 255 * 8 = 0x7F008
	 reg [19:0] xx_sum1;
   reg [19:0] yy_sum1;
   reg [19:0] xy_sum1;

	 reg [19:0] xx_sum2;
   reg [19:0] yy_sum2;
   reg [19:0] xy_sum2;

	 reg [19:0] xx_sum3;
   reg [19:0] yy_sum3;
   reg [19:0] xy_sum3;

            reg [19:0] xx_sum4;
   reg [19:0] yy_sum4;
   reg [19:0] xy_sum4;
           //module derive_pixel( p0, p1, p2, p3, p5, p6, p7, p8, out);

           assign xx_sum1_out = xx_sum1;

           //pixel gradients

           //FIRST ROW OF X AND Y GRADIENTS (1,1) through (1,6)
           //pixel gradients at location (1,1)
           derive_pixel d1(.p0(local_window[0][0]),.p1(local_window[0][1]),.p2(local_window[0][2]),
                                       .p3(local_window[1][0]),.p5(local_window[1][2]),

           .p6(local_window[2][0]),.p7(local_window[2][1]),.p8(local_window[2][2]),.abs_gx(gx11),
.abs_gy(gy11) );

           //pixel gradients at location (1,2)
           derive_pixel d2(.p0(local_window[0][1]),.p1(local_window[0][2]),.p2(local_window[0][3]),
                                       .p3(local_window[1][1]),.p5(local_window[1][3]),

           .p6(local_window[2][1]),.p7(local_window[2][2]),.p8(local_window[2][3]),.abs_gx(gx12),.abs_gy(
gy12) );

           //pixel gradients at location (1,3)
           derive_pixel d3(.p0(local_window[0][2]),.p1(local_window[0][3]),.p2(local_window[0][4]),
                                       .p3(local_window[1][2]),.p5(local_window[1][4]),

           .p6(local_window[2][2]),.p7(local_window[2][3]),.p8(local_window[2][4]),.abs_gx(gx13),.abs_gy(
gy13) );

           //pixel gradients at location (1,4)
           derive_pixel d4(.p0(local_window[0][3]),.p1(local_window[0][4]),.p2(local_window[0][5]),
                                       .p3(local_window[1][3]),.p5(local_window[1][5]),

           .p6(local_window[2][3]),.p7(local_window[2][4]),.p8(local_window[2][5]),.abs_gx(gx14),
.abs_gy(gy14) );

           //pixel gradients at location (1,5)
           derive_pixel d5(.p0(local_window[0][4]),.p1(local_window[0][5]),.p2(local_window[0][6]),
                                       .p3(local_window[1][4]),.p5(local_window[1][6]),

           .p6(local_window[2][4]),.p7(local_window[2][5]),.p8(local_window[2][6]),.abs_gx(gx15),
.abs_gy(gy15) );

           //pixel gradients at location (1,6)
```

```
        derive_pixel d6(.p0(local_window[0][5]),.p1(local_window[0][6]),.p2(local_window[0][7]),
                                .p3(local_window[1][5]),.p5(local_window[1][7]),

        .p6(local_window[2][5]),.p7(local_window[2][6]),.p8(local_window[2][7]),.abs_gx(gx16),
.abs_gy(gy16) );

        ///////////////////////////////////////////////////
        //SECOND ROW OF X AND Y GRADIENTS (2,1) through (2,6)
        derive_pixel d7(.p0(local_window[1][0]),.p1(local_window[1][1]),.p2(local_window[1][2]),
                                .p3(local_window[2][0]),.p5(local_window[2][2]),

        .p6(local_window[3][0]),.p7(local_window[3][1]),.p8(local_window[3][2]),.abs_gx(gx21),
.abs_gy(gy21) );

        //pixel gradients at location (2,2)
        derive_pixel d8(.p0(local_window[1][1]),.p1(local_window[1][2]),.p2(local_window[1][3]),
                                .p3(local_window[2][1]),.p5(local_window[2][3]),

