Final Project Proposal Draft

Overview

The following is the overall block diagram of our project:

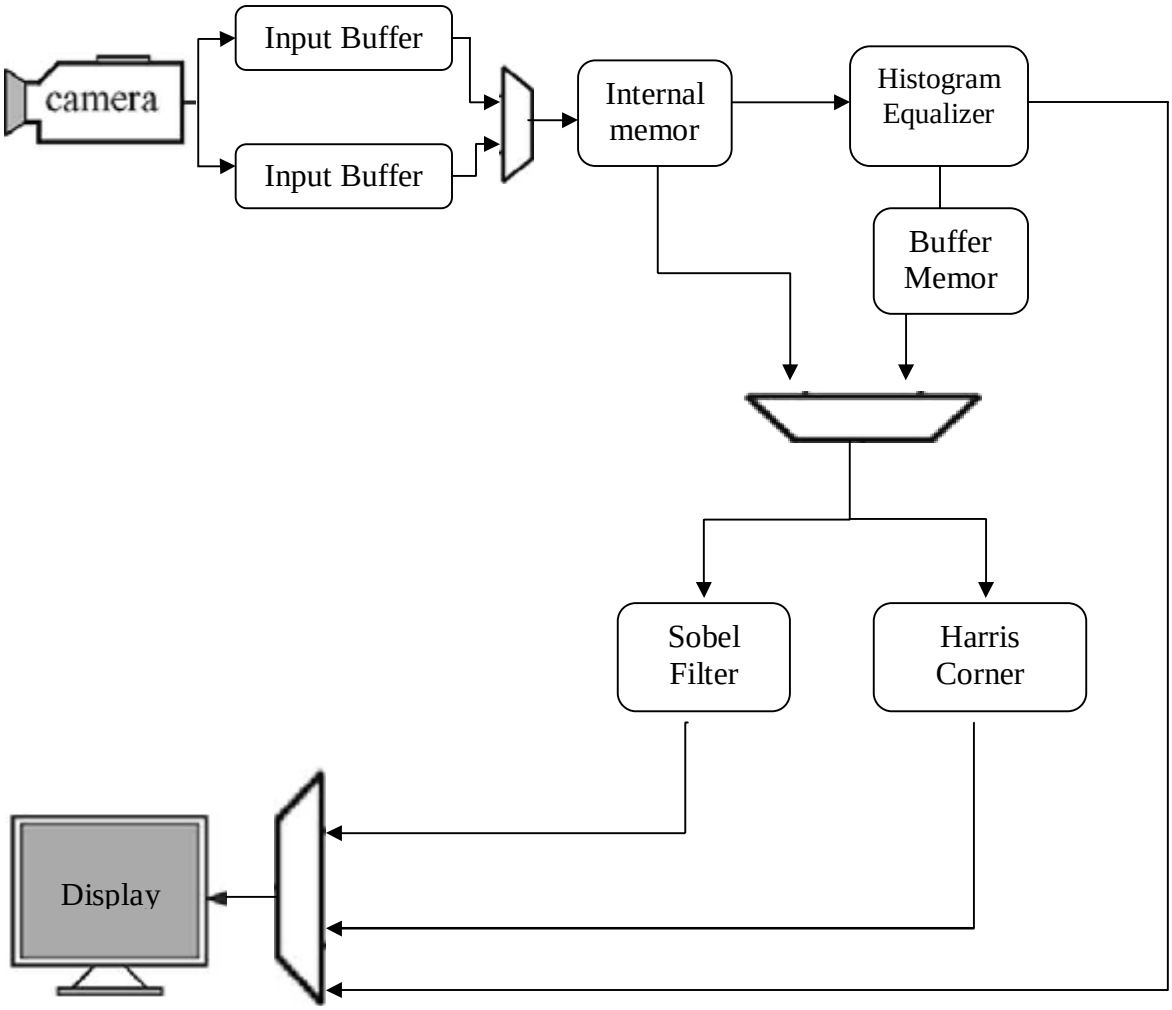


Figure 1. Block diagram of the Image Processing Unit

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

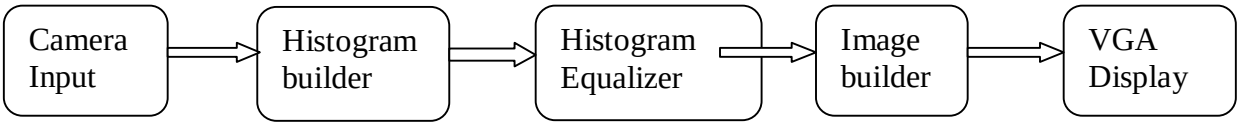


Figure 1. Block diagram for the Histogram Equalizer

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.

$$C(k) = \frac{\sum_0^k n(k)}{p * q}$$, where p*q is the size of the image

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.

$$K\_new(i,j) = 256*C(k)$$

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. Introducing parallelism at this point can make the process execute much faster. Also the histogram builder and calculation of the summation of the n(k) values for the histogram equalizer can be calculated at the same time using a switching decoder circuit[1]. This implementation mostly has incremental operations with one division. This module uses memory space to store the original image and counter values. More memory may be used depending on the number of copies of the counter created. It also creates a LUT for the original intensity values and the scaled CDF

