6.111 Final Project

The Video Vitalizer - an NTSC Video System for Quadcopter Applications

J. Abel, JT. McGuire - Fall 2019

Contents

Project Overview

The goal of this project is to design and implement an NTSC camera system that could be used as an FPV camera on a quadcopter. The system takes in a video feed from an NTSC camera, and displays it in real time on a VGA monitor. Additionally, filters can be applied to the image, which enhance the image quality in adverse conditions and improve the flying experience for the drone pilot. The fact that the computations needed to apply these filters must occur in real-time makes this problem ideally suited to being implemented with an FPGA.

The board receives the image from the camera as an NTSC composite waveform, which is sampled by an ADV7185 NTSC video decoder. As per the NTSC standard, the received image is interlaced, and is received at a frequency of 30 Hz and a resolution of 640 TV lines per frame. The chip then encodes the video data as a 10 bit value containing luminance and chroma information in YCbCr color space. These values are extracted from this 10 bit value, and stored in a frame buffer inside a ZBT SRAM memory. The frame buffer is necessary to account for the interlaced nature of the incoming image, since the image processing filters require chunks of adjacent pixels to function. Once an entire frame has been stored, the output pixels are read into a circular buffer of four video lines, with one line being the active write buffer and the other three providing the data for the 3x3 pixel output kernel. The three read line buffers are latched into a series of shift registers on the output clock rate such that an output kernel can be generated with a height of three (each line buffer) and a width of three (three pixel-deep shift registers). This configuration means that the pixel currently being read from the frame buffer is actually two lines beneath and two pixels ahead of the pixel that is being output to the filtering modules.

For each pixel in the image, the image processing modules receive a 3x3 kernel of luminance data from adjacent pixels, which allows convolutional filtering to be applied to the image. Four convolutional filters are available to be applied to the image, as described below:

- An edge detection filter, which uses a Sobel kernel to highlight sharp transitions in brightness
- A noise reduction filter, which uses a median filter to remove salt and pepper noise
- A sharpening filter, which uses an unsharp mask to clarify a blurry image
- A sun blocking filter, which uses thresholding to improve image clarity in direct sunlight by blocking overly bright pixels

In addition to convolutional filter, a separate filter exists that uses linear mapping to adjust the brightness levels, hence improving the readability of the image in poor lighting. This adjustment level can either be set manually or automatically based on the mean brightness of the image.The filtered pixels are then converted to RGB color space, and are displayed on a monitor using a VGA interface. A text overlay is also applied to

the image at this stage to display the current filter state and overall brightness of the frame.

Hardware Setup

The hardware setup consists of the following key components:

- An NTSC camera, which captures images in real-time and transmits them as an interlaced analog composite waveform (CVBS video signal)
- A VGA monitor, which displays the image from the camera in real time. The monitor is able to display an image with a frame rate up to 1280x1024 pixels @ 60 Hz. The final prototype had an output of 640x480 pixels @ 60 Hz.
- The 6.111 Labkit, centered around a Xilinx XC2V6000 Virtex 2 series FPGA. The Labkit contains the following periphery chips, which are used to interface the FPGA with external devices
    - An ADV7185 NTSC video decoder, which digitizes the incoming NTSC signal and extracts pixel information in YCrCb space. It outputs this data in a CCIR656 4:2:2 color standard.
    - A 512kx36 bit Cypress ZBT synchronous SRAMi, which is used to store the incoming frame in a frame buffer to account for the interlaced nature of NTSC video.
    - A 24 bit high speed video DAC for VGA output to a monitor

The intended application of our system forced us to use some legacy hardware due to the requirement for a large, high-bandwidth memory that could be simultaneously written to and read from within a single pixel interval. The DDR RAM on the Nexys4 board requires several clock cycles in order to switch between read and write modes and therefore was insufficient for out purposes. We required a Zero Bus Turnaround [ZBT] memory in order to interlace our reads and writes to make the project possible with a single external memory. This need arises because we have to be able to save incoming pixel data while also reading outgoing pixel data within a single pixel clock cycle.

The Virtex 2 series of FPGAs has a 2002 vintage and the newest software that still supports them was released in 2008 and last updated in 2011. This need to use legacy hardware caused us a great deal of stress and strife, and we strongly recommend to all future students that the old labkit be avoided at all costs. Xilinx ISE 10.1 does not support Windows 10 and only the 32-bit version of the software was usable. However, the driver incompatibility meant that we could only use ISE on Windows for simulation. To actually interface with the hardware, we had to use Linux. Fortunately, the debian athena machines in the lab are capable of running ISE with reasonable effectiveness, though the software is still prone to crashing and strange file errors. ISE 10.1 definitely does not play nicely with the newest version of Ubuntu (we checked).

ISE 10.1 proved to be our biggest challenge in implementation. It lacked some programmable logic IP that is present in Vivado and the Nexys4 that would have been incredibly useful, most notably an internal logic analyzer and a functional clock generator. The lack of these features made timing control and debugging more difficult than it would have been on a newer board. We did not want to mess with Chipscope for obvious reasons and instead used the physical logic analyzer in the lab, which was not quite fast enough to accurately receive our 108 MHz memory interface signals reliably over its lengthy wires. Additional delays were caused by the ISE software itself, which was unstable even on the lab machines and would occasionally interpret Verilog code slightly differently to Vivado, leading to unexpected results. We found that ISE was often unable to delete configuration files from the computer RAM for IP after first creation so that one could not alter any generated modules without a full system reboot. It had many errors and really took user input more as a guideline than a rule for anything related to IP. We had major difficulty in generating digital clock modules especially, and were only able to finally produce properly configured DCMs by skipping the IP generator and giving express directives in Verilog. Timing issues were the bane of our project.

Most hobbyist FPV quadcopter camera systems send video using analog NTSC broadcast signal on a 5.8 GHz RF carrier. To maintain compatibility with this type of transmission, we needed to adhere to the NTSC video input specification. This signal is digitized by the ADV7185 video decoder IC before being received by our system. Since NTSC video signal is sent field-by-field in interlaced format, a frame must first be stored in a buffer before any convolutional processing can take place (which requires kernels of adjacent pixels).

Implementation

In part due to the issues described above, the project was not able to be implemented as intended in the time available. Hence, for the purposes of demonstrating the functionality of all written Verilog modules, two independent systems were built; one to demonstrate the memory control and NTSC signal conversion, and another to demonstrate the image processing and user interface.

To demonstrate the camera input and memory control modules, a system was built to display an image from an NTSC camera on a VGA monitor using the old Labikt. This system only included two 3x3 convolution filters to demonstrate how kernels of adjacent pixels are read from the ZBT memory. This module was implemented in two ways. Our first implementation included an output module that attempted to upscale the output from 640x480 to 1280x960 using averaging. This module proved to have a spotty output and was susceptible to timing issues that we were unable to track down. The system clock rate was 108 MHz, the same speed as the dot clock required by the output specification. Thus, we needed 100% throughput from our system and had to pipeline several combinational paths to meet timing spec with this design. Our second

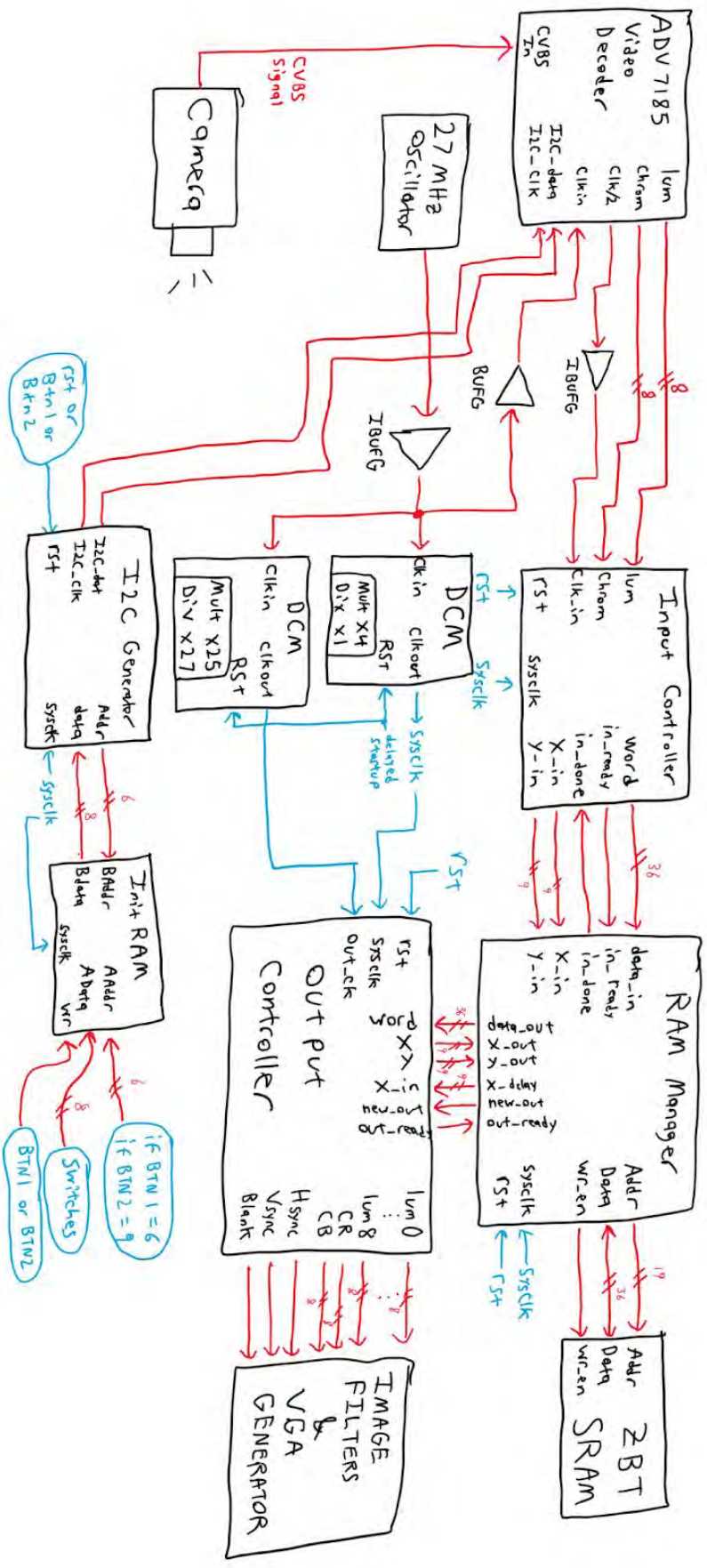
implementation was drastically simplified and included only a 3x3 output with the original 640x480 resolution. This produced a much more manageable output with fewer timing issues, no doubt helped by the pixel clock at 25 MHz, or roughly one fourth of the speed. This eased our already tight frame buffer memory bandwidth constraint because we were able to keep the system clock rate (and thus frame buffer access rate) at the same 108 MHz. This simplified 3x3 system was the same one described in the checklist and the original project outline, so it meets the project goals sufficiently. We managed to get it to full functionality, though it did still have some light ghosting and memory timing issues that caused sliding lines to appear on the displayed image. We will discuss the deficiencies of the implementation more in the reflection section of this paper.

To demonstrate the image processing features of this system, a simulated front end was built on a Nexys 4 FPGA. It was chosen to demonstrate these features on the Nexys 4 instead of the old Labkit due to its faster build time and more reliable programming interface, under the assumption that any logic written in Verilog would be portable. Due to the limited memory available on the Nexys 4, this system could only display a single image at a reduced resolution (640x480 pixels per frame, 14 bits per pixel) that was stored in a BRAM. Additionally, clock speed limitations meant a 3x3 kernel was generated for each pixel as opposed to a 5x5 kernel as originally designed, resulting in a slightly simplified implementation of some convolutional filters. Nevertheless, this system was able to demonstrate the effect of all four possible convolutional filters and the level adjustment filter on the static image. The application of these filters was controlled via the buttons on the FPGA and a text-based on screen user interface.

