Real-Time Animated Video

6.111 - Final Project Report - Fall 2013

David (Woo Hyeok) Kang
Tarun Malik
Ariana J. Eisenstein

December 8, 2013

Table of Contents

1. Overview

This document details the design of a modular architecture for an implementation of a Real-time animated video or cartoonifier. Animated effect makers, implemented either with software or done by hand, are targeted to images and are used after an image has been taken. The process takes a substantial amount of time and must be carried out for each frame individually. Our project implements a real-time animated video maker with dedicated FPGA hardware. The FPGA allows us to apply a cartoonify-ing effect to each video frame efficiently and thus produce a video from these frames, in real time. Our project implements edge detection on each input video frame using a point-wise multiplication with three-by-three Sobel operator; the output then passes through a color reduction module. The VGA monitor displays the final modified output, to give a cartoon-like video output.

2. High Level System Block Diagram

The diagram below (Figure 1) shows an overall logical flow of our project and how data coming in from the NTSC camera moves through modules. These blocks depict the transformations done to an image, finally outputting the proper effects on the VGA display. As shown in Figure 1, the processing of the Edge Detection module and Color Reduction Modules are done in parallel. Possible extensions include additional parallel processing along with the two middle modules without interfering the original flow of data.

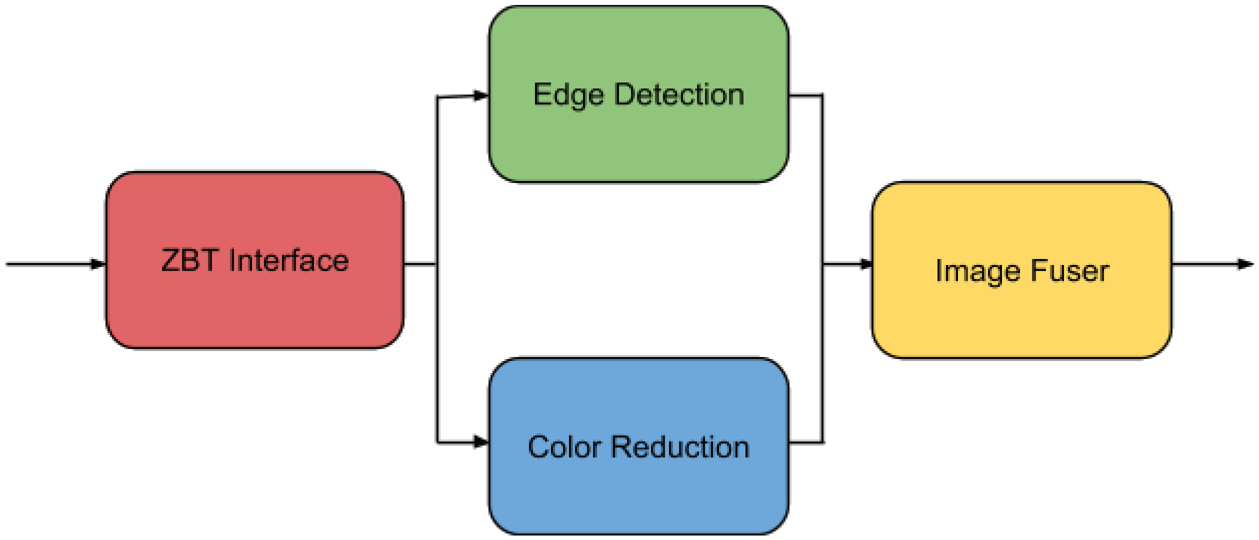


Figure 1: System Block Diagram

The system block diagram depicts the flow of the image data taken by a NTSC camera. Pixel data is passed through the Edge detection and the Color reduction modules which run in parallel. The outputs from the modules are overlapped to give the final animated image.

3. Description Of Modules

The proposed real-time cartoonifier is divided into three parts: an interface between the camera input and a block of ZBT memory to store data, two parallel cascades of image processing and transformations of the data stored in memory, and a final process to combine the images generated by the parallel processes.

Section 3.1 discusses how a control module, called ZBT Interface, directs a stream of information from the stored memory to the parallel processing cascades; section 3.2 and 3.3 discusses processing algorithms that comprise each cascade. The processing is done concurrently, as Figure 1 shows the parallelism between Color Reduction, Section 3.3, and Edge Detection, Section 3.2, modules.

The outputs of the parallel processed pixels are then fused, by the Image Fuser module described in Section 3.4, to have a pixel value corresponding to a cartoon image. Then the cartoon image pixel values are output on the VGA screen.