values for a given intensity. The VGA display just reads of the new intensity value pixel from the lookup table based on the intensity value at that pixel. In order to test this module a software implementation has been done to compare the results.

References:

[1] Alsuwailem A. M., A Novel FPGA Based Real-time Histogram Equalization Circuit for Infrared Image Enhancement.
http://www.oldcitypublishing.com/FullText/JAPEDfulltext/JAPED3.3-4fulltext/Alsuwailem2.pdf

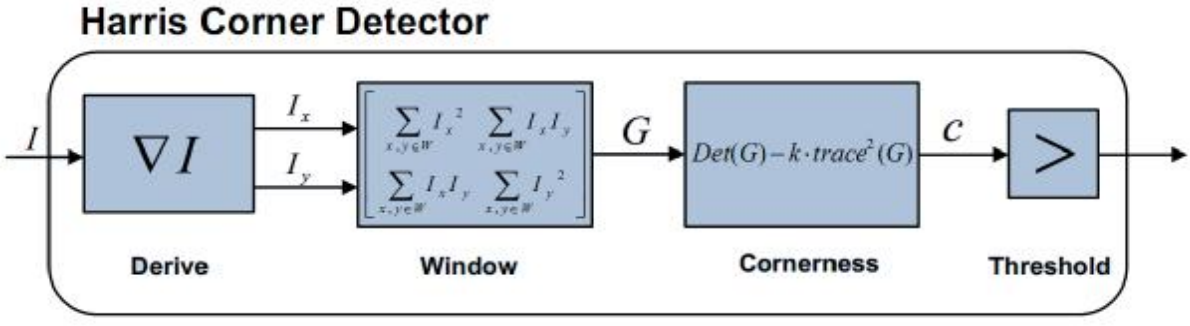
Harris Corner Detection (Dember)

Overview

The harris corner detection module will be used for performing feature detection on the output from the camera. The module will capture a still image, save it to memory and then use the harris corner detection algorithm to mark corners on the image that meet a certain criteria set by the user. These corners will then become an overlay on the original picture and will be displayed on the VGA monitor.

In order to extract the features, it needs to perform the following computations for each pixel, which it will perform in 3 x 3 pixel windows:

1) Convolve image with horizontal and vertical differential operator to obtain Ix and Iy.
2) Generate the three summations necessary from Ix and Iy over the weighted window.
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 (x,y) coordinates of pixel; otherwise output (0,0).