Testing and Simulation

A critical aspect of development in our project was design verification through simulation. I (JT) was fortunate enough to get the 32-bit version of ISE 10.1 running on my Windows 10 laptop and could therefore conduct simulations outside of the 6.111 lab. Though the Xilinx simulator never worked, I managed to download ModelSim as well and used that software for simulation purposes to tweak the timing of my design. I am happy to say that because of this testing, the video input and I2C modules both worked flawlessly on the first implementation try. The frame buffer manager and output modules were not so lucky, but this was more due to timing and clocking issues resulting from poor Xilinx software and my assumptions about how the ZBT RAM would work. The simulations showed that the output would be totally functional, but simulation timing and real life timing are two very different things as we learned. For brevity's sake, we have not included the Verilog testbench code used to simulate our designs, but we have submitted it on the course website.

Design Overview: Block Diagram



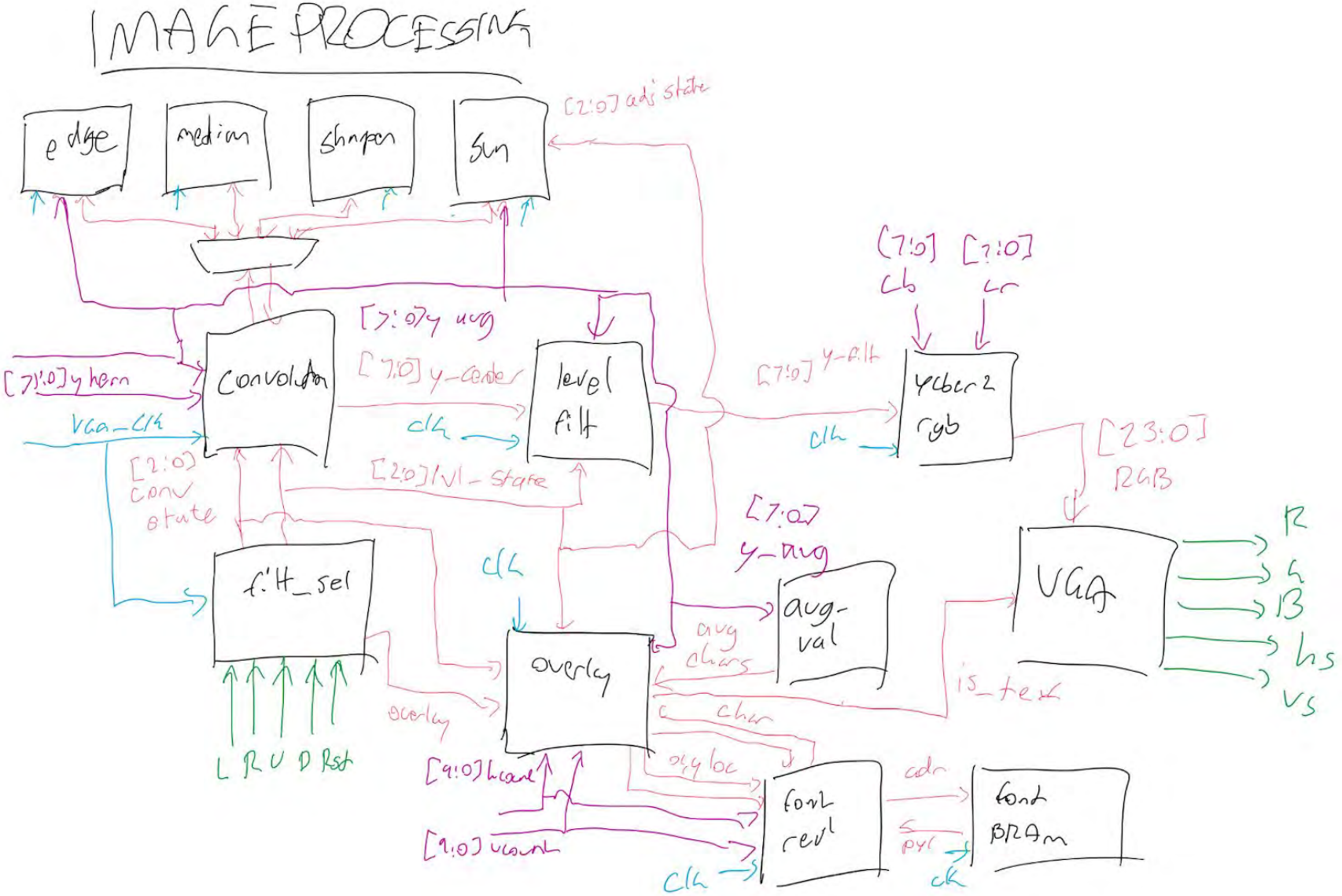


IMAGE PROCESSING
edge
medium
sharpen
sum
[2:0] adj state
[7:0] y avg
[7:0] y_center
[7:0] y_filt
[7:0] y kern
vga_clk
convolution
level filt
clk
[2:0] lvl_state
[2:0] conv state
filt_sel
[7:0] y_avg
avg_val
avg chars
overlay
char
char_y_loc
[9:0] hcount
[9:0] vcount
font rom
font BRAM
addr
pixel
clk
[7:0] Cb
[7:0] Cr
ycbcr2 rgb
[23:0] RGB
VGA
R
G
B
hs
vs
is_text
L R U D Rst

Memory Control & Camera Interface - JT. McGuire

NTSC Input Decoding

The input CVBS video received from the camera was digitized by means of an ADV7185 video decoder IC on the labkit board. This chip latches out digital data in the CCIR656 4:2:2 standard which essentially means that color data rate is half of luminance data rate. This is acceptable because the human eye is much more sensitive to light intensity than light color. We designed our system to receive 16-bit wide output from this chip at half of the typical CCIR656 clock rate of 27 MHz. With this method, the input clock rate is 13.5 MHz and includes luminance data for consecutive pixels on 8 of the 16 bits. The other 8 bits have multiplexed color data for every other pixel such that 8 bits of red channel and 8 bits of blue channel data are sent for every two luminance values.

To control input video timing, the CCIR656 standard describes specialized codes that are sent within the data stream that indicate transitions between states for video field, horizontal blanking, and vertical blanking. The receiver module therefore required a state machine to trigger off of these codes to perform tasks such as switching vertical and horizontal indices or initiating or ending the data output to the frame buffer control module. The input module was configured with a multiple-register delay. This allowed the start and stop codes to be recognized on the oldest registers and the encoded data to be read from the one-cycle newer registers. This was a simple way of extracting the video timing for triggering the state machine.

Another state machine had to be implemented for this module to upscale the color data in the image. Because the color channels were only received for every other pixel, the color information for off-pixels was computed as the average of adjacent pixels. This required a four-register delay of input information such that the red and blue channels could be received for the two adjacent pixels. If the oldest pixel is number 4 and the newest is number 1 in the registers, The full color for pixel 3 was computed using the red channels received with pixels 4 and 2 and the blue channels with pixels 3 and 1. Pixels 2 and 4 already had all color data, and pixel one would not be yet computable (Fig 1).

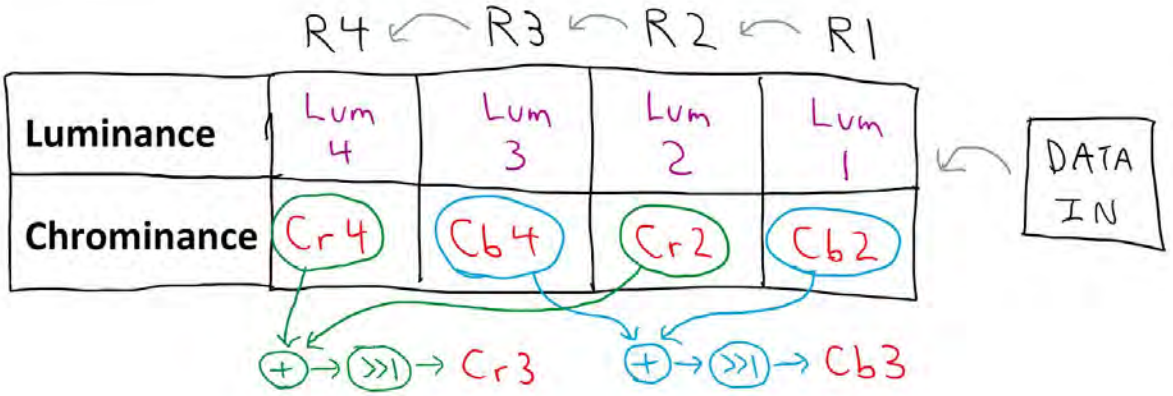
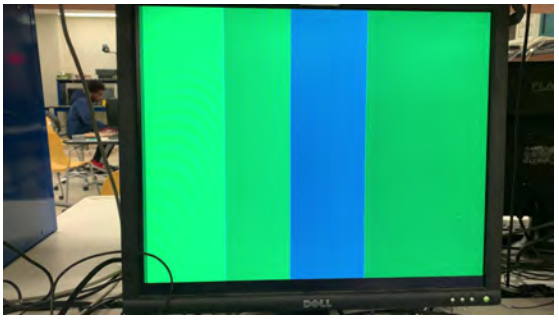


Figure 1 - Visualized chroma upscaling method

The input module latched the upscaled color data into output registers at the rate of reception. The data was rescaled into 36 bit words, which is the width of the frame buffer memory. Each word contained two pixels of information, with 8 bits dedicated to luminance data and 10 bits to color data (5 for each channel). A control line was triggered with each new input data to inform the frame buffer controller that a write needed to occur. The frame buffer controller also had a response line to indicate when the write was initiated. If the control line was ever triggered twice without a response, an overflow fault was indicated and the system reset. The clocking of the input module was purposefully chosen to be exactly one eighth of the system clock rate such that no syncing registers would be required. The input module also kept track of the vertical and horizontal counts for the current output pixel and latched out the nine bit X and Y position signals as appropriate to the memory module.

Colorbar Generator



As a system testing method, it was important to create a colorbar generator module to verify that the input and output was timed properly. This enabled us to visualize skew in the output and physically see horizontal timing issues in the implementation. The colorbar generator merely produced different pixel luminance and color values as the output progressed across the horizontal line. It also replicated the frame save signal trigger that was output by the video input module.

I2C Control

To enable the output mode that we used with the ADV7185, we had to write an I2C interface module to program the configuration registers on the video encoder. This was essentially a long and complicated state machine that pushed data across the

output lines using the I2C standard. It was designed to read from a BRAM configured as a dual port PROM for easy reconfiguration of the registers. The BRAM held three 8-bit values for every register to be programmed: the slave address for the ADV7185, the subaddress for the memory register, and the actual value to program. The state machine produced a start condition, latched out the 8 bits, received the ack bit from the slave, and then sent the stop condition for each byte sent at a bitrate of 20 kHz. For proper configuration, we had to change one bit value in one register in the ADV7185 at every system startup.

This module also included programmable video brightness and saturation via the same I2C interface with the ADV7185 module. This chip included these adjustments natively and was controlled by the setting of memory registers. Thus, we used a series of buttons and switches to write values to the PROM and to retrigger the I2C output such that new brightness and saturation levels could be set on the fly during operation (Fig 2).

Figure 2 - Comparison of normal, high-brightness, and low saturation images from left to right (modified using I2C module for ADV7185 register control)

ZBT Memory Control

The frame buffer control module was deceptively simple in theory and incredibly complicated in practice due to timing constraints. It operated on the 108 MHz system clock rate, interlacing read and write data onto the ZBT memory data lines while computing the proper addresses. The address computation had to include a hardware multiplier to compute the proper location for the target pixel with the equation Y*FRAME_WIDTH+X. Because of the 108 MHz clock rate and the additions and multiplexing that had to occur for address computation from the input and the output modules, this had to be implemented in three pipeline stages. Additionally, the ZBT SRAM itself had a two stage pipeline delay from input to output, meaning that reads had a total latency of five clock cycles through the system.

Read and write control was implemented using the synchronous write control lines on the ZBT RAM. However, because the data lines were multiplexed for both reading and writing, the FPGA had to be told when to set the ports to high impedance mode for data reception and when to set the ports to drive mode for writing. This needed to occur two clock cycles after a write was indicated. This module in essence became a massive series of shift registers for input signals and assignments to output signals such that the proper operations were conducted at the proper times for both

reads and writes. Luckily, the ZBT RAM had a nifty feature wherein it was automatically configured to put its data port into high impedance mode whenever a write was indicated at the proper time, provided the output enable pin was held low. Thus, the FPGA remained in high impedance mode and the RAM in drive mode by default, with the roles switching when a write was indicated. See figure 3 for a sample timing diagram.