3.1 ZBT Interface (David):

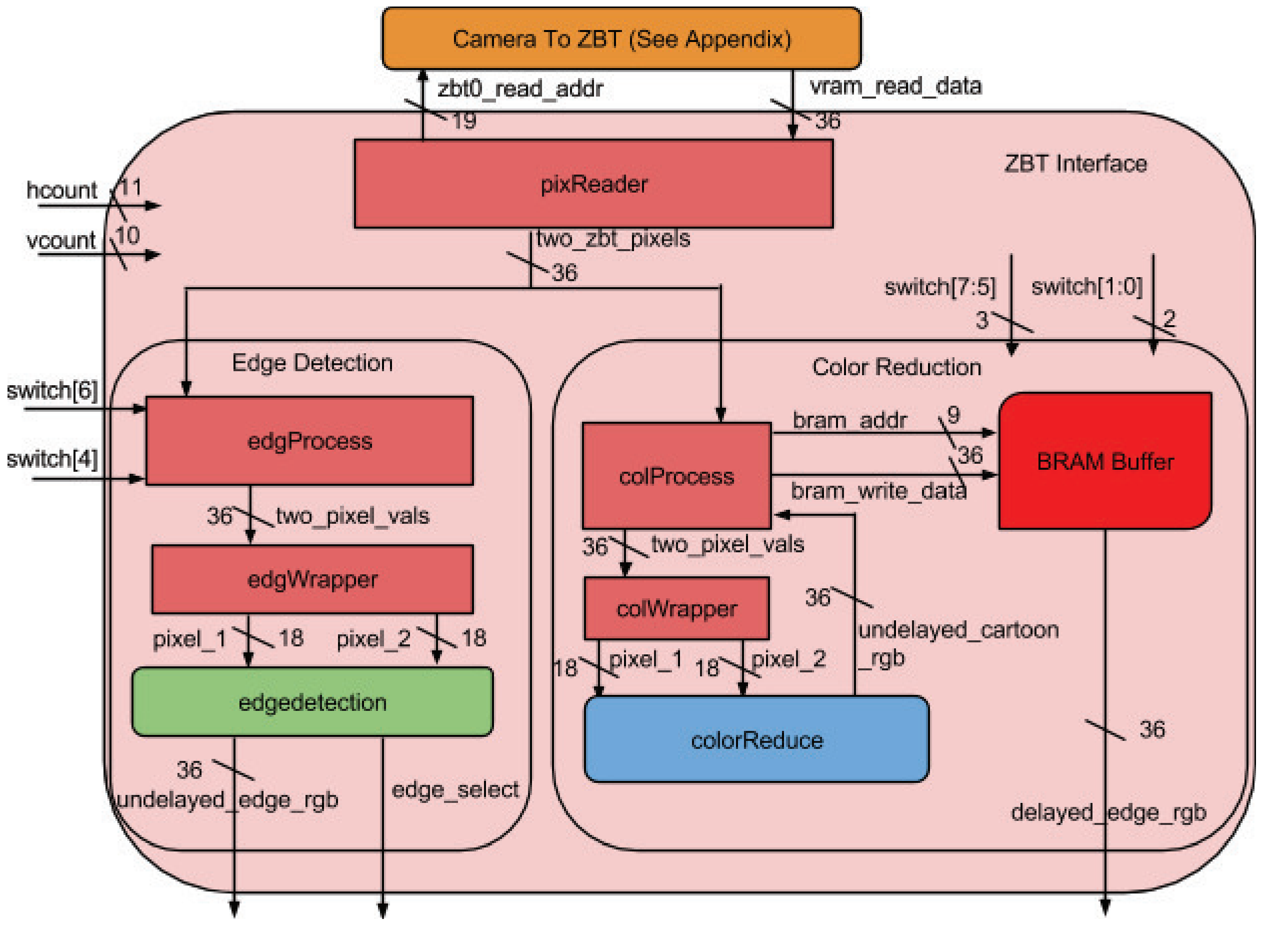


Figure 2: ZBT Interface Block Diagram

The ZBT interface block diagram depicts the flow of data from the ZBT memory block to the edge detection and color reduction modules. The Edge Detection and Color Reduction processes happen in parallel.

The ZBT Interface has four goals:
(i) store an incoming pixel data from a source camera into a memory, called memory_S
(ii) read a pixel data from memory_S
(iii) pass that pixel data onto two processing modules described in Section 3.2 and Section 3.3
(iv) store an incoming stream of processed pixels from (iii) into a second memory, called memory_P.

Instead of re-working all painstaking details to implement all four features, our interface reuses existing tools with desired functionalities; a sample code, called zbt_6111_sample, along with ntsc signal decoder, chip decoder, initializes memory_S and stores in memory_S intensity values. (See Appendix for a detailed block diagram) In that same code, there also exists a module called vram_display, which reads from memory_S pixel values and displays that pixel value onto the VGA output.

These valuable elements lead to an engineering decision: to re-use the vram_display module, by storing in a separate memory, memory_P of goal (iv), to output processed pixels. So instead of splicing up one memory structure, our interface utilizes two banks of ZBT memory, memory_S and memory_P.

With the above decision made, the ZBT Interface implementation divides up into three phases:
(A) Implmenting a functuonality of writing and displaying, independent of one another, to memory_S and memory_P exclusively
(B) Connecting ZBT Interface with each processing components described in Section 3.2 and Section 3.3
(C) Inter-connecting two processing components in parallel and understanding latencies involved in each processing process

Part (A), as suggested by Professor Gim Hom, is the single most important aspect of the project. With a control over writing and reading from different memories , our interface captures still images and also stores in a live feed. Using two switches and simple multiplexer hardwares in the display module and control signals such as write_enable that go into each memory element, the implementation takes a few minutes of debugging; its ability to debug non-working components in latter stages, however, saves hours of debugging. For initial implementation, the ZBT Interface writes to memory_S a source image and to memory_P an inverted image. Our goal states that only one pixel is intended to be processed; a constraint on memory size, however, requires storage of two pixels per address in memory_S. All processes described below work with two pixels at a time. To do so, memory_S stores only six bits per Red/Green/Blue pixel value, instead of its full eight bit values. These are reflected in Figure 2, where we have 36 bits of data being read from readPixel module and being passed onto downstream processing pipeline.

Part (B) saves compilation greatly and takes advantage of the modular nature of engineering projects. Instead of trying to compile everything else at once, the interface coordinates and tests processes in Section 3.2 and Section 3.3 incrementally. To do so, we note that a future team ought to start as early as possible in developing and testing with the ZBT Interface such that they have enough time to test things incrementally and debug. To give a feel for the necessity of Part (B), here are some compilation time details: the compilation of verilog code, on one hand, for the processing pipeline with just color reduction code took nearly 25 minutes; compilation time for the edge detection, on the other hand, took only 5 minutes. In light of Figure 2, only one of (edgProcess, edgWrapper) and (colProcess, colWrapper) pair is present in the compilation process.

Part (C) involves a fair amount of trial-and-error: comparison of simluation results on processing pipelines tells us exactly how much latency difference there is to account for. This latency difference, gotten from simulation, is a significant hurdle to our particular real-time application, as our architecture dictates that the interface stores into memory_P one of the two processed pixels. To compensate for this latency difference, the processing process with the smaller latency (one that outputs pixel values faster than the other), which happens to be the color reduction, stores its output into a separate memory within a Block RAM in the FPGA. Note that this memory structure is an intermediate memory structure used just before pixel datas are stored into memory_P. As for the trial-and-error, the exact dimensions and data structure of the Block RAM, the red block in Figure 2, takes some trial-and-error to figure out.