        .p6(local_window[3][1]),.p7(local_window[3][2]),.p8(local_window[3][3]),.abs_gx(gx22),.abs_gy(
gy22) );

        //pixel gradients at location (2,3)
        derive_pixel d9(.p0(local_window[1][2]),.p1(local_window[1][3]),.p2(local_window[1][4]),
                                .p3(local_window[2][2]),.p5(local_window[2][4]),

        .p6(local_window[3][2]),.p7(local_window[3][3]),.p8(local_window[3][4]),.abs_gx(gx23),.abs_gy(
gy23) );

        //pixel gradients at location (2,4)
        derive_pixel d10(.p0(local_window[1][3]),.p1(local_window[1][4]),.p2(local_window[1][5]),
                                .p3(local_window[2][3]),.p5(local_window[2][5]),

        .p6(local_window[3][3]),.p7(local_window[3][4]),.p8(local_window[3][5]),.abs_gx(gx24),
.abs_gy(gy24) );

        //pixel gradients at location (2,5)
        derive_pixel d11(.p0(local_window[1][4]),.p1(local_window[1][5]),.p2(local_window[1][6]),
                                .p3(local_window[2][4]),.p5(local_window[2][6]),

        .p6(local_window[3][4]),.p7(local_window[3][5]),.p8(local_window[3][6]),.abs_gx(gx25),
.abs_gy(gy25) );

        //pixel gradients at location (2,6)
        derive_pixel d12(.p0(local_window[1][5]),.p1(local_window[1][6]),.p2(local_window[1][7]),
                                .p3(local_window[2][5]),.p5(local_window[2][7]),

        .p6(local_window[3][5]),.p7(local_window[3][6]),.p8(local_window[3][7]),.abs_gx(gx26),
.abs_gy(gy26) );
```

```
	////////////////////////////////////////////////////
	//THIRD ROW OF X AND Y GRADIENTS (3,1) through (3,6)
	derive_pixel d13(.p0(local_window[2][0]),.p1(local_window[2][1]),.p2(local_window[2][2]),
				.p3(local_window[3][0]),.p5(local_window[3][2]),

	.p6(local_window[4][0]),.p7(local_window[4][1]),.p8(local_window[4][2]),.abs_gx(gx31),
.abs_gy(gy31) );

	//pixel gradients at location (3,2)
	derive_pixel d14(.p0(local_window[2][1]),.p1(local_window[2][2]),.p2(local_window[2][3]),
				.p3(local_window[3][1]),.p5(local_window[3][3]),

	.p6(local_window[4][1]),.p7(local_window[4][2]),.p8(local_window[4][3]),.abs_gx(gx32),.abs_gy(
gy32) );

	//pixel gradients at location (3,3)
	derive_pixel d15(.p0(local_window[2][2]),.p1(local_window[2][3]),.p2(local_window[2][4]),
				.p3(local_window[3][2]),.p5(local_window[3][4]),

	.p6(local_window[4][2]),.p7(local_window[4][3]),.p8(local_window[4][4]),.abs_gx(gx33),.abs_gy(
gy33) );

	//pixel gradients at location (3,4)
	derive_pixel d16(.p0(local_window[2][3]),.p1(local_window[2][4]),.p2(local_window[2][5]),
				.p3(local_window[3][3]),.p5(local_window[3][5]),

	.p6(local_window[4][3]),.p7(local_window[4][4]),.p8(local_window[4][5]),.abs_gx(gx34),
.abs_gy(gy34) );

	//pixel gradients at location (3,5)
	derive_pixel d17(.p0(local_window[2][4]),.p1(local_window[2][5]),.p2(local_window[2][6]),
				.p3(local_window[3][4]),.p5(local_window[3][6]),

	.p6(local_window[4][4]),.p7(local_window[4][5]),.p8(local_window[4][6]),.abs_gx(gx35),
.abs_gy(gy35) );