The interlacing method gave precedence to the output module and always prioritized reads over writes because reads happened twice as fast due to the 60 Hz output. Additionally, write data and address information was always saved into a latch that was guaranteed for eight cycles, so address and data sequencing was much less critical for writes and could be reasonably ignored. Reads, on the other hand, required shift registers for all signals to ensure proper timing. Thus, a high output ready trigger would be shifted back five times to trigger the output capture word latch for each incoming read address. Incoming X data was also shifted back five times to produce an X output that was synchronized to the output data, simplifying the output module design. If a read was not triggered on a given clock cycle and a write was, the system would raise the write-initiated line to prevent overflow and service the write by triggering the write-enable output for the ZBT RAM after three clock cycles of address computation. Then it would put the FPGA data port into drive mode two clock cycles later to write the output data to the RAM.

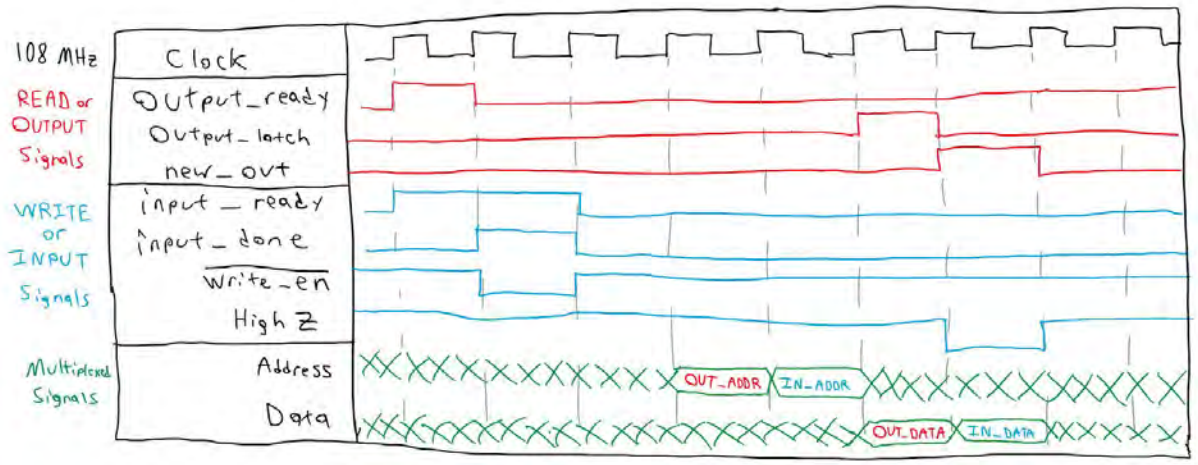


Figure 3 - Sample timing diagram of typical interleaved read/write operations

This module was a mess of timing. We know it is functional because we conclusively proved by suspending input that it was properly writing and reading to the memory. However, the output contains shifting lines when the camera is engaged that are not present with suspended input, and this leads us to conclude that there are still some subtle timing issues where writes are somehow affecting reads when they should be independent. We unfortunately did not have the time to debug the module, and the physical logic analyzer does not even have the speed to do so effectively. To find this

error, I firmly believe that we would need to switch to a completely different FPGA such that we could use an integrated logic analyzer and software that was not flaming hot garbage.

Output Module

The output module went through two design iterations. The first iteration was more ambitious and involved an image upscaling computation from 640x480 to 1280x1024 that drastically increased complexity. It will be described with less detail because it did not come to full fruition. The second iteration was the design as implemented and did achieve full functionality with a 3x3 kernel and straight 640x480 output. For all iterations, full kernels were only generated for luminance data, and chrominance data was only output for the center pixel of the kernel. This step was taken to reduce complexity, as we deemed filtering of the color data an unnecessary step due to the lack of sensitivity to it for human viewers.

The upscaling module worked through a series of five line buffers built from BRAMs that each saved a single line of video data, or 312 thirty-six-bit words in our implementation (625 pixels input became 312 words in the input module). These were written to with an index and switched in a circular style, with the index pointing to the line that was the active read buffer from the frame stored on the ZBT RAM. This circular structure had to be faked with large multiplexers, as BRAMs are not actually configured with indices that wrap around like a real circular buffer. The multiplexers connected the proper line buffer data outputs to intermediate wires for the previous four consecutive video lines that occupied the circular frame buffer. The index incremented with each new line and the multiplexers switched the intermediate wires to point to the new consecutive buffer data outputs. In this way, four consecutive lines could always be read simultaneously from four video line buffers while the fifth buffer was used to capture the next line from the external frame buffer on the ZBT SRAM. From here, the four line buffer outputs were each sent through shift registers with four cycle depth to generate a 4x4 array of registers that could be simultaneously read for output. This was fed into a vast array of multiplexers, shifters, and adders that computed pixel values “in between” the actual values for upscaling, and a state machine controlled the multiplexers to output these values alternately to generate a full 5x5 kernel of luminance data that was sent to the image processing modules. We found that this upscaled 5x5 output kernel with a 108 MHz dot clock had an impressive output data rate of 24 gigabits per second.

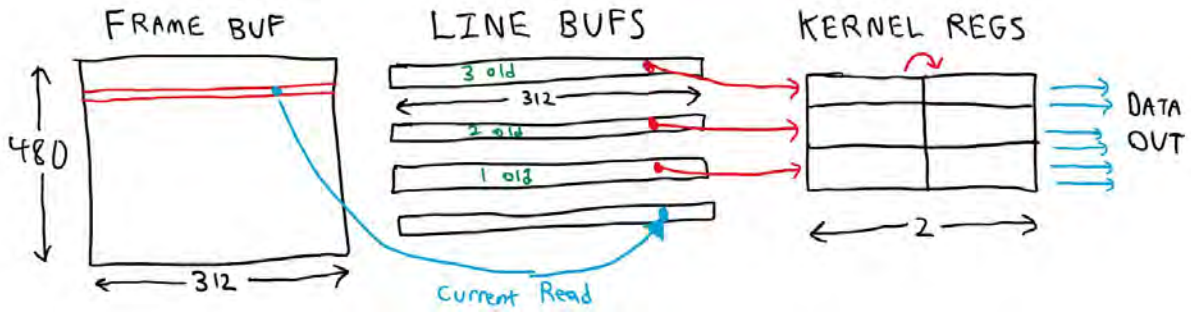


Figure 4 - Diagram of the output memory architecture and data flow paths

Unfortunately, we were not able to get the upscaled system to a fully functional state. We think this was the case chiefly because of timing issues. In our second system, we removed the upscaling and dropped to a 3x3 output kernel for simplicity. We think that the further timing modifications made with the simplified system may have enabled full operation of the upscaled system as well, but we were unable to test this theory due to time constraints.

The 640x480 version used a very similar structure to the upscaled 5x5 version. However, it required only four line buffers instead of five (three to read from and one to write to) and a register depth of two instead of four (two words = four pixels of depth) (Fig 4). Additionally, we were able to skip the upscaling computations and seriously simplify the state machine that controlled the multiplexers for output. The output merely had to switch between even and odd pixels to deal with the two-pixel word storage mechanism (Fig 5). In the upscaled version, the multiplexer also had to deal with properly sequencing the half-pixels between actual pixels and the half-lines between actual lines. This was also a nightmare from a timing perspective.

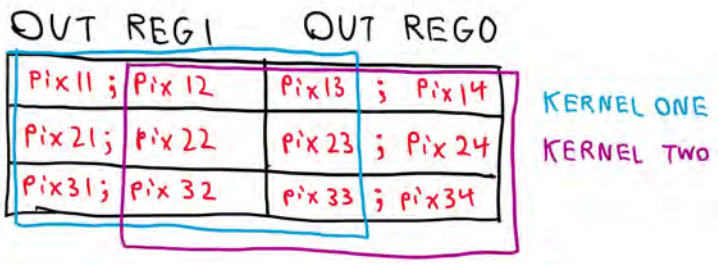


Figure 5 - Diagram of output kernel multiplexing for two-pixel word storage

The multiple line buffer and shift register structure that we implemented for kernel output was an effective way of reducing the data rate required from the ZBT memory. While it would have been easier to directly read the pixel values from the frame buffer, this was impossible with our system clock speed, as there is no way that we could have read the value for each of the nine pixel values needed for each output and still had time for writing. At best, we could have possibly doubled our memory read bandwidth, but even that would have pushed the limit. This method is more robust and also proved to be scalable in that we were able to quickly refactor the code to reduce the

complexity while keeping the same structure. We were able to exploit the speed benefits of multiple BRAMs, the storage benefits of the ZBT external memory, and the repetitive nature of image kernels to achieve an output with a very high bitrate that could deal with interlaced input video signal with a different clock rate than the output.

Image Processing - J. Abel

Convolution Filtering

The majority of the image processing was performed using convolutional filters. These filters perform a mathematical operation on a square kernel of adjacent pixels, producing a single, processed pixel in the center of the array. In a strictly convolutional filter, this operation is an element-wise multiplication with another square array known as the convolutional kernel. This is the operation used for the sharpening and edge detection filters. Other implemented filters are not strictly 'convolutional' in their application, instead performing a different operation on the kernel (a median operation for the noise reduction filter, and a thresholding operation for the sun blocking filter). It is important to note that a convolutional filter is applied to an array of numbers, however a color image is an array of tuples. Hence, only one channel of the image is processed - in this case the luminance channel, which we felt to be the best representation of the overall composition of each image.

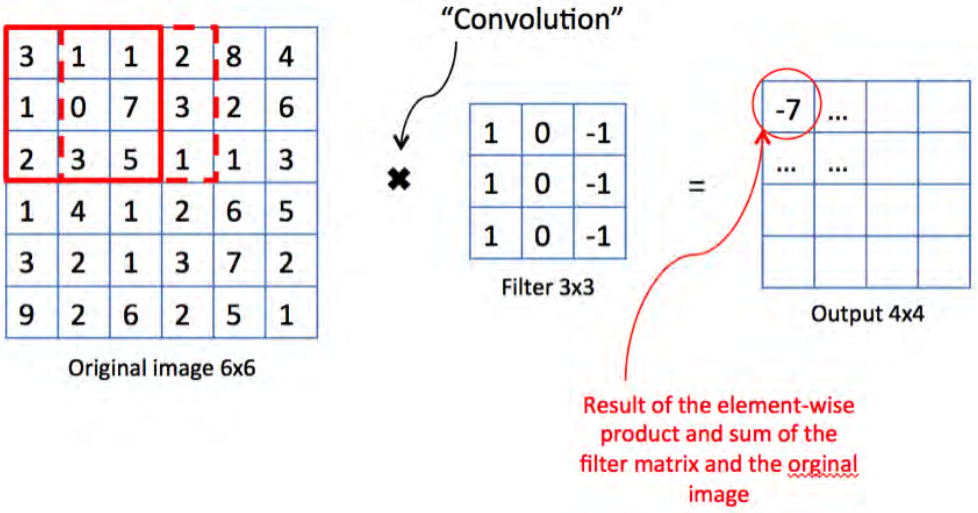


Figure 6 - an example of a simple 3x3 convolutional filter (image source - M. Cavaioni, Medium)

To implement convolutional filtering in Verilog, a 72 bit or 200 bit register (for a 3x3 or 5x5 kernel respectively) is used to represent the incoming kernel of adjacent pixels. Generating such a kernel requires a memory read clock that is significantly faster than the clock used to drive the VGA output.