Module: Obtain_image
Input: 640 x 480 8-bit output from camera

This module saves a snapshot taken by the camera to memory. This requires 307.2 KB of memory. It converts the image to grayscale from the RGB or the luminance values (to be determined)

Module: Derive_pixel
Input: 3 x 3 pixel window (8 8-bit pixels)
Output: Ix [0:10]
Output: Iy [0:10]

This module will be purely combinational.

-1 0 1 Horizontal Gradient Operator
-2 0 2
-1 0 1

-1 -2 -1 Vertical Gradient Operator
 0 0 0
 1 2 1

These two masks are convolved with the 3x3 window to obtain the gradient of the image. This step is done by convolving each individual pixel in the image with the two 3 x 3 masks.

Result of x-convolution: Ix
Result of y-convolution: Iy

$$Ix[i, j] = Im[i+1, j-1] + 2*Im[i+1, j] + Im(i+1, j+1) - (Im[i-1, j-1] + 2*Im[i-1, j] + Im(i-1, j+1))$$

$$Iy[i, j] = Im[i-1, j+1] + 2*Im[i, j+1] + Im(i+1, j+1) - (Im[i-1, j-1] + 2*Im[i, j-1] + Im(i+1, j-1))$$

Ix, Iy will be 11-bit signed values, because the maximum value is 255*4 plus 1 sign bit.

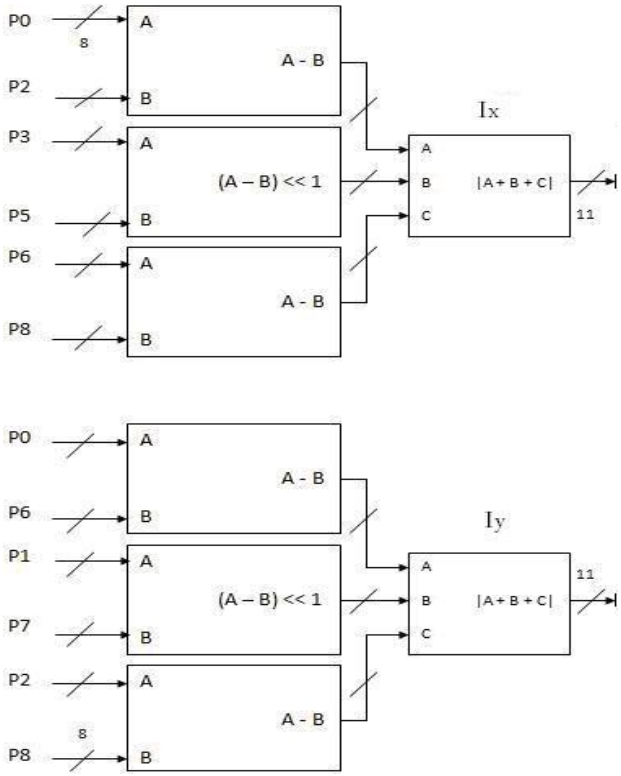


Figure: Block diagram for Deriv_pixel

Module: Derive
Input: 5 x 5 pixel window from memory.
Output: 3 x 3 pixel window for each Ix and Iy (total of 18 11-bit values)

This module instantiates several Derive_pixel modules and feeds it pixels that it reads from the ROM with the captured image. After it is done processing the image, it feeds Ix and Iy to the next module.

I considered the option of saving all the Ix and Iy values to memory. Each pixel has 11 bits of information and there are 640 x 480 pixels. Ix and Iy will each need to save 3,379,200 bits or 422.4 KB. The module ignores the outer layer of pixels, since it cannot be convoluted with a 3x3 filter and is not very important.

The approach I am going to try however, it to feed each Ix and Iy forward to the next module, so that the whole calculation is completed for that particular pixel and only the result is stored. This way Ix and Iy will not have to be saved for all 640 x 480 pixels.