3.2. Edge Detection (Tarun):

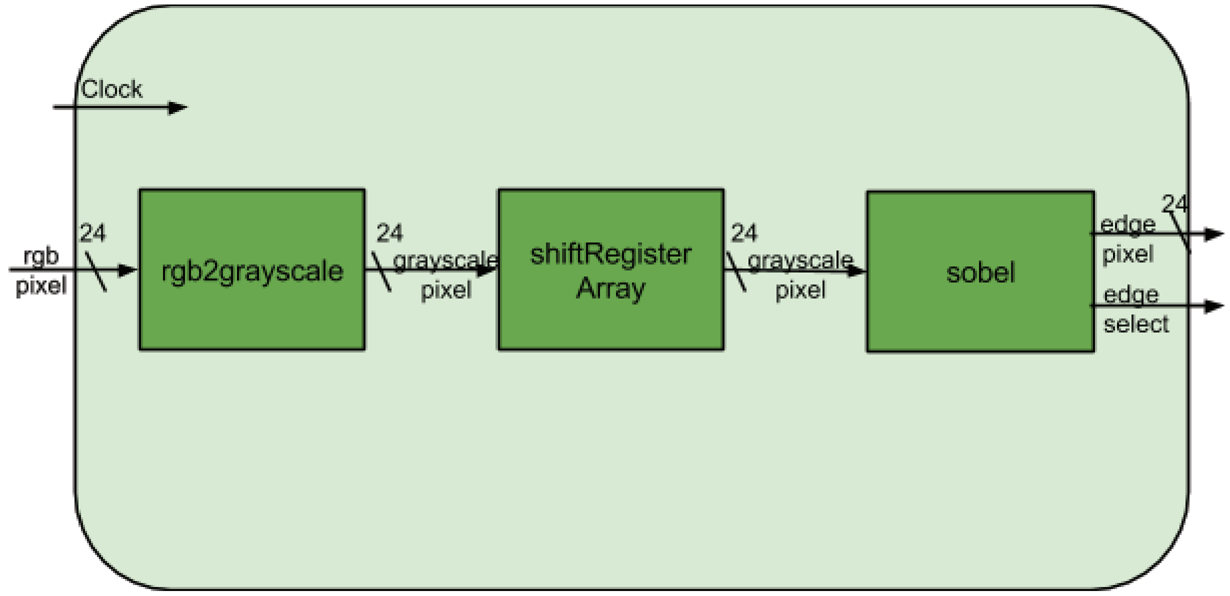


Figure 3: Edge Detetion Block Diagram

The Edge Detection block diagram depicts the flow of data from through the edge detection module. The pixels must be passed through a shiftregister array inorder to ensure that we have all the pixels to use in the sobel.

We implemented edge detection using a sobel gradient method to bold the images and give the it a cartoon-like effect. The Sobel method computes an approximation of the gradient of image intensity by masking the image data block with horiozontal and a vertical kernels.

The edge detection module consist of 4 sub-modules helping us achieve the effect,

3.2.1 RGB2Grayscale :

Our first module is a rgb2gray scale conver.The conversion equation for converting a 24 bit RGB pixel into a 8 bit grayscale value is:
Grayscale = 0.299 * R + 0.587 * G + 0.114 * B

However, this can be a difficult calculation & is time consuming for hardware implementation.
We have approximated this equation as
Grayscale = R/4 + G/2 + B/4

We have selected the G value to have a higher weight than the B or G values as the grayscale values primarily rely on the G value of the pixel. The divisions are implemented by shifting the bits.The output of this module is a 8-bit grayscale value passed on to the shiftregister storage.

3.2.2 Shift-Register Storage :

To implement sobel, we are required to store the image data to implement the sobel algorithm. It requires us to store 3 rows of pixels. We stored these pixels in a shiftregister. On every clock cycle , pixels are shifted to the right. At the end of every row, pixel are moved to the next row.
The output of the shift-register is a 3x3 matrix, with the elements as the last 3 elements of each row.

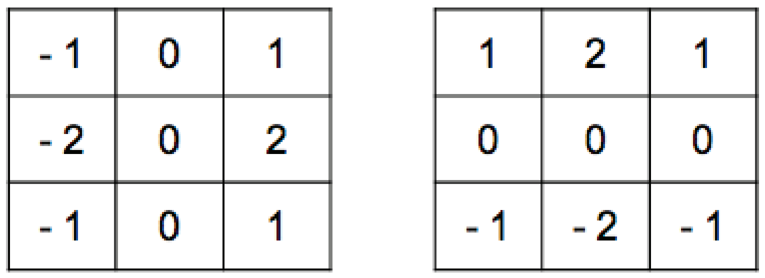
3.2.3 Sobel Edge Detection :

The edge detection module performs a mathematical operation on the image to identify pixels with sharp image brightness changes. These pixels are marked as edges, and are given a bold black color value after the edge detection operation.

The **input** to the this module is a 8-bit grayscale pixel value,from the rgb2gray converter module.
The **output** of this module is an 8-bit edge which is overlapped on the center pixel of the 3x3 input block.

The Sobel Filter performs a 2-D spatial gradient measurement on an image and identifies regions of high spatial frequency that would correspond to edges.

The Sobel operator consists of a pair of 3x3 kernels which are masked over a 3x3 portion of the original image. One of the kernels is simply the other rotated by 90 degrees.The kernels are shown in the figure below.



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

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

Figure 4: Sobel Kernels
The image shows the Sobel kernels which are masked over the image, for use in the edge detection module

The two kernels are for horizontal and vertical masking, respectively. The kernels are applied separately to the input image, to produce separate measurements of the gradient component in each orientation. The directional gradients, two 1-D spatial gradient measurements, are then used to find the absolute gradient magnitude at each point in the image.

After the absolute gradient value calculation, a threshold is applied to the image. If the threshold value is greater than the current pixel value, the current pixel value is set to the threshold value. Otherwise, the new threshold value is set to the current pixel value. The output of this module is an 8-bit edge which is overlapped on the center pixel of the 3x3 input block.

Here is a mathematical description of individual filters within the edge detection module:
The **x-direction derivative** after masking the sobel operator on the image is given by:
$Gx = (z_7+2z_8+z_9) - (z_1+2z_2+z_3)$

The **y-direction derivative** after masking the sobel operator on the image is given by:
$Gy = (z_3+2z_6+z_9) - (z_1+2z_4+z_7)$

The **Gradient** of the pixel after combining the horizontal and vertical component is given by:
$G = abs(Gx) + abs(Gy)$

The output from sobel is either an edge 8’h00 or not an edge 8’hFF. This 8-bit value from the sobel is extended to a 24-bit value by this module. It also outputs a select bit, to be used by the image fuser to select between the edge detection or the color reduced output. If the pixel is an edge it output the select bit as 1, otherwise selectbit is given a 0 value.The final output of this module can be seen in figure 5 below.

3.2.4 Select-Bit Module:

The output from sobel is either an edge 8’h00 or not an edge 8’hFF. This 8-bit value from the sobel is extended to a 24-bit value by this module. It also outputs a select bit, to be used by the image fuser to select between the edge detection or the color reduced output. If the pixel is an edge it output the select bit as 1, otherwise selectbit is given a 0 value.