Edge Detection - Sobel Filtering

The edge highlighter applies a Sobel convolution kernels to each pixel to detect and highlight sharp transitions in brightness in the image. This filter works by performing a convolution operation to each pixel, then comparing the output to a threshold value to determine the presence of a sharp transition. This filter actually requires two parallel convolution operations using convolution kernels that are 90° rotation of each other (shown below), which allows the system to detect horizontal and vertical edges.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \; G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

In Verilog, this filter was implemented by applying arithmetic operations to subsets of a 72 bit value representing the input kernel. An OR statement was used to assign an edge to the output pixel based on the thresholded outputs of the two convolution filters. This is achieved by assigning either full or zero brightness to an edge pixel, the color selected based on the aggregate brightness of the frame to maximise the visibility of the edge. Since the arithmetic for the module only consists of addition and bit shifts, it can be performed using almost exclusively combinational logic (however, registers are used to provide a one cycle delay for pipelining purposes).

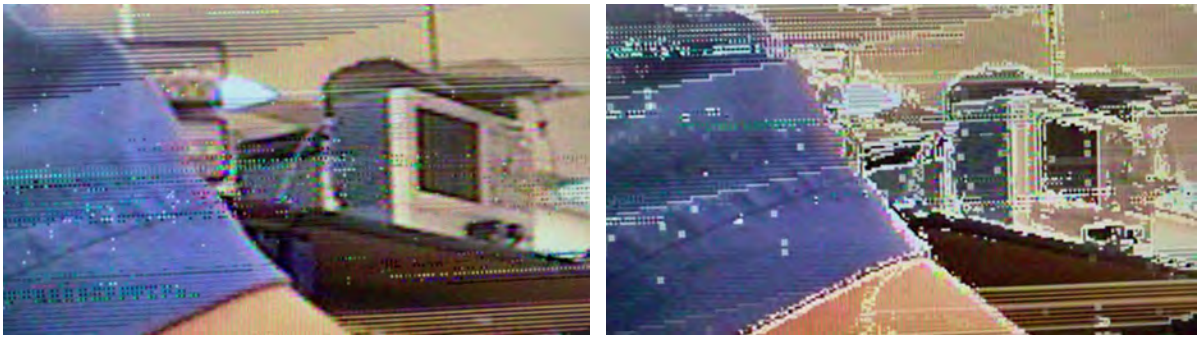


Figure 7 - Application of the Sobel filter on an image from the NTSC camera

Noise Removal - Median Filtering

A noise removal filter is included to remove salt and pepper noise from an image, which is a common problem in NTSC video. Removing this kind of noise is best done using a median filter, which outputs the median value of a 3x3 kernel of adjacent pixels. While this filter results in some overall smoothing of the image and subsequent loss in resolution, it can remove almost all salt and pepper noise.

This filter is surprisingly difficult to implement in Verilog, since there is no simple logical operator that can be used to sort a list of values. Nevertheless, it was found that the median value could be obtained using the sorting matrix shown below. Each submodule of this matrix uses combinational logic to find the median, maximum, and minimum of a set of three values. Since each pixel needs to pass through a maximum of three subfilters to find the median, this filter introduces the most delay of the four possible convolutional filters. Hence, all other filters should be pipelined relative to this filter.

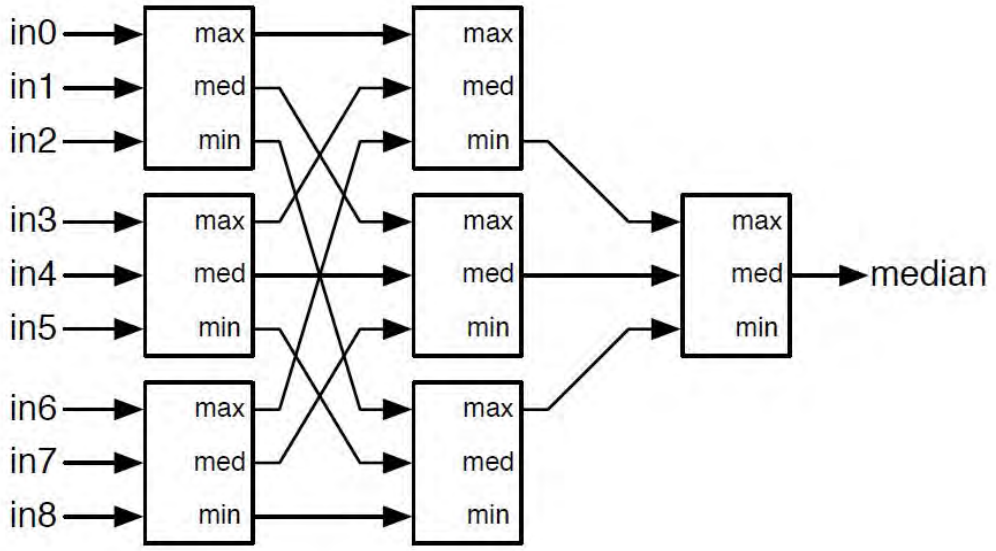


Figure 8 - A diagram of the sorting matrix used to find the median of a nine values (image source: R. Avizienis, Berkeley)

Figure 9 - An image with added salt-and-pepper noise before (left) and after (right) passing through a median filter

Sharpening - Unsharp Masking

A convolution filter is also available in this system to sharpen the image. Two different types of sharpening kernels were tested in this project: a simpler 3x3 sharpening kernel, and a 5x5 unsharp mask, which combines a sharpening operation with a Gaussian blur to create a clearer image. Because of its improved performance, we planned to implement a 5x5 unsharp mask on the completed system, which is designed to be able to output a kernel of 25 adjacent pixels for every cycle of the VGA clock. However, clock speed and memory limitations of our imaging processing test system meant that the simpler 3x3 sharpening filter was demonstrated instead.

$$G_s = \begin{bmatrix} 0 & \frac{-1}{2} & 0 \\ \frac{-1}{2} & 3 & \frac{-1}{2} \\ 0 & \frac{-1}{2} & 0 \end{bmatrix}, \; G_u = \frac{-1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & -476 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Since both of these filters require multipliers to function, this operation cannot be done without a latency of at least one clock cycle. Hence, the operations for this filter needed to be registered. In addition to performing the multiplication operations, the module performs clipping of the output to keep it within the permissible range of luminance values of a single pixel (between 16 and 235 for an 8-bit pixel).

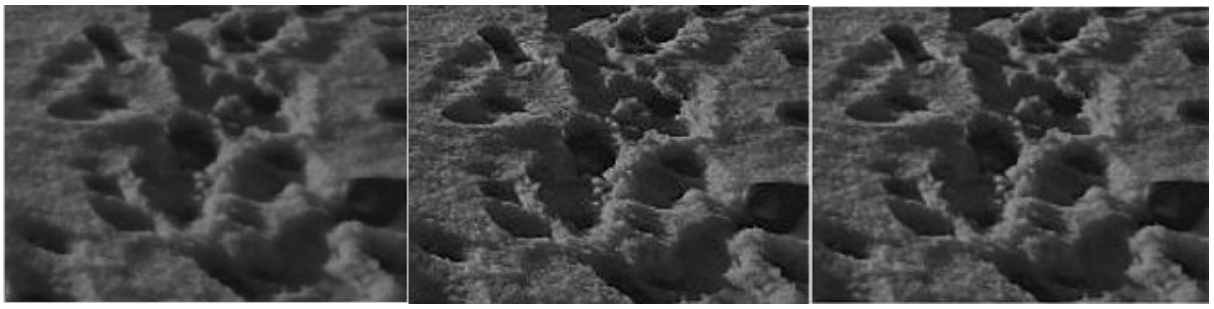


Figure 9 - The effect of a 3x3 sharpening filter (middle) and 5x5 unsharp mask (right) on a blurry raw image (middle)

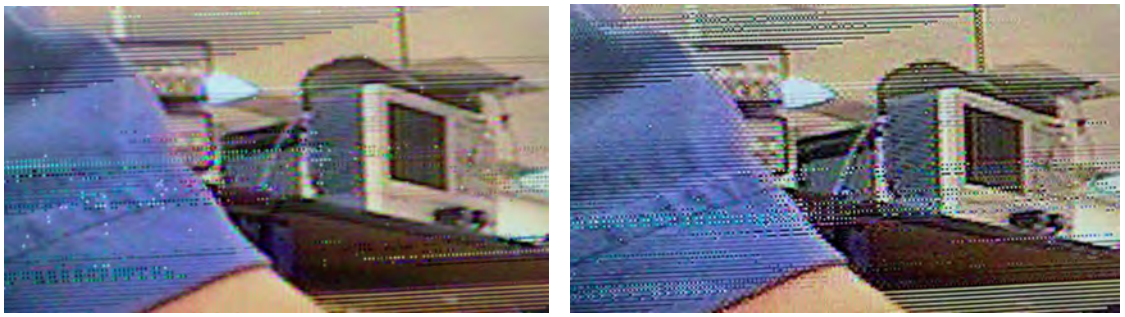


Figure 10 - Application of a 3x3 sharpening filter (right) on an image from the NTSC camera

Sun-Blocker - Threshold-Based Filtering

A common issue faced by drone-mounted FPV cameras is that their image tends to be washed out by direct sunlight. To counter this, a 'sun blocking' filter has been included, which uses thresholding to block-out pixels constituting the light source. When used in conjunction with level adjustment to darken the pixels surrounding the blacked out area, this filter can significantly improve the usability of the camera in direct sunlight. This filter assumes that a pixel that is part of a light source will have the maximum possible value of luminance. Based on this assumption, it compares the luminance value of the pixel against a high threshold, and outputs a black pixel if the threshold is exceeded.

Initially, this filter was applied on each pixel independently as part of the brightness level adjustment filter (below). However, this resulted in a noisy output. To counter this, we applied the thresholding filter to each pixel in a kernel of adjacent pixels, and output a black pixel only if at least half the pixels in the kernel exceeded the threshold. Using this method, the resultant black-out area was found to be smoother and more continuous.

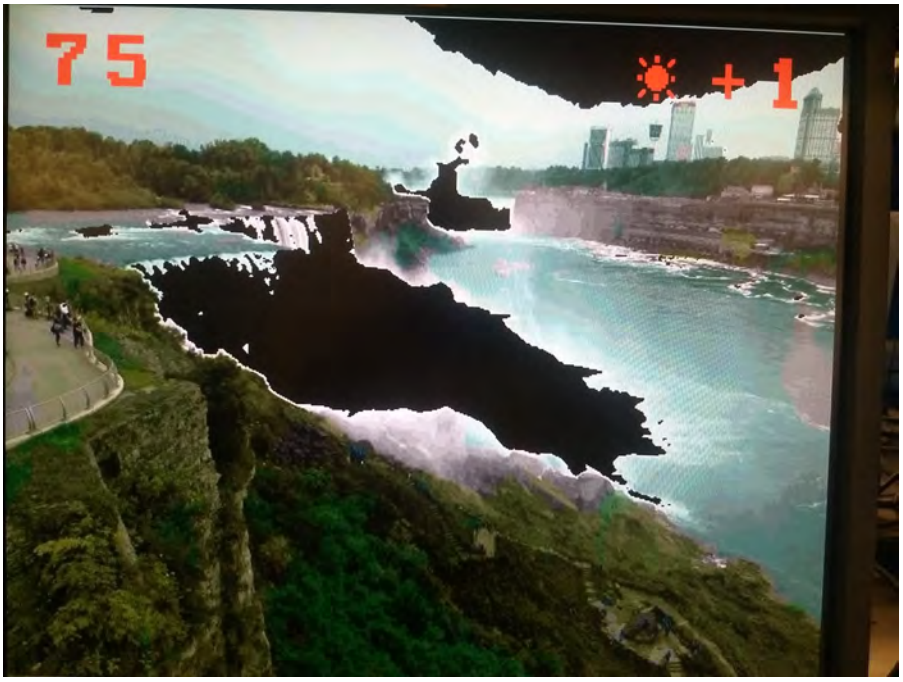


Figure 11 - The application of the sun blocking filter to an overexposed image. In this case, the system is blocking out the whitewater section of the waterfall, which is otherwise completely washed out.

Level Adjustment

Once convolution filtering has been applied to a pixel, the single output pixel then passes through a level adjustment filter. This feature is intended to supplement the brightness correction performed by the camera, which we can control using the I2C communication module, by allowing the brightness level of pixels to be adjusted after additional processing has been performed. The filter uses linear mapping to adjust the luminance value of the pixel, making the overall image brighter or darker without reducing the brightness range of the image. To increase the brightness of an image for example, pixels with a luminance values between 0 and 50% can be remapped to values between 0 and 100%, while pixels outside of this range are assigned the maximum luminance value possible. This method is a linear approximation of gamma correction, which remaps pixels on the nonlinear power curve AV_{in}^{γ}.