	//pixel gradients at location (3,6)
	derive_pixel d18(.p0(local_window[2][5]),.p1(local_window[2][6]),.p2(local_window[2][7]),
				.p3(local_window[3][5]),.p5(local_window[3][7]),

	.p6(local_window[4][5]),.p7(local_window[4][6]),.p8(local_window[4][7]),.abs_gx(gx36),
.abs_gy(gy36) );

	//outputs with c value for each of the four possible corners
	wire [23:0] c_out11;
	wire [23:0] c_out21;
	wire [23:0] c_out31;
	wire [23:0] c_out41;

	reg [23:0] c_out1;
	reg [23:0] c_out2;
	reg [23:0] c_out3;
	reg [23:0] c_out4;

	//compute c value for each corner
	corner corner1(.clk(clk),.reset(reset),.xx_sum((|xx_sum1[19:12]) ? 12'hFFF :
xx_sum1[11:0]),.yy_sum((|yy_sum1[19:12]) ? 12'hFFF : yy_sum1[11:0]),.xy_sum((|xy_sum1[19:12]) ?
12'hFFF : xy_sum1[11:0]),.c_out(c_out11));
	corner corner2(.clk(clk),.reset(reset),.xx_sum((|xx_sum2[19:12]) ? 12'hFFF :
xx_sum2[11:0]),.yy_sum((|yy_sum2[19:12]) ? 12'hFFF : yy_sum2[11:0]),.xy_sum((|xy_sum2[19:12]) ?
12'hFFF : xy_sum2[11:0]),.c_out(c_out21));
	corner corner3(.clk(clk),.reset(reset),.xx_sum((|xx_sum3[19:12]) ? 12'hFFF :
xx_sum3[11:0]),.yy_sum((|yy_sum3[19:12]) ? 12'hFFF : yy_sum3[11:0]),.xy_sum((|xy_sum3[19:12]) ?
12'hFFF : xy_sum3[11:0]),.c_out(c_out31));
	corner corner4(.clk(clk),.reset(reset),.xx_sum((|xx_sum4[19:12]) ? 12'hFFF :
xx_sum4[11:0]),.yy_sum((|yy_sum4[19:12]) ? 12'hFFF : yy_sum4[11:0]),.xy_sum((|xy_sum4[19:12]) ?
12'hFFF : xy_sum4[11:0]),.c_out(c_out41));

	assign c_out42 = c_out1;

	//reg flip = 0;

		always @ (posedge clk)begin

			if (reset) begin
				window_ready <= 0;
				window_ready2 <= 0;
				end

			if(window_ready)begin

				//flip <= ~flip;

				c_out1 <= c_out11;
				c_out2 <= c_out21;
				c_out3 <= c_out31;
				c_out4 <= c_out41;

				//sums for pixel (2,2)
				xx_sum1 <= gx11*gx11 + gx12*gx12 + gx13*gx13 + gx23*gx23 +
gx33*gx33 + gx32*gx32 + gx31*gx31 + gx21*gx21;
				yy_sum1 <= gy11*gy11 + gy12*gy12 + gy13*gy13 + gy23*gy23 +
gy33*gy33 + gy32*gy32 + gy31*gy31 + gy21*gy21;
				xy_sum1 <= gx11*gy11 + gx12*gy12 + gx13*gy13 + gx23*gy23 +
gx33*gy33 + gx32*gy32 + gx31*gy31 + gx21*gy21;
```

```
                                        //sums for pixel (2,3)
                                        xx_sum2 <= gx12*gx12 + gx13*gx13 + gx14*gx14 + gx24*gx24 +
gx34*gx34 + gx33*gx33 + gx32*gx32 + gx22*gx22;
                                        yy_sum2 <= gy12*gy12 + gy13*gy13 + gy14*gy14 + gy24*gy24 +
gy34*gy34 + gy33*gy33 + gy32*gy32 + gy22*gy22;
                                        xy_sum2 <= gx12*gy12 + gx13*gy13 + gx14*gy14 + gx24*gy24 +
gx34*gy34 + gx33*gy33 + gx32*gy32 + gx22*gy22;