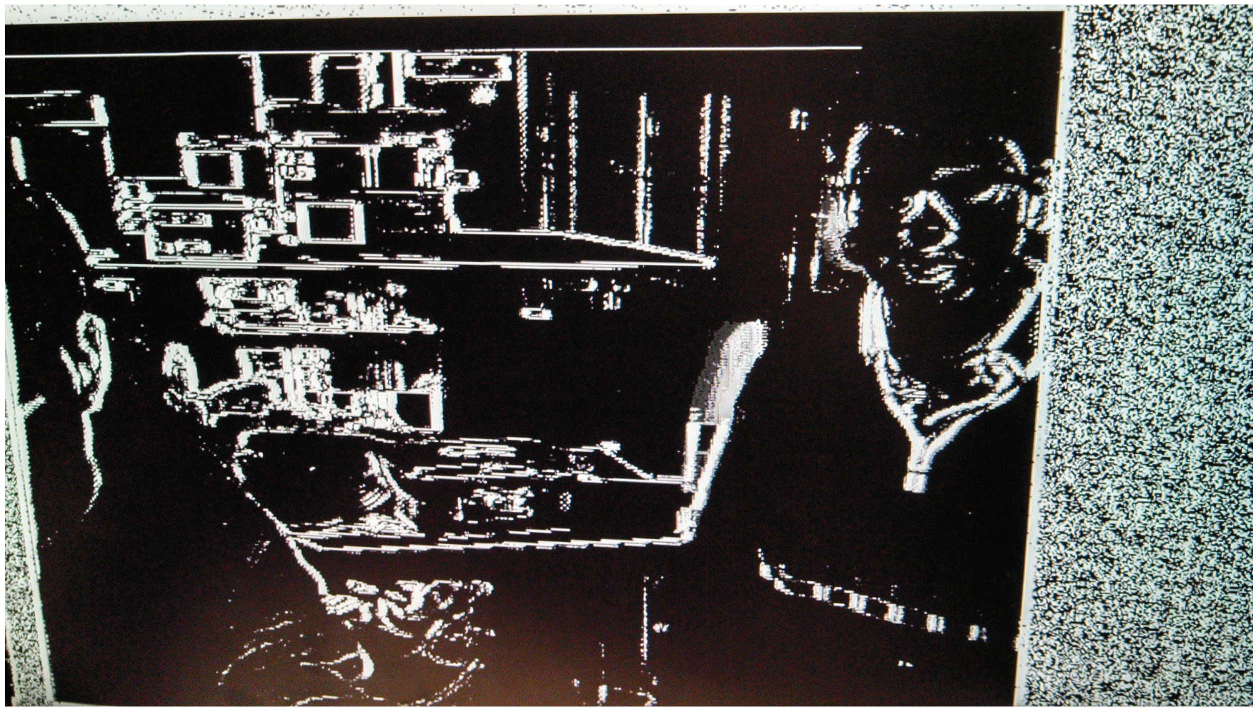


Figure 5: 3x3 Sobel Edge Detected Output

An edge detected image with a 3x3 sobel operator.The edge are shown in white in this output to better see the result.

3.3. Color Reduction (Ariana):

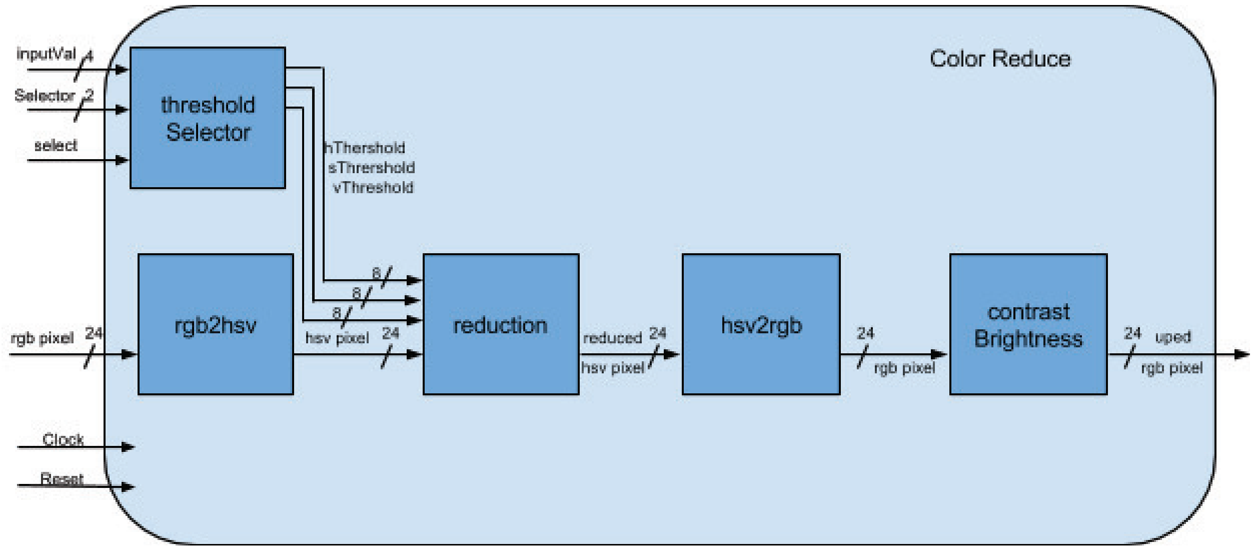


Figure 6: Color Redution Block Diagram

The color reduction block diagram depicts the flow of the color data taken given by the first ZBT. Pixel data is passed through four modules, which transforms the data to thresholded colors. The output from the module is sent to the image fuser. A threshold selector allows for variation in the threshold levels of the color reduction.

In order to achieve a cartoon-like effect with solid colors, the total number of colors in the image must be reduced, a reduction from 2^{24} colors to approximatley 2^{7} colors. A thresholding method decreases the number of colors. This method changes all values within a specified range to the same value. In order to achieve more accurate colors, the thresholding is done in a Hue-Saturation-Value (HSV) color scheme, with thresholding of the Hue, Saturation, and Value parameters done separately.

The color reduction module consits of five individual modules, four in cascade for pixel operations and one for selecting threshold levels as seen in Figure 6. The pipeline operates on a pixel by pixel basis meaning it outputs one pixel a clock cycle.