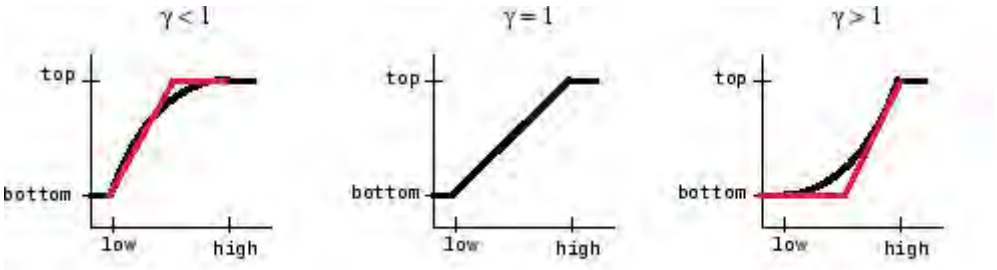


Figure 12 - Gamma correction curves to brighten (left) and darken (right) an image, with their linear approximations superimposed (image source - Mathworks)

Six different adjustment levels have been programmed into this system: three which darken the image, and three which lighten it. Each adjustment level has its own linear mapping relation, which is represented as a clipped multiplication operation in Verilog. The user can either manually set the adjustment level of the filter, or set the system to automatically apply a filter based on the average brightness of the frame.

Figure 13 - The same image with a luminance level adjustment of -1 (left) and +3 (right)

Colorspace Conversion - YCbCr to RGB

NTSC camera systems send data in a YCbCr color space, which encodes luminance data on one channel and color data as a blue and red chroma on two other channels. VGA monitors, by contrast, require data to be received in an RGB color space, with three channels representing the amount of red, green, and blue in a pixel. We elected to perform all prior processing on the luminance channel in YCbCr space, which is the format output by the camera, as it best represents the overall composition of the image. Thus, the color space conversion occurs after filtering has been applied immediately before sending the data to the VGA monitor.

Pixel data can be converted between these two color spaces using the matrix multiplication below. For implementation in Verilog, the coefficients of the matrix were rounded such that no floating point arithmetic was required. Since this multiplication includes negative numbers, signed values are used to perform this operation. It is important to note that some inputs to this function will result in values of R, G, or B that are outside the permissible range for an 8 bit pixel (0 to 255). Hence, the output must be clipped.

$$\begin{bmatrix} {}_{255}R' \\ {}_{255}G' \\ {}_{255}B' \end{bmatrix} = \frac{1}{256} \begin{bmatrix} 298.082 & 0 & 408.583 \\ 298.082 & -100.291 & -208.120 \\ 298.082 & 516.411 & 0 \end{bmatrix} \cdot \left(\begin{bmatrix} {}_{219}^{601}Y' \\ C_B \\ C_R \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \right)$$

Figure 14 - Image Source - C. Poynton, A Technical Introduction to Digital Video

Text Overlay

The application of these filters was driven by an FSM, which was in turn controlled by the buttons on the FPGA. The user can see the current state of the FSM, and hence the filters that are currently applied to the image, using a text-based user interface overlayed on top of the image. Data to generate the font for this overlay is stored in a single 1 bit wide array in a BRAM. A Verilog module takes in the current pixel location on the screen, determines which character (if any) should be displayed at that location, and reads the corresponding BRAM address to determine whether the particular pixel should show the processed image or part of the font.

In addition to displaying the state of the filter selection FSM, the overlay also displays the average frame brightness as a percentage. This requires converting the average screen brightness to a binary coded decimal, in which each digit of a decimal number is represented as a 4 bit binary value. This conversion is achieved using the double-dabble algorithm, which is iteratively applied to the input value using the following steps:

1. Create a 4 bit register for each digit of the number
2. Iteratively shift the value leftwards into the registers
3. If any register is greater than 5, add 3 to it (this represents carry-over)

Such an algorithm is able to make use of the parallel computing capabilities of the FPGA to be carried out in one clock cycle.

100's	10's	1's	Binary	Operation
			1010 0010	162
		1	010 0010	<< #1
		10	10 0010	<< #2
		101	0 0010	<< #3
		1000		add 3
	1	0000	0010	<< #4
	10	0000	010	<< #5
	100	0000	10	<< #6
	1000	0001	0	<< #7
	1011			add 3
1	0110	0010		<< #8
1	6	2		

Figure 15 - Visualization of the conversion of a number to binary-coded decimal using the double-dabble algorithm (image source: N. McDonald, University of Utah)

Average Brightness Level

Many of these features (e.g. the automatic level adjuster) use the average frame luminance as an input. This value can be estimated at the frame read stage of the memory controller, by storing the cumulative sum of luminance of outgoing pixels in a register and dividing it by the number of pixels in a frame. A new frame signal latches the calculated value for the previous frame to the output and resets the counter. The key challenge in this module is performing the division operation needing to calculate the average, since the FPGA cannot divide by any value apart from powers of two (which is achieved through bit shifting). A frame as output by the ZBT memory contains 307,200 pixels, which is not a power of two and is hence impossible to divide by. To mitigate this constraint, the pixels are evenly sampled so that every 64 out of 75 pixels is added to the cumulative sum. The sum hence needs to be divided by 2^{18} to calculate the average, which is possible using bit shifts.

Reflection

Challenges/Difficulties

A critical issue that we had to deal with in the implementation of our system was timing of the system output. In the original design, we were using the input frame rate to clock the output frame rate. We also had a switchable buffer to avoid writing to the same set of pixels that we were reading from as was the case in the finalized version. The essential plan was to use the video input timing from the camera to trigger the buffer switch. For every input frame of 30 Hz NTSC video, we would output two frames of 60 Hz VGA video. However, we failed to realize that the NTSC standard *actually operates at a frame rate of 29.97 Hz*. This is a vestige of historical bandwidth constraints that makes this method of operation impossible because the output rate does not equal the input rate and therefore one cannot trigger the other without producing serious timing issues (trust us; we tried). If one does try to use the input as a trigger, the VGA timing specification can never actually be met because the horizontal line timing is irregular, with alternating lines pushing sync signals at different times. VGA monitors cannot cope with this sort of drastic timing variation, and thus our hopes of using a switchable frame buffer were dashed.

To solve this issue, we had to guarantee clocking independence between the input and output modules. We accomplished this by using a single shared frame buffer where reads and writes were scheduled and clocked without regard for one another. In this method, alternate fields are written willy-nilly to the frame buffer while sequential lines are read out, and strange screen tearing can occur at the output as a result with alternate lines shifted away from each other due to the interlaced NTSC input. We also had to use synchronization registers and data latches to interface between the output

clock domain and the system clock domain for reads from the frame buffer, because the output had to operate at 25 MHz, which is not an even multiple of the system 108 MHz.

Another battery of critical problems arose with configuring digital clock managers using the flaming hot pile of trash that is also known as Xilinx ISE 10.1. As previously mentioned, this software tends to forget itself and randomly save untraceable and unchangeable configuration data somewhere in the complicated chain of software transformations that must occur to configure the FPGA. This means that one cannot configure IP using the built-in IP configurator with any degree of confidence. We had to learn how to direct verilog to use Xilinx primitives to generate DCM clock modules, and even that process still caused errors that could only be fixed by rebooting the Linux machine. This software is complete shit. Avoid at all costs. Please.

Advice for Future Projects

We recommend the following advice for students wishing to pursue video/camera based projects in future iterations of 6.111:

- Simulations are incredibly useful for designing and debugging modules. It is almost always faster to write a testbench to visualize and debug a module than to try to debug a module when implemented in hardware. However, simulations make certain assumptions that make them an imperfect representation of the behavior of real physical systems. Many of these simplifications, such as assuming all combinational operations occur instantaneously or all clocks are synchronized, will cause systems to appear to work in simulation, only to fail when implemented. Hence, don't assume that a module will work first try just because it worked in simulation.
- Don't wait until just before the project deadline to begin integrating your system. While it may seem like the hardest part of writing Verilog is writing functioning modules, some of the more frustrating bugs come from issues in the interfaces between multiple modules. Additionally, it is possible that a module will behave entirely different when connected to a physical system than in the ideal world of simulations (see above). Hence, be sure to start this process early to allow time for debugging
- Use the Nexys 4 and Vivado for your product, instead of trying to work with legacy hardware and software (i.e. ISE and the old labkit). These tools will become ever more incompatible with modern computers as time goes on, making them even more frustrating to work with.
- Timing is important, especially when interfacing with external devices. If the timings of two modules are not synchronized correctly, they will likely not transfer information between themselves correctly. Creating a pipelining diagram is often a good way to resolve these errors.
- Piazza is a great way to get help, especially outside of staffed lab hours - use it.