Module Window:
Input: Ix, Iy (+ surrounding 8 Ix and Iy in 3 x 3 window)
Output: $\sum$ Ix^2, $\sum$ Iy^2), $\sum$ IxIy (20 bits each).

This module will take the two 3 x 3 windows with values for Ix and Iy. It will compute sums corresponding to the window module: Sum(Ix^2), Sum(Iy^2) and Sum (IxIy).

Module Corner:
Input: ∑ Ix^2, ∑ Iy^2, ∑ IxIy (22 bits each)
Output: value for C.

This module needs to compute the following:

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

C will be computed in a pipelined manner.

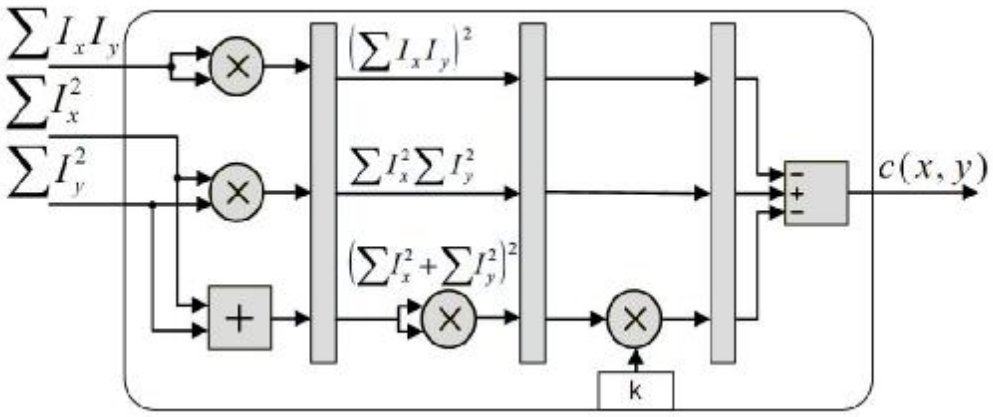


Figure: Pipelined Corner module

Module Threshold
Input: C
Output: True or False

This module takes in C and compares it against a set threshold used to determine whether the pixel associated with C is a corner or not. The threshold can be changed depending on whether one wants a large or small threshold and that is why it is in a separate module. However, this parameter can only be changed before compile time. Optional additions in the future could be having threshold that can be changed with a couple of the switches on the FPGA.

Testing

Testing will rely on what is being output to the VGA display. For different corner thresholds, it should display different numbers of corner overlays on the screen.

I also wrote a software implementation of the harris corner detector in python and plan to use that as well to compare the implementation in hardware to the one in software for similar images.

References:

B. Dietrich, “Design and Implementation of an FPGA-based Stereo Vision System for the EyeBot M6,” Diplomatarbeit, Technische Universitaet Muenchen, Munich, Germany, 2009.

SOBEL OPERATOR MODULE MODULE (Jon):

Module:

Sobel edge detector

Input:

3x3 block of 8-bit grayscale values, taken from histogram equalizer or internal memory.

Output:

One 8-bit edge value for the pixel in the center of the 3x3 input block, with higher values corresponding to higher changes in intensity at a given point. This can be used directly by the VGA display module as a grayscale pixel value to show edges, with whiter pixels corresponding to sharper edges.

Required throughput:

NTSC camera outputs 30 frames of video per second, with 640x480 pixels per frame. To do real time video with no downsampling, need 9,216,000 per second throughput, giving us 108 ns to do one input to output conversion. If we need to go faster than this, we can do the edge detection in parallel and also, the module can easily be pipelined to increase throughput.

Complexity:

Module does a convolution with two masks and adds the two results together. This requires doing 3 stages of signed adds. Since we only ever need to multiply by 2, we can just left shift, which takes no time to do if it’s hard wired in. There is also a compare at the end so that the module does not output a value greater than 255, which constitutes another add.

dx gradient mask:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

dy gradient mask: [1 2 1
0 0 0
-1 -2 -1]