The first module, rgb2hsv, is a conversion of pixels values from the RGB space to the HSV space. This module was provided by the 6.111 staff. We had to generate the Coregen Divider necessary for the functionality of the module, as well as account for the delay caused by this division. We used the smallest Coregen Divider (16 bits, unsigned) possible for our project constraints to minimize compile time of our project.

It may have been better to use a divider module, rather than the Coregen Divider, as the color reduction had a shorter latency than the edge detection, and the divider did not need to be pipelined. This implementation would have reduced the compile time of our project significantly, which would have helped with debugging.

The pixel, now in HSV space, is masked with thresholds in the H, S, and V spaces separately, setting a select number of least significant bits to zero, using a threshold value and a bitwise AND. This process reduces the total number of colors that the colors can now represent. These thresholds set in in the threshold selection module.

The threshold selection module allows the user to change the threshold values using the switches on the FPGA. This module was extremely useful in testing as it allowed us to look at different threshold values without waiting for the project to recomplile allowing for faster debugging. The inputs to this module are: an input value, dictating the number of bits to preserve; a selector value, dictating which of H, S, or V are to be changed; and select, a button dictating to switch now. The input value is subtracted from eight. The differences is then used as the number of bits to shift an eight bit number containing only 1's. This creates the mask necessary to threshold the color values. The selector value is then used in a case statement to change the selected values mask to the input value.

By default, the hThreshold is 2^3 (11100000), the vThreshold is 2^2 (11000000), and the sThreshold is 2^2 (11000000). The number of significant bits is the power 2 is raised to number of colors possible for H, S, or V. Thus, the minimum number of colors corresponds to 0 ($2^0 = 1$) for all three values and the maximum number of colors, full color, is 8 (all bits are significant, implies 2^{24} colors). After testing, we determined that the colors from the camera were so close in hue that the hThresholds from 2^3 to 2^8 looked the same. Thus, most adjustment occured in the sThreshold and vThreshold.

After the H, S, and V values have been thresholded the pixels is converted back to the RGB space. This conversion normally uses float values, so required careful divisions. The lack of interger conversion algorithms made this the most time consuming module to implement. There are two algorithm used for conversion of HSV to RGB. The first utilizes several comprarisons and requires multiple division and mods. The second require a single division and mod initially from which the other values are calculated. We used the second algorithm due to lower complexity and less division, resulting in less latency. We had divide the original H value by 43 and obtain the remainder. In order to mimic the float operations, we multiplied the H value by 6 and increased increased the number of bits of this value to 16. We then divided by 256 by a left shift and took the eight most significant bits of this number, which is the remainder of the original H value divided by 43. As verilog naturally rounds we obtain the interger value by scaling the H value by 6 and then dividing by 256. We then calculated P, Q, T values, utililizing the correct division by 256 to maintain 8 bit numbers. Finally, based on the division of the h value, we set the P, Q, T, and V values as the thresholded RGB values. For a more detailed view of the algorithm, see the verilog module hsv2rgb.v. The output of these module can be seen in figure 7 below.

The final module is a contrast and brightness filters, which takes the thresholded pixel values and adds contrast and brightness to the image. This creates a better looking image. This module will be explained further in the Extensions section below.

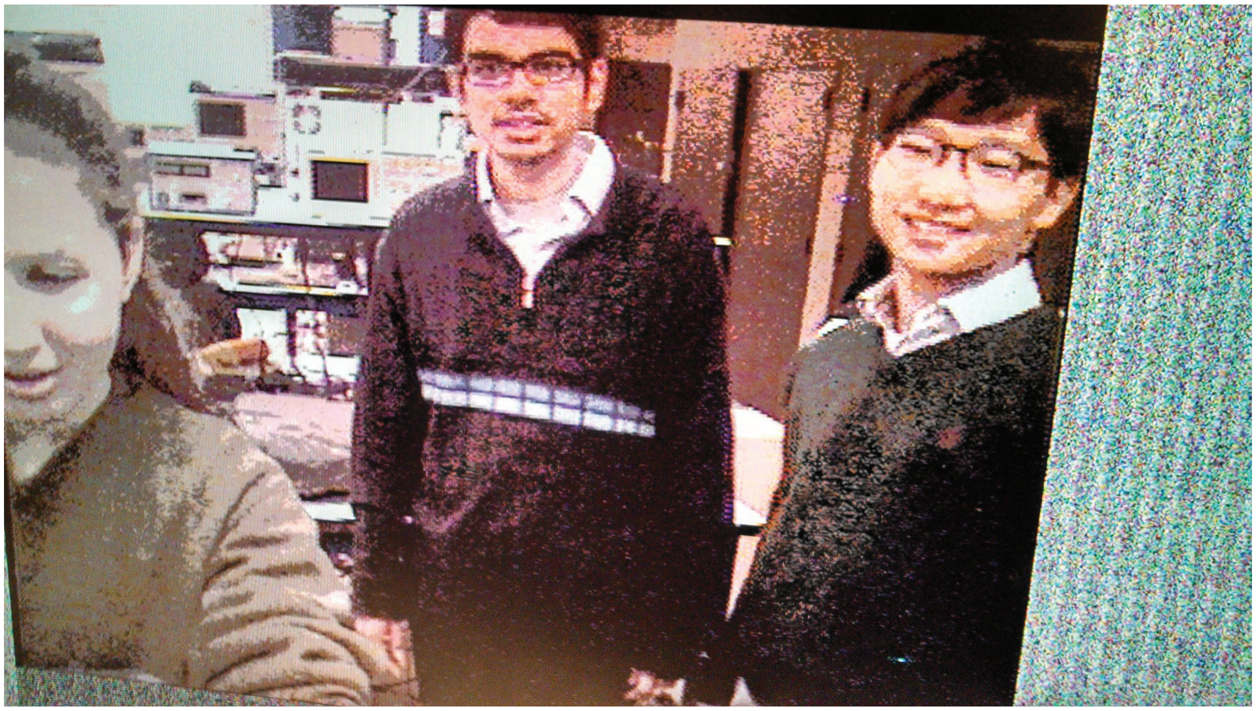


Figure 7: Color Redution Output Image (No Contrast and Brightness Filter)
An image that has taken NTSC camera data and reduced the number of colors, creating blocks of color to give the image more cartoon like effects.