Appendix - Verilog Modules

Note - The modules shown are for the system as demonstrated (i.e. with a 640x480 screen resolution and 3x3 convolution kernels). Modules were also written and tested in simulation for the intended specifications (1280x1024 screen resolution, 5x5 kernels), however these modules were not implemented into the final system. These modules are indicated with a * in their title.

```
//////////////////////////////////////////////////////////////////////////////
// Engineer:   jt.mcguire (80%), j.abel (20%)
//
// Module Name: labkit
// Description: top level module
//
//////////////////////////////////////////////////////////////////////////////

module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
ac97_synch,
              ac97_bit_clock,

              vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
              vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
              vga_out_vsync,

              tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
              tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
              tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

              tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
              tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
              tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
              tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

              ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
              ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

              ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
              ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

              clock_feedback_out, clock_feedback_in,

              flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
              flash_reset_b, flash_sts, flash_byte_b,

              rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

              mouse_clock, mouse_data, keyboard_clock, keyboard_data,

              clock_27mhz, clock1, clock2,
```

```
          disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
          disp_reset_b, disp_data_in,

          button0, button1, button2, button3, button_enter,
          button_right,
          button_left, button_down, button_up,

          switch,

          led,

          user1, user2, user3, user4,

          daughtercard,

          systemace_data, systemace_address, systemace_ce_b,
          systemace_we_b, systemace_oe_b, systemace_irq,
          systemace_mpbrdy,

          analyzer1_data, analyzer1_clock,
          analyzer2_data, analyzer2_clock,
          analyzer3_data, analyzer3_clock,
          analyzer4_data, analyzer4_clock);

   // Misc outputs
output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input  ac97_bit_clock, ac97_sdata_in;

   // VGA output
output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
     vga_out_hsync, vga_out_vsync;

   // Composite output
output [9:0] tv_out_ycrcb;
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
     tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
     tv_out_subcar_reset;

   // Composite input
input  [19:0] tv_in_ycrcb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
       tv_in, tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
       tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

   // RAM JAM
inout  [35:0] ram0_data;
output [18:0] ram0_address;
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b,
       ram0_we_b;
```

```
output [3:0] ram0_bwe_b;
inout  [35:0] ram1_data;
output [18:0] ram1_address;
output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b,
       ram1_we_b;
output [3:0] ram1_bwe_b;

   // RAM clock feedback
input  clock_feedback_in;
output clock_feedback_out;
   //assign clock_feedback_out=1'b0;

   // Flash crap
inout  [15:0] flash_data;
output [23:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
input  flash_sts;

   // Misc outputs
output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;
input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   // Clocks
input  clock_27mhz, clock1, clock2;

   // System display
output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
input  disp_data_in;
output  disp_data_out;

   // Buttons and LEDs
input  button0, button1, button2, button3, button_enter, button_right,
     button_left, button_down, button_up;
input  [7:0] switch;
output [7:0] led;

   // Data ports
inout [31:0] user1, user2, user3, user4;
inout [43:0] daughtercard;
inout  [15:0] systemace_data;
output [6:0]  systemace_address;
output systemace_ce_b, systemace_we_b, systemace_oe_b;
input  systemace_irq, systemace_mpbrdy;
output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
              analyzer4_data;
output analyzer1_clock, analyzer2_clock, analyzer3_clock,
       analyzer4_clock;

//////////////////////////
//// IO Assignments /////
//////////////////////////
```

```
    // Audio Input and Output
    assign beep= 1'b0;
    assign audio_reset_b = 1'b0;
    assign ac97_synch = 1'b0;
    assign ac97_sdata_out = 1'b0;
    // ac97_sdata_in is an input

    // VGA Output
//    assign vga_out_red = 8'h0;
//    assign vga_out_green = 8'h0;
//    assign vga_out_blue = 8'h0;
//    assign vga_out_sync_b = 1'b1;
//    assign vga_out_blank_b = 1'b1;
//    assign vga_out_pixel_clock = 1'b0;
//    assign vga_out_hsync = 1'b0;
//    assign vga_out_vsync = 1'b0;

    // Video Output
    assign tv_out_ycrcb = 10'h0;
    assign tv_out_reset_b = 1'b0;
    assign tv_out_clock = 1'b0;
    assign tv_out_i2c_clock = 1'b0;
    assign tv_out_i2c_data = 1'b0;
    assign tv_out_pal_ntsc = 1'b0;
    assign tv_out_hsync_b = 1'b1;
    assign tv_out_vsync_b = 1'b1;
    assign tv_out_blank_b = 1'b1;
    assign tv_out_subcar_reset = 1'b0;

    // SRAM unused
    assign ram1_data = 36'hZ;
    assign ram1_address = 19'h0;
    assign ram1_adv_ld = 1'b0;
    assign ram1_cen_b = 1'b1;
    assign ram1_ce_b = 1'b1;
    assign ram1_oe_b = 1'b1;
    assign ram1_we_b = 1'b1;
    assign ram1_bwe_b = 4'hF;
    // clock_feedback_in is an input

    // Flash ROM
    assign flash_data = 16'hZ;
    assign flash_address = 24'h0;
    assign flash_ce_b = 1'b1;
    assign flash_oe_b = 1'b1;
    assign flash_we_b = 1'b1;
    assign flash_reset_b = 1'b0;
    assign flash_byte_b = 1'b1;
    // flash_sts is an input

    // RS-232 Interface
```

```
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

// LED Displays
assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0;
// disp_data_in is an input

// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;
// systemace_irq and systemace_mpbrdy are inputs

// Logic Analyzer
assign analyzer1_clock = 1'b1;
assign analyzer3_clock = 1'b1;

   reg [8:0] rstcnt;

   wire sysclk;
   wire locked;
   wire rst;
   wire ram_clk;
   wire out_clk;

   assign rst = ~button0;
   assign ram1_clk = 1'b0;

   ////////// CLOCK MANAGERS ////////////////
   //////////////////////////////////////////

   wire refclk;
```

```
    IBUFG ref_buf1 (.I(clock_27mhz), .O(refclk));

    // Main sysclk generator
    wire sysclk_buf;
    BUFG CLKFX_BUFG_main (.I(sysclk_buf),
                          .O(sysclk));
    wire clkfb_main, clk0_main, ramclkout;
    BUFG CLK0_BUFG_main (.I(clk0_main),
                         .O(clkfb_main));
    DCM DCM_main (.CLKFB(clkfb_main),
                .CLKIN(refclk),
                .DSSEN(1'b0),
                .PSCLK(1'b0),
                .PSEN(1'b0),
                .PSINCDEC(1'b0),
                .RST(),
                .CLKDV(),
                .CLKFX(sysclk_buf),
                .CLKFX180(),
                .CLK0(clk0_main),
                .CLK2X(),
                .CLK2X180(),
                .CLK90(),
                .CLK180(),
                .CLK270(),
                .LOCKED(),
                .PSDONE(),
                .STATUS());
defparam DCM_main.CLK_FEEDBACK = "1X";
defparam DCM_main.CLKDV_DIVIDE = 2.0;
defparam DCM_main.CLKFX_DIVIDE = 1;
defparam DCM_main.CLKFX_MULTIPLY = 4;
defparam DCM_main.CLKIN_DIVIDE_BY_2 = "FALSE";
defparam DCM_main.CLKIN_PERIOD = 37.037;
defparam DCM_main.CLKOUT_PHASE_SHIFT = "NONE";
defparam DCM_main.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
defparam DCM_main.DFS_FREQUENCY_MODE = "LOW";
defparam DCM_main.DLL_FREQUENCY_MODE = "LOW";
defparam DCM_main.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM_main.FACTORY_JF = 16'hC080;
defparam DCM_main.PHASE_SHIFT = 0;
defparam DCM_main.STARTUP_WAIT = "FALSE";

//assign ram0_clk = sysclk;
// Main ramclk generator
wire clkfb_ram, clk0_ram, ramram, fb_out;
BUFG CLK0_BUFG_ram (.I(clk0_ram), .O(fb_out));
IBUFG RAM_FB (.I(clock_feedback_in), .O(clkfb_ram));
BUFG del1 (.I(fb_out), .O(clock_feedback_out));
assign ram0_clk = fb_out;
DCM DCM_ram (.CLKFB(clkfb_ram),
             .CLKIN(sysclk),
```

```
                .DSSEN(1'b0),
                .PSCLK(1'b0),
                .PSEN(1'b0),
                .PSINCDEC(1'b0),
                .RST(rstcnt<=9'b111111100),
                .CLKDV(),
                .CLKFX(),
                .CLKFX180(),
                .CLK0(clk0_ram),
                .CLK2X(),
                .CLK2X180(),
                .CLK90(),
                .CLK180(),
                .CLK270(),
                .LOCKED(),
                .PSDONE(),
                .STATUS());
defparam DCM_ram.CLK_FEEDBACK = "1X";
defparam DCM_ram.CLKDV_DIVIDE = 2.0;
defparam DCM_ram.CLKFX_DIVIDE = 1;
defparam DCM_ram.CLKFX_MULTIPLY = 4;
defparam DCM_ram.CLKIN_DIVIDE_BY_2 = "FALSE";
defparam DCM_ram.CLKIN_PERIOD = 37.037;
defparam DCM_ram.CLKOUT_PHASE_SHIFT = "NONE";
defparam DCM_ram.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
defparam DCM_ram.DFS_FREQUENCY_MODE = "LOW";
defparam DCM_ram.DLL_FREQUENCY_MODE = "LOW";
defparam DCM_ram.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM_ram.FACTORY_JF = 16'hC080;
defparam DCM_ram.PHASE_SHIFT = 0;
defparam DCM_ram.STARTUP_WAIT = "FALSE";

// Main out clk generator
wire outclk_buf;
BUFG CLKFX_BUFG_out (.I(outclk_buf),
                     .O(out_clk));
wire clkfb_out, clk0_out;
BUFG CLK0_BUFG_out (.I(clk0_out),
                    .O(clkfb_out));
DCM DCM_out (.CLKFB(clkfb_out),
             .CLKIN(refclk),
             .DSSEN(1'b0),
             .PSCLK(1'b0),
             .PSEN(1'b0),
             .PSINCDEC(1'b0),
             .RST(),
             .CLKDV(),
             .CLKFX(outclk_buf),
             .CLKFX180(),
             .CLK0(clk0_out),
             .CLK2X(),
             .CLK2X180(),
```

```
            .CLK90(),
            .CLK180(),
            .CLK270(),
            .LOCKED(),
            .PSDONE(),
            .STATUS());
defparam DCM_out.CLK_FEEDBACK = "1X";
defparam DCM_out.CLKDV_DIVIDE = 2.0;
defparam DCM_out.CLKFX_DIVIDE = 27;
defparam DCM_out.CLKFX_MULTIPLY = 25;
defparam DCM_out.CLKIN_DIVIDE_BY_2 = "FALSE";
defparam DCM_out.CLKIN_PERIOD = 37.037;
defparam DCM_out.CLKOUT_PHASE_SHIFT = "NONE";
defparam DCM_out.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
defparam DCM_out.DFS_FREQUENCY_MODE = "LOW";
defparam DCM_out.DLL_FREQUENCY_MODE = "LOW";
defparam DCM_out.DUTY_CYCLE_CORRECTION = "TRUE";
defparam DCM_out.FACTORY_JF = 16'hC080;
defparam DCM_out.PHASE_SHIFT = 0;
defparam DCM_out.STARTUP_WAIT = "FALSE";

   ////////// END CLOCK MANAGERS ////////////////////////

   assign tv_in_clock = refclk;

   wire out_done;
   wire in_ready, out_ready;
   wire in_done;
   wire new_out;
   wire nf, f;
   reg f0;
   reg [21:0] fcnt;
   reg [20:0] fcnt2;
   wire [8:0] in_x, in_y, out_x, read_x, read_y;
   wire [35:0] word, data_in;
   wire tv_clk;
   wire highZ;

   ram_manager2 deconflict (.clk(sysclk), .rst(rst),
                            .out_ready(out_ready), .read_x(read_x),
                            .read_y(read_y), .out_x(out_x),
                            .new_out(new_out), .in_ready(in_ready),
                            .in_done(in_done), .in_x(in_x), .in_y(in_y),
                            .data_in(data_in), .addr(ram0_address),
                            .word(word), .data(ram0_data),
                            .adv(ram0_adv_ld), .clk_en(ram0_cen_b),
                            .chip_en(ram0_ce_b), .write_en(ram0_we_b),
                            .bwrite_en(ram0_bwe_b), .highZ(highZ));

   assign ram0_oe_b=1'b0;

   // Video Input
```

```
assign tv_in_fifo_read = 1'b1;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b1;
assign tv_in_reset_b = ~rst;
// tv_in_ycrcb, tv_in_data_valid, nalyzer4_clock = tv_in_clock
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   wire write_tv_prom;

   // Half speed ADV7185 output clock buffer