                                        //sums for pixel (2,4)
                                        xx_sum3 <= gx13*gx13 + gx14*gx14 + gx15*gx15 + gx25*gx25 +
gx35*gx35 + gx34*gx34 + gx33*gx33 + gx23*gx23;
                                        yy_sum3 <= gy13*gy13 + gy14*gy14 + gy15*gy15 + gy25*gy25 +
gy35*gy35 + gy34*gy34 + gy33*gy33 + gy23*gy23;
                                        xy_sum3 <= gx13*gy13 + gx14*gy14 + gx15*gy15 + gx25*gy25 +
gx35*gy35 + gx34*gy34 + gx33*gy33 + gx23*gy23;

                                        //sums for pixel (2,5)
                                        xx_sum4 <= gx14*gx14 + gx15*gx15 + gx16*gx16 + gx26*gx26 +
gx36*gx36 + gx35*gx35 + gx34*gx34 + gx24*gx24;
                                        yy_sum4 <= gy14*gy14 + gy15*gy15 + gy16*gy16 + gy26*gy26 +
gy36*gy36 + gy35*gy35 + gy34*gy34 + gy24*gy24;
                                        xy_sum4 <= gx14*gy14 + gx15*gy15 + gx16*gy16 + gx26*gy26 +
gx36*gy36 + gx35*gy35 + gx34*gy34 + gx24*gy24;

                                        zbt_address <= win_address;
                                        window_ready <= 0;

                                harris_out <= {local_window[2][2], (c_out1 > th) ? 1'b1:1'b0,
local_window[2][3], (c_out2  > th) ? 1'b1:1'b0, local_window[2][4], (c_out3  > th) ? 1'b1:1'b0,
local_window[2][5], (c_out4  > th) ? 1'b1:1'b0};


                                        end

                                //load new data from input onto local window
                                if (new_data)begin
                                        if (!win) begin
                                                win_address <= {2'b0, row, column[9:2]};
                                                //populate 5 x 4 window
                                                {local_window[4][0], local_window[4][1], local_window[4][2],
local_window[4][3]} <= new_pixels;
                                                {local_window[3][0], local_window[3][1], local_window[3][2],
local_window[3][3]} <= {local_window[4][0], local_window[4][1], local_window[4][2],
local_window[4][3]};
                                                {local_window[2][0], local_window[2][1], local_window[2][2],
local_window[2][3]} <= {local_window[3][0], local_window[3][1], local_window[3][2],
local_window[3][3]};
```

```
						{local_window[1][0], local_window[1][1], local_window[1][2],
local_window[1][3]} <= {local_window[2][0], local_window[2][1], local_window[2][2],
local_window[2][3]};
						{local_window[0][0], local_window[0][1], local_window[0][2],
local_window[0][3]} <= {local_window[1][0], local_window[1][1], local_window[1][2],
local_window[1][3]};
						end
					if (win) begin
						{local_window[4][4], local_window[4][5], local_window[4][6],
local_window[4][7]} <= new_pixels;
						{local_window[3][4], local_window[3][5], local_window[3][6],
local_window[3][7]} <= {local_window[4][4], local_window[4][5], local_window[4][6],
local_window[4][7]};
						{local_window[2][4], local_window[2][5], local_window[2][6],
local_window[2][7]} <= {local_window[3][4], local_window[3][5], local_window[3][6],
local_window[3][7]};
						{local_window[1][4], local_window[1][5], local_window[1][6],
local_window[1][7]} <= {local_window[2][4], local_window[2][5], local_window[2][6],
local_window[2][7]};
						{local_window[0][4], local_window[0][5], local_window[0][6],
local_window[0][7]} <= {local_window[1][4], local_window[1][5], local_window[1][6],
local_window[1][7]};
						window_ready <= 1;
						end
					end
				window_ready2 <= window_ready;
			end

			//we is high for one cycle when the row is even and greater than 3
			assign we = window_ready2;

endmodule
```