3.4 Image Fuser (David):

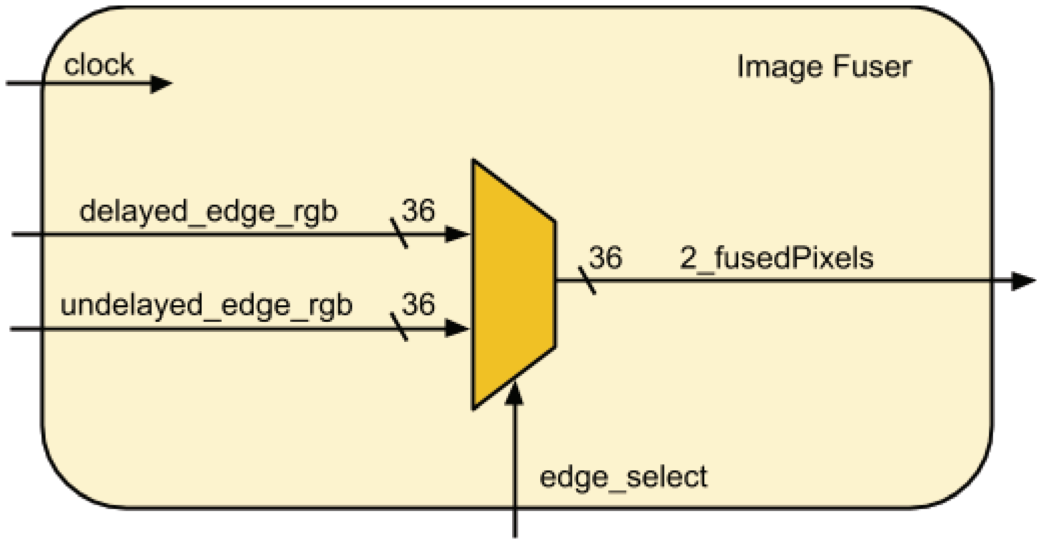


Figure 8: Image Fuser Block Diagram

The image fuser block diagram depicts the flow of the image data taken in from the Edge Detection and Color Reduction Modules. Which pixel data to use is chosen by a multiplexor using the edge_select generated by the Edge Detection module. Due the need to stored to pixels per memory address in ZBT1, this module assigns two pixels' worth of data each clock cycle.

The Image Fuser is a multiplexer that chooses between the pixel values generated by the Edge Detection and the pixel values generated by the Color Reduction. The selector bit is sent from the Edge Detection module. The output of this module is stored in the ZBT1, which sends values to the VGA output. Two pixels' data is stored in each location of the ZBT1, so the the Image Fuser selects between two pixels at a time. The final fused image (utilizing all the extensions we implemented) is shown in Figure 7.

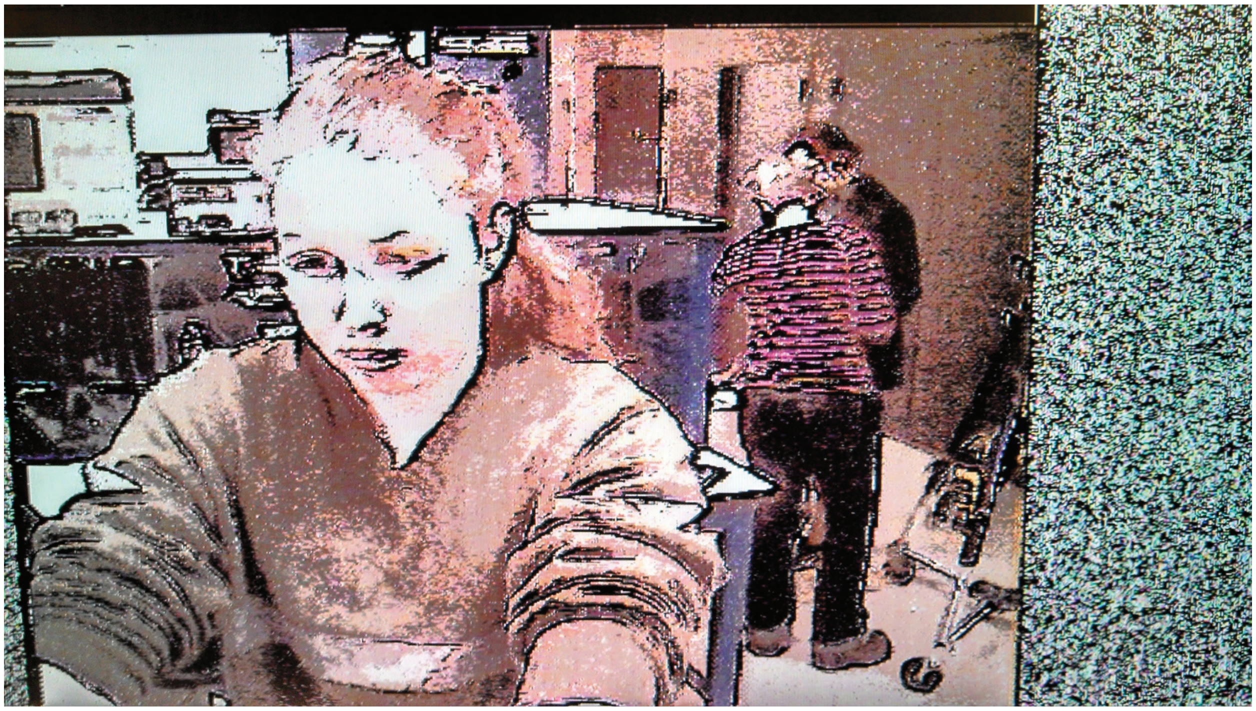


Figure 9: Fused Image Output

This image shown the final output of our project. Both Color Reduction and Edge Detection have been implemented and the fused result is shown on the VGA monitor.

4. Extensions

We have implemented two extensions beyond our initial project proposal, a **5x5 Sobel Kernel** and a **Contrast and Brightness Filter**. We believe these extensions add to the overall effect of the cartoonifier.

4.1 5x5 Sobel (Tarun):

To optimize the edge detection operation, we masked a 5x5 Sobel kernel on to the image. It was based on the same concept as a 3x3 sobel, however required a storage of five rows of pixel data. The kernels used for a 5x5 sobel are:

-1	-4	-6	-4	-1
-2	-8	-12	-8	-2
0	0	0	0	0
2	8	12	8	2
1	4	6	4	1

1	2	0	-2	-1
4	8	0	-8	-4
6	12	0	-12	-6
4	8	0	-8	-4
1	2	0	-2	-1

A 5x5 sobel edge detection had a better performance than a 3x3 sobel. It also had a better resolution.