   IBUFG buf_half ( .O(tv_clk), .I(tv_in_line_clock2) );

   assign led = {refclk, out_clk, f, rst, ram0_clk, tv_in_i2c_clock,
                 tv_in_i2c_data,  sysclk}; //tv_in_ycrcb[9:2];

   wire ir1,ir0;
   wire [8:0] in_x_c, in_y_c, in_x_cam, in_y_cam;
   wire [35:0] data_in_c, data_in_cam;

   // Instantiate the module
   tvInControl2 tv2 ( .in_clk(tv_clk), .clk(sysclk), .rst(rst),
                      .lum_in(tv_in_ycrcb[19:12]),
                      .chrom_in(tv_in_ycrcb[9:2]), .in_ready(ir0),
                     .in_done(in_done), .in_x(in_x_cam),
                     .in_y(in_y_cam), .data_in(data_in_cam), .ovf(ovf),
                     .newFrame(f) );

   assign analyzer2_clock = sysclk;

   colorbar cbar1 (.clk(sysclk), .rst(rst), .in_ready(ir1),
                   .in_x(in_x_c), .in_y(in_y_c), .data_in(data_in_c) );

   assign in_x = ~button1 ? in_x_c : in_x_cam;
   assign in_y = ~button1 ? in_y_c : in_y_cam;
   assign data_in = ~button1 ? data_in_c : data_in_cam;
   assign in_ready = button3 & (~button1 ? ir1 : ir0);

   assign analyzer4_clock = tv_in_clock;

   wire ack;
   wire sending;
   wire [3:0] tv_addr;
   wire [3:0] rom_addr;
   wire [7:0] tv_dout;

   assign write_tv_prom = (~sending && (~button_left | ~button_right));

   TV_i2c tv_prg ( .rst(rst | write_tv_prom), .clk(sysclk),
                   .i2c_data(tv_in_i2c_data),
                   .i2c_clock(tv_in_i2c_clock), .isAcked(ack),
                   .sending(sending), .byteAddr(tv_addr), .dout(tv_dout)
                   );
```

```
// Left button programs saturation and right button programs contrast
// from switch[7:0] value
assign rom_addr = ~button_left ?  8'd5 : ( ~button_right ? 8'd8 :
                                          8'd15 ) ;

initRAM tv_prg_prom( .clka(sysclk), .clkb(sysclk), .dina(switch),
                  .addra(rom_addr), .wea(write_tv_prom),
                   .addrb(tv_addr),  .doutb(tv_dout));

wire [7:0] lum;
wire [7:0] lum0,lum1,lum2,lum3,lum4,lum5,lum6,lum7,lum8;
reg [7:0] Cr, Cb;
wire [7:0] cr, cb;
reg [10:0] x,y;
wire [10:0] xx,yy;
reg h_sync, v_sync, blank;
wire hs,vs,bl;
wire hbl, vbl;
wire [7:0] vout;

read_ram2 final_out (.clk(out_clk), .sysclk(sysclk), .rst(rst),
                     .new_out(new_out), .rx(read_x), .ry(read_y),
                     .out_ready(out_ready), .inx(out_x), .word(word),
                     .h_sync(hs), .v_sync(vs), .blank(bl), .Cr(cr),
                     .Cb(cb), .lum0(lum0), .lum1(lum1), .lum2(lum2),
                     .lum3(lum3), .lum4(lum4), .lum5(lum5),
                     .lum6(lum6), .lum7(lum7), .lum8(lum8),
                     .out_x(xx), .out_y(yy));

assign analyzer4_data = {8'b0, refclk, out_clk, ram0_clk, h_sync,
                         v_sync, out_ready, new_out, 1'b0};
assign analyzer1_data = {8'b0, word[17:10]};
assign analyzer2_data = {8'b0, lum4};
assign analyzer3_data = {8'b0, lum0};

reg [7:0] lum_real;
wire [7:0] lum_edge;

reg [7:0] l0,l1,l2,l3,l4,l5,l6,l7,l8;

always @(posedge refclk)begin
      rstcnt<= rst ? 9'b0 : (rstcnt>=9'b111111111) ? 9'b111111111 :
               rstcnt+1;
end

always @(posedge out_clk) begin
      f0<=f;
      // Switch the buffer select on a frame rising edge
      h_sync<=~hs;
      v_sync<=~vs;
      blank<=~bl;
```

```
            l0<=lum0;
            l1<=lum1;
            l2<=lum2;
            l3<=lum3;
            l4<=lum4;
            l5<=lum5;
            l6<=lum6;
            l7<=lum7;
            l8<=lum8;
            x<=xx;
            y<=yy;
            Cr<=cr;
            Cb<=cb;
            lum_real <= ~button_enter ? lum : ~button_up ? lum_edge : l4;
      end

      // Choose a new frame output at 60 Hz (both edges of frame signal)
      assign nf = f & ~f0;

      wire [23:0] rgb_out;

      sharpen_3 sharp1 (.clk_in(out_clk), .lum0(l0),.lum1(l1), .lum2(l2),
                        .lum3(l3), .lum4(l4), .lum5(l5), .lum6(l6),
                        .lum7(l7), .lum8(l8), .y_out(lum) );

      edge_filt edge1 (.clk_in(out_clk),
                       .y_in({l0,l1,l2,l3,l4,l5,l6,l7,l8}),
                       .y_out(lum_edge) );

      ycbcr_2_rgb tl (.clk_in(out_clk), .y_in(lum_real), .cb_in(Cb),
                      .cr_in(Cr), .rgb_out(rgb_out));

      assign vga_out_blue = rgb_out[7:0];
      assign vga_out_red = rgb_out[23:16];
      assign vga_out_green = rgb_out[15:8];
      assign vga_out_hsync = h_sync;
      assign vga_out_vsync = v_sync;
      assign vga_out_sync_b = 1'b1;
      assign vga_out_blank_b = blank;
      assign vga_out_pixel_clock = out_clk;
      wire [23:0] rgb;

endmodule

//////////////////////////////////////////////////////////////////////////////
// Engineer:   j.abel
//
// Module Name: average_finder
// Description: Estimates the average luminance of a frame
//
//////////////////////////////////////////////////////////////////////////////
```

```
module average_finder(
      input clk_in,                  // VGA clock
      input rst_in,                  // global reset
      input new_frame,               // signal indicating start of frame
      input pixel_active,            // indicates whether pixel in frame or
blanking interval
      input [7:0] data_in,           //pixel y_value
      output reg [7:0] y_av          //average value
      );

      reg [22:0] y_sum = 0;          //cumulative sum
      reg [6:0] sample_clock = 0;
      reg sample_active;
      reg frame_reset;

      always @ (*) begin
      // Sampling - pixel value is not added to y_sum at regularly spaced
      // intervals, such that 64 samples are taken every 75 pixels
            sample_active = ((sample_clock == 7'd3) ||
                  (sample_clock == 7'd10) ||
                  (sample_clock == 7'd17) ||
                  (sample_clock == 7'd24) ||
                  (sample_clock == 7'd31) ||
                  (sample_clock == 7'd37) ||
                  (sample_clock == 7'd44) ||
                  (sample_clock == 7'd51) ||
                  (sample_clock == 7'd58) ||
                  (sample_clock == 7'd65) ||
                  (sample_clock == 7'd72)) ?
                  0 : 1;
      end

      always @ (posedge clk_in) begin
      // reset
         frame_reset <= new_frame;
            if (rst_in) begin
                  y_sum <= 0;
                  sample_clock <= 0;
                  y_av <= 0;
            end else begin
                  sample_clock <= (sample_clock > 7'd74) ? 7'd0 :
                                  (pixel_active) ? sample_clock + 1:
                                  sample_clock;
                  y_sum <= (frame_reset) ? 0: (^data_in === 1'bX) ? 0:
                           (sample_active && pixel_active) ? y_sum +
                          data_in[7:4] : y_sum;
                  y_av <= (frame_reset) ? y_sum >> 14: y_av;
            end
      end

endmodule
```

```
//////////////////////////////////////////////////////////////////////////////
// Engineer:   j.abel
//
// Module Name: avg_value
// Description: assign character addresses for frame average brightness
//              display in overlay
//////////////////////////////////////////////////////////////////////////////

module avg_value(
    input clk_in,
    input [7:0] y_avg,
    output reg [4:0] char_tens,      //address of characters
    output reg [4:0] char_ones
    );

    //character addresses
    parameter ZERO = 15;
    parameter ONE = 8;
    parameter TWO = 14;
    parameter THREE = 13;
    parameter FOUR = 4;
    parameter FIVE = 3;
    parameter SIX = 11;
    parameter SEVEN = 10;
    parameter EIGHT = 2;
    parameter NINE = 7;

    //reg to convert average to a percentage
    reg [14:0] percent_big;
    reg [7:0]  percent;
    //unclipped digits of percentage
    wire [3:0] hundreds, tens_raw, ones_raw;
    //clipped digits of percentage
    reg [3:0] tens, ones;

    //converts percentage to a binary-coded decimal
    num2bcd conv (percent, hundreds, tens_raw, ones_raw);

    always @ (*) begin
        //percentage calculation (of 255 - max brightness of a pixel)
        percent = percent_big >> 8;
        //output clipping
        tens = (hundreds > 0) ? 4'd9 : tens_raw;
        ones = (hundreds > 0) ? 4'd9 : ones_raw;
        //character assignment
        case (tens)
            0: char_tens = ZERO;
            1: char_tens = ONE;
            2: char_tens = TWO;
            3: char_tens = THREE;
            4: char_tens = FOUR;
            5: char_tens = FIVE;
```

```
            6: char_tens = SIX;
            7: char_tens = SEVEN;
            8: char_tens = EIGHT;
            9: char_tens = NINE;
        endcase
        case (ones)
            0: char_ones = ZERO;
            1: char_ones = ONE;
            2: char_ones = TWO;
            3: char_ones = THREE;
            4: char_ones = FOUR;
            5: char_ones = FIVE;
            6: char_ones = SIX;
            7: char_ones = SEVEN;
            8: char_ones = EIGHT;
            9: char_ones = NINE;
        endcase
    end

    always @ (posedge clk_in) begin
        //allows percentage to be converted with discrete arithmetic
        percent_big <= y_avg * 100;
    end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j. abel
//
// Module Name: num2bcd
// Description: converts a number to a binary-coded decimal - submodule of
//              avg_value
//////////////////////////////////////////////////////////////////////////////////



module num2bcd(
    input [7:0] percent,
    output reg [3:0] hundreds,
    output reg [3:0] tens_raw,
    output reg [3:0] ones_raw
    );

    integer i;
    always @ (*) begin
        hundreds = 4'd0;
        tens_raw = 4'd0;
        ones_raw = 4'd0;
        //iterative algorithm to convert to BCD (shift, add 3)
        for (i=7; i>=0; i = i-1) begin
            if (tens_raw >= 5)
                tens_raw = tens_raw + 3;
```

```
            if (ones_raw >= 5)
                ones_raw = ones_raw + 3;

            hundreds[0] = tens_raw[3];
            tens_raw = tens_raw << 1;
            tens_raw[0] = ones_raw[3];
            ones_raw = ones_raw << 1;
            ones_raw[0] = percent[i];
        end
    end

endmodule

//////////////////////////////////////////////////////////////////////////////
// Engineer:   jt. mcguire
//
// Module Name: colorbar
// Description: generates colorbars for testing
//
//////////////////////////////////////////////////////////////////////////////

module colorbar(clk, rst, in_ready, in_x, in_y, data_in );

       input clk, rst;
       output reg in_ready;
       output [8:0] in_x,in_y;
       output reg [35:0] data_in;

       //pixel location
       reg [8:0] hcount;
       reg [8:0] vcount;

       //replicates camera timing
       reg [2:0] cnt;

       assign in_x = hcount;
       assign in_y = vcount;

       always @(posedge clk) begin
            if(rst)begin
                  //reset
                  hcount<=9'b0;
                  vcount<=9'b0;
                  data_in<=36'b0;
                  cnt<=3'b0;
                  in_ready<=1'b0;
            end else begin
                  cnt<=cnt+1;
                  //color bar generation
                  if(cnt==3'b0)begin
                        in_ready<=1'b1;
                        if(hcount<70)begin
```

```
                              hcount<=hcount+1;
                              data_in<={8'hff, 10'b0, 8'hff, 10'b0};
                              vcount<=vcount;
                        end else if(hcount<130)begin
                              hcount<=hcount+1;
                              data_in<={8'h5f, 10'haa, 8'h5f, 10'haa};
                              vcount<=vcount;
                        end else if(hcount<200)begin
                              hcount<=hcount+1;
                              data_in<={8'h2f, 10'h36, 8'h2f, 10'h36};
                              vcount<=vcount;
                        end else if(hcount<311)begin
                              hcount<=hcount+1;
                              vcount<=vcount;
                              data_in<={8'h01, 10'h01, 8'h01, 10'h01};
                        end else begin
                              hcount<=9'b0;
                              data_in<=data_in;
                              if(vcount>=479)begin
                                    vcount<=9'b0;
                              end else begin
                                    vcount<=vcount+1;
                              end
                        end
                  end else begin
                        //holds values otherwise
                        hcount<=hcount;
                        data_in<=data_in;
                        vcount<=vcount;