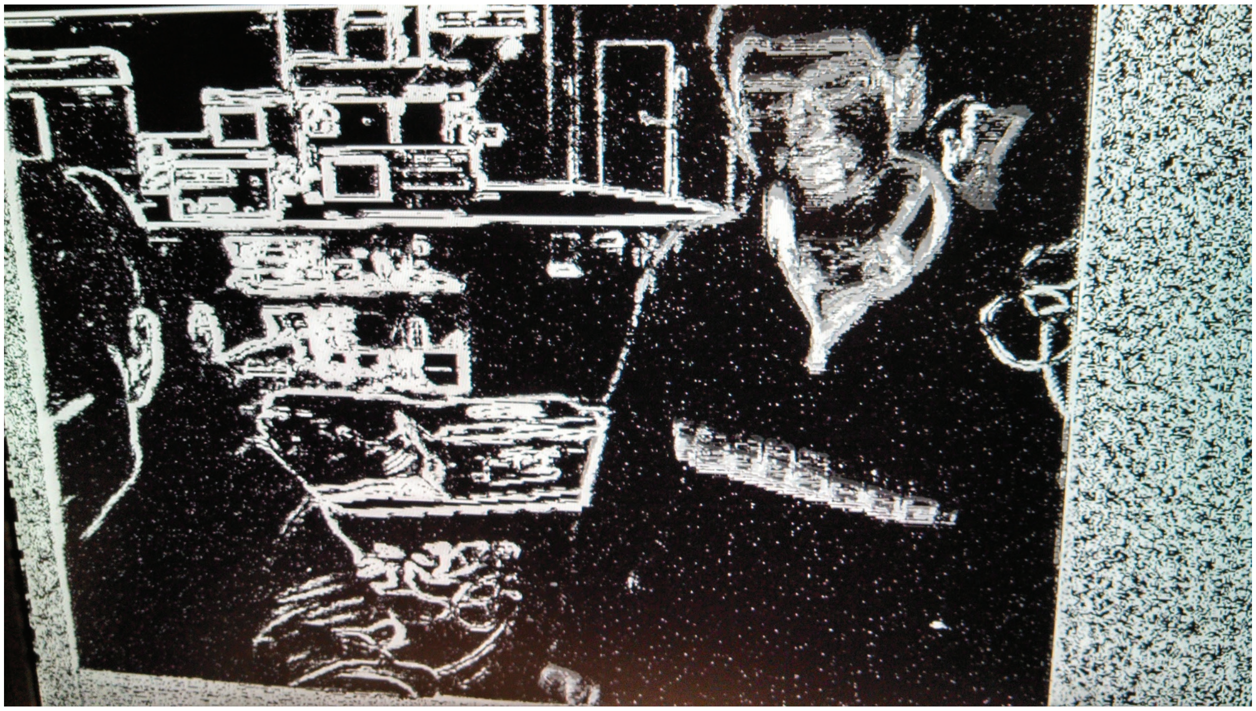


Figure 10: 5x5 Sobel Edge Detection Output

Diagram showing the output of the 5x5 sobel operator. This picture shows a more sensitive edge detection resulting in better defined edges.

4.2. Contrast and Brightness (Ariana):

The contrast and brightness filter was added to the end of the color reduction module due to the dissatisfaction with the dull image of the low-quality camera data and thresholding of the colors. Changing the contrast and brightness off the pixels in RGB space requires a linear transform of each pixel. A factor dictating contrast, alpha, is multiplied by each of the R, G, and B values and then added to a factor dictating contrast, beta. A check is implemented to ensure that these new values do not surpase 255 and thus wrap around, back to zero.

The user can choose whether to use the Contrast and Brightness filter with a switch on the labkit (switch[7]). This gives the user more control over the final image.

The effect of this module is a brighter image that we believe has more realistic colors and looks more cartoonish. It offsets the dimming done by the thresholding and creates fuller color then the NTSC camera initially provides. These changes can be observed by contrasting figure 5 above with figure 8 below.

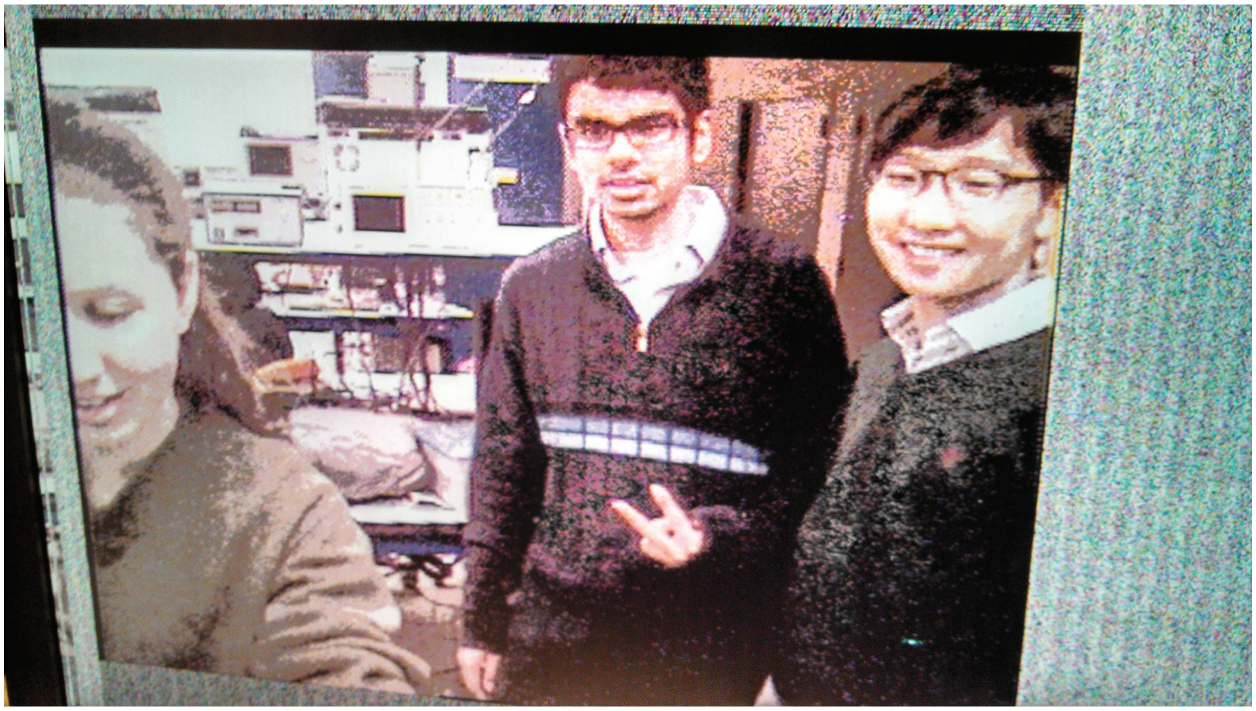


Figure 11: Color Redution Output Image with Contrast and Brightness Filter
A color reduce image that has increased the contrast (1.5) and brightness (16) factors to create a better looking image.

5. Further Extensions and Modifications

If we had time and more sophisticated tools, we would have liked to implement the following extensions and modifications to our project.

First, we would have added user inputs to many more modules. Currently, our project uses all eight switches and several buttons. Had we more switches, we would probably make the threshold values in the Edge Detection Module variable as well as allowed the user to select the Contrast and Brightness that he or she desired. These extension give the user more control as well as decrease our debugging time. We quickly found out that the images we worked with in our Matlab models were much higher quality than the images produced by the camera. Thus the values we had predetermined no longer held and we had to experimentally determine several values through extended testing, hampered by long verilog compile times.

Second, an initial extension we had planned for was a halftoning filter. In order to create the effect, divide the image into blocks of 5x5 pixels. We would then use the gradient sum calculated in the Edge Detection

Module to determine the size of the circle to be used as per the halftone effect. From this size we would generate a map of pixel colors (black or white). We would then use these values to determine the halftone color of the pixel. These blacks and whites would be alpha blended with the color data in the Image Fuser before being muxed with the Edge Detection. While we understand the follow of data through this module, the asynchronous spurts of pixel data it provided would have necessitated a large block RAM to ensure synchronous delievery of pixels to the Image Fuser. In our project's current iteration, we use 85% of available space on the FPGA and did not believe the remaining 15% would be enough for the block RAM we would require. While utilizing other memory was an option, the proximity of the deadline caused us to abandon this extension.

7. Conclusion

In conclusion, we completed our project goals and were able to add on some extensions to increase the image quality. We showed a good cartoonified image can be produced with only hardware and all calculations can be done in real time. A key to completeing this project was good time management and willingness to start work early.

8. References

http://www.ijetae.com/files/Volume2Issue1/IJETAE_0112_51.pdf

Appendix:

A. Camera Input to ZBT Interface

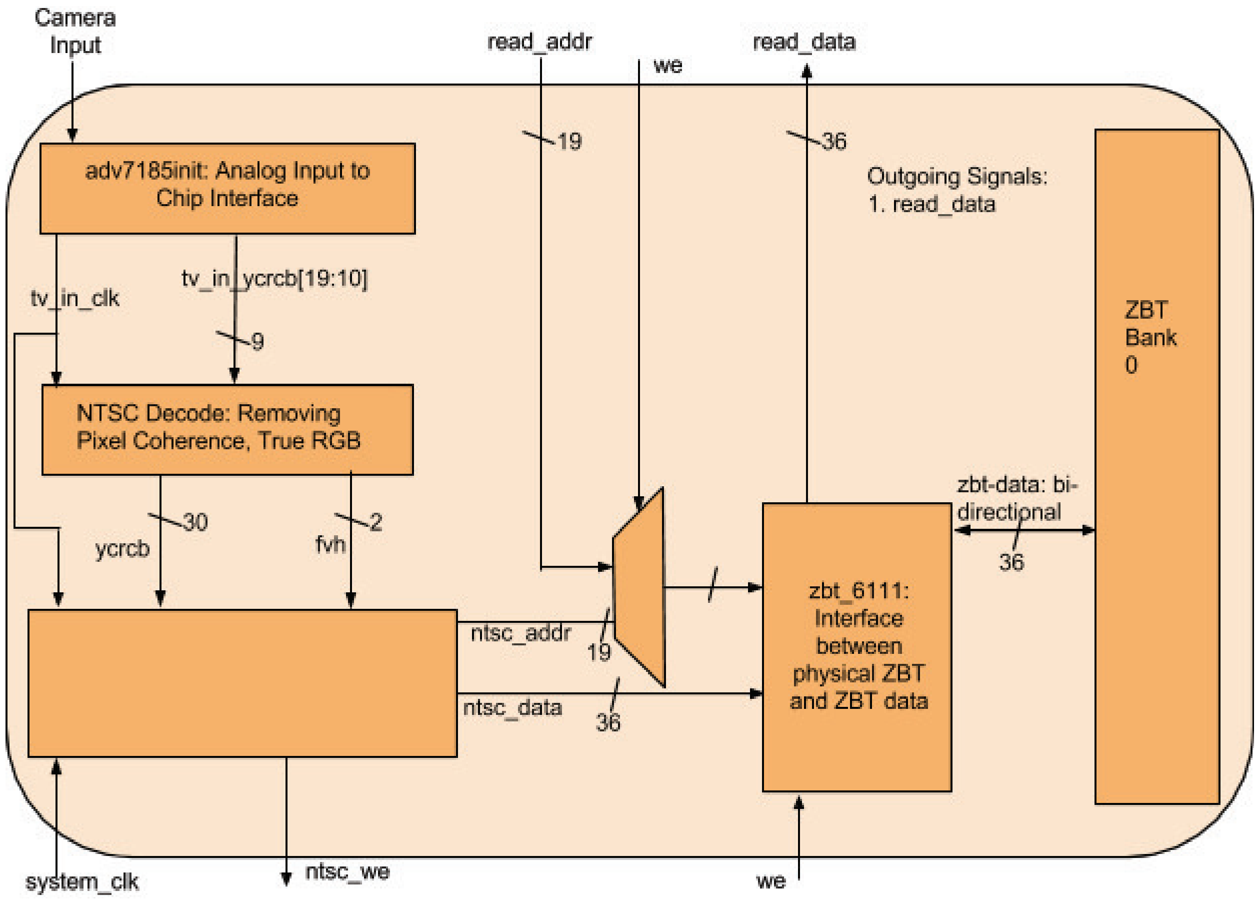


Figure 5: Camera Input to ZBT Block Diagram

The camera input to the ZBT block diagram depicts the flow of data from the camera to the first ZBT. External signals to this module are: system_clk, the 65 MHz Clock; read_addr, some address generated; and we, the write_enable signal. The outgoing signals are read_data.

B. Verilog

B.i. ZBT0 Interface, NTSC Camera to ZBT (David):

B.ii. Edge Detection (Tarun):

B.iii. Color Reduction (Ariana):

B.iv. Image Fuser, ZBT1 Interface (David):