                        in_ready<=1'b0;
                  end
            end
      end


endmodule


//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j. abel
//
// Module Name: convolution
// Description: Applies convolution filters to kernels of pixels
//
//////////////////////////////////////////////////////////////////////////////////

module convolution(
    input clk_in,
    input [71:0] y_kern_in,    //3x3 kernel of adjacent 8 bit pixels
                               //Size changed to [199:0] for a 5x5 kernel
    input [7:0] y_avg,         //Average value
    input [2:0] conv_state,
```

```
    input [2:0] lvl_state,     //Filter states from FSM
    input is_edge,             //Detects whether pixel on edge of frame
    output reg [7:0] y_out     //Single pixel out
    );

    parameter SOBEL = 3'd1;
    parameter NOISE = 3'd2;
    parameter SHARP = 3'd3;
    parameter SUN   = 3'd4;

    // Filter Outputs
    wire [7:0] y_sobel, y_med, y_sharp, y_sun;

    //For edge dilation
    wire current_edge;
    reg old_edge;

    //Filter module calls
    edge_filt edge1(.clk_in(clk_in), .y_in(y_kern_in), .y_avg(y_avg),
                    .old_edge(old_edge), .y_out(y_sobel),
                    .current_edge(current_edge));
    median_filt med1(.clk_in(clk_in), .y_in(y_kern_in), .y_out(y_med));
    sharpen_filt sharp1(.clk_in(clk_in), .y_in(y_kern_in),
                        .y_out(y_sharp));
    sun_filt sun1(.clk_in(clk_in), .y_in(y_kern_in), .lvl_state(lvl_state),
                  .y_avg(y_avg), .y_out(y_sun));

    always @ (*) begin
        //filter selection
        y_out = (is_edge) ? y_kern_in[29:24]:
            (conv_state == SOBEL && ~is_edge) ? y_sobel:
            (conv_state == NOISE && ~is_edge) ? y_med:
            (conv_state == SHARP && ~is_edge) ? y_sharp:
            (conv_state == SUN && ~is_edge)   ? y_sun:
            y_kern_in[29:24];
    end

    always @ (posedge clk_in) begin
       old_edge <= current_edge;
    end

   endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   course staff
//
// Module Name: debounce
// Description: debounces incoming button signals
//
//////////////////////////////////////////////////////////////////////////////////

module debounce(
```

```
    input clk_in,
    input noisy_in,
    output wire clean_out
    );

    reg [19:0] count;
    reg output_reg;
    reg old;

    assign clean_out = output_reg;

    always @ (posedge clk_in) begin
        if(count == 16'd50000) begin
            output_reg <= old;
            count <= 20'b0;
        end else if(noisy_in == old) begin
            count <= count + 1;
        end else begin
            count <= 20'b0;
            old <= noisy_in;
        end
    end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j. abel
//
// Module Name: edge_filt
// Description: Sobel edge detection filter
//
//////////////////////////////////////////////////////////////////////////////////

module edge_filt(
    input clk_in,
    input [71:0] y_in,         //3x3 pixel kernel
    input [5:0] y_avg,         //average frame brightness
    input old_edge,            //indicates whether previous pixel was a
                               //Sobel edge
    output reg [5:0] y_out,
    output reg current_edge   //indicates that current pixel is an edge
    );

    parameter THRESHOLD = 8'd160;          //threshold for edge detection
    parameter EDGE_COLOR_TSH = 8'd160;     //threshold frame brightness to
                                           //determine edge display color
    parameter BLACK = 8'd16;
    parameter WHITE = 8'd235;

    //Sums of positive and negative values in Sobel kernel
    wire [9:0] g_x_pos;
    wire [9:0] g_x_neg;
```

```
    wire [9:0] g_y_pos;
    wire [9:0] g_y_neg;
    reg [7:0] edge_color;

    //Sobel convolution - arithmetic
    assign g_x_pos = y_in[55:48] + y_in[7:0] + (y_in[31:24]<<1);
    assign g_x_neg = y_in[71:64] + y_in[23:16] + (y_in[47:40]<<1);
    assign g_y_pos = y_in[23:16] + y_in[7:0] + (y_in[15:8]<<1);
    assign g_y_neg = y_in[71:64] + y_in[55:48] + (y_in[63:56]<<1);

    //Edge color assignment
    always @ (*) begin
        edge_color = (y_avg > EDGE_COLOR_TSH) ? BLACK : WHITE;
    end

    always @ (posedge clk_in) begin
       //edge detection from sums
        current_edge <= (((g_x_pos >= g_x_neg) &&
                         ((g_x_pos-g_x_neg) >= THRESHOLD)) ||
                         ((g_x_neg >= g_x_pos) &&
                         ((g_x_neg-g_x_pos) >= THRESHOLD)) ||
                         ((g_y_pos >= g_y_neg) &&
                         ((g_y_pos-g_y_neg) >= THRESHOLD)) ||
                         ((g_y_neg >= g_y_pos) &&
                         ((g_y_neg-g_y_pos) >= THRESHOLD))) ?
                         1 : 0;
        //output assignment
        y_out <= (current_edge || old_edge) ? edge_color : y_in[29:24];
    end

endmodule

//////////////////////////////////////////////////////////////////////////////
// Engineer:   j. abel
//
// Module Name: filt_sel
// Description: FSM for selecting image processing filters
//
//////////////////////////////////////////////////////////////////////////////

module filt_sel(
    input clk_in,
    input l_btn,
    input r_btn,
    input u_btn,
    input d_btn,
    input reset,
    output reg [2:0] adj_state,            //controls adjustment filter
    output reg [2:0] conv_state,           //controls convolution filters
    output reg overlay                     //activates overlay
    );
```

```
    // Button rising edge detection
    reg l_old; reg r_old; reg u_old; reg d_old;
    wire l_edge; wire r_edge; wire u_edge; wire d_edge;

    assign l_edge = l_btn & !l_old;
    assign r_edge = r_btn & !r_old;
    assign u_edge = u_btn & !u_old;
    assign d_edge = d_btn & !d_old;

    always @ (posedge clk_in) begin
    // Reset
    if (reset) begin;
        adj_state <= 3'd0;
        conv_state <= 3'd0;
        overlay <= 0; end
    // Filter selection FSM
    else begin
        adj_state <= (l_edge) ? adj_state - 1: (r_edge) ? adj_state + 1:
                     adj_state;
        conv_state <= (u_edge) ? conv_state + 1: (conv_state == 3'd5) ?
                      3'd0 : conv_state;
        overlay <= (d_edge) ? ~overlay: overlay; end
    l_old <= l_btn;
    r_old <= r_btn;
    u_old <= u_btn;
    d_old <= d_btn;
    end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j. abel
//
// Module Name: font_reader
// Description: reads font character from BRAM
//
//////////////////////////////////////////////////////////////////////////////////

module font_reader(
    input clk_in,
    input [10:0] hcount,       //pixel location
    input [9:0] vcount,
    input [10:0] x_loc,        //location of corner of digit
    input [9:0] y_loc,
    input [4:0] char_in,       //shortened address of character
    output wire is_text        //indicates whether current pixel is text
    );

    //size parameters
    parameter HEIGHT = 36;
    parameter WIDTH = 32;
```

```
    //actual memory address
    reg[14:0] address;
    reg in_box;
    wire pixel_out;

    //BRAM reader
    font_bram font(.clka(clk_in), .addra(address), .douta(pixel_out));
    //prevents erroneous pixel assignment outside a box defining a digit
    assign is_text = (in_box) ? pixel_out : 0;

    always @ (*) begin
        //determines whether current pixel in digit box
        in_box = ((hcount >= x_loc && hcount < (x_loc+WIDTH)) &&
                  (vcount >= y_loc && vcount < (y_loc+HEIGHT))) ?
                  1 : 0;
        //converts shortened address to actual address
        address = (in_box) ? ((hcount-x_loc) + (vcount-y_loc)*WIDTH)
                  + (HEIGHT*WIDTH*char_in): 0;
    end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j.abel
//
// Module Name: level_adjustment
// Description: pixel level adjustment using linearized gamma correction
//
//////////////////////////////////////////////////////////////////////////////////

module level_adjustment(
    input clk_in,
    input [7:0] y_in,
    input [2:0] adj_mode,
    input [7:0] y_avg,
    output reg [7:0] y_out
    );

    reg [7:0] y_reg;
    wire [10:0] y_mult; //prevents overflow when multiplying
    reg [1:0] auto_mode; //mode of automatic level adjustment

    // brightness levels
    parameter BRIGHT3 = 3'd3;
    parameter BRIGHT2 = 3'd2;
    parameter BRIGHT1 = 3'd1;
    parameter ZERO = 3'd0;
    parameter DARK1 = 3'd7;
    parameter DARK2 = 3'd6;
    parameter DARK3 = 3'd5;
    parameter AUTO = 3'd4;
```

```
// automatic levels and thresholds
parameter A_DARK = 2'b01;
parameter A_OK = 2'b00;
parameter A_LIGHT = 2'b10;
parameter THR_DARK = 8'd72;
parameter THR_LIGHT = 8'd168;

assign y_mult = y_in;

always @ (*) begin
    // automatic mode assignment
    auto_mode = (y_avg > THR_LIGHT) ? A_DARK :
                (y_avg < THR_DARK) ? A_LIGHT : A_OK;
end

always @ (posedge clk_in) begin
    // automatic level adjustment
    if (adj_mode == AUTO) begin
        case (auto_mode)
            A_LIGHT: begin
                     y_out <= (y_in >= 8'd191) ? 8'd235 :
                              ((y_mult*5)>>2)-4;  end
            A_DARK: begin
                    y_out <= (y_in < 8'd61) ? 8'd16 :
                             ((y_mult*5)>>2)-59; end
            default: y_out <= y_in;
       endcase
    end else begin
        // manual level adjustment - based on linear remapping
        case (adj_mode)
            BRIGHT3: begin
                y_out <= (y_in >= 8'd125) ? 8'd235 : (y_mult*2)-16;
                end
            BRIGHT2: begin
                y_out <= (y_in >= 8'd162) ? 8'd235 : ((y_mult*3)>>1)-8;
                end
            BRIGHT1: begin
                y_out <= (y_in >= 8'd191) ? 8'd235 : ((y_mult*5)>>2)-4;
                end
            DARK1: begin
                y_out <= (y_in < 8'd61) ? 8'd16 : ((y_mult*5)>>2)-59;
                end
            DARK2: begin
                y_out <= (y_in < 8'd90) ? 8'd16 : ((y_mult*3)>>1)-118;
                end
            DARK3: begin
                y_out <= (y_in < 8'd126) ? 8'd16 : (y_mult*2)-235;
                end
            default: y_out <= y_in;
        endcase
    end
 end
```

```
endmodule


//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j. abel
//
// Module Name: median_filt
// Description: Finds median value of 3x3 convolution kernel
//
//////////////////////////////////////////////////////////////////////////////////

module median_filt(
    input clk_in,
    input [71:0] y_in,
    output reg [5:0] y_out
    );

    //pixels
    wire[7:0] p_1 = y_in[71:64];
    wire[7:0] p_2 = y_in[63:56];
    wire[7:0] p_3 = y_in[55:48];
    wire[7:0] p_4 = y_in[47:40];
    wire[7:0] p_5 = y_in[39:32];
    wire[7:0] p_6 = y_in[31:24];
    wire[7:0] p_7 = y_in[23:16];
    wire[7:0] p_8 = y_in[15:8];
    wire[7:0] p_9 = y_in[7:0];

    //intermediate stage outputs in sorting matrix
    wire[7:0] q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9;
    wire[7:0] r_1, r_2, r_3;

    //junk outputs (needed for correct number of submodule arguments)
    wire[7:0] junk_1, junk_2 ,junk_3, junk_4,
              junk_5, junk_6, junk_7, junk_8;

    //submodule calls - stage 1
    three_median p_a (.clk_in(clk_in), .a_1(p_1), .a_2(p_2), .a_3(p_3),
                      .min(q_7), .med(q_4), .max(q_1));
    three_median p_b (.clk_in(clk_in), .a_1(p_4), .a_2(p_5), .a_3(p_6),
                      .min(q_8), .med(q_5), .max(q_2));
    three_median p_c (.clk_in(clk_in), .a_1(p_7), .a_2(p_8), .a_3(p_9),
                      .min(q_9), .med(q_6), .max(q_3));
    //submodule calls - stage 2
    three_median q_a (.clk_in(clk_in), .a_1(q_1), .a_2(q_2), .a_3(q_3),
                      .min(r_1), .med(junk_1), .max(junk_2));
    three_median q_b (.clk_in(clk_in), .a_1(q_4), .a_2(q_5), .a_3(q_6),
                      .min(junk_3), .med(r_2), .max(junk_4));
    three_median q_c (.clk_in(clk_in), .a_1(q_7), .a_2(q_8), .a_3(q_9),
                      .min(junk_5), .med(junk_6), .max(r_3));
    //submodule calls - stage 3
```

```
    three_median fin (.clk_in(clk_in), .a_1(r_1), .a_2(r_2), .a_3(r_3),
                      .min(junk_7), .med(y_out), .max(junk_8));
    end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j. abel
//
// Module Name: three_median
// Description: Finds min, med, and max of 3 values - submodule of
//              median_filt
//////////////////////////////////////////////////////////////////////////////////

module three_median(
    Input clk_in,
    input [7:0] a_1,
    input [7:0] a_2,
    input [7:0] a_3,
    output reg [7:0] min,
    output reg [7:0] med,
    output reg [7:0] max
    );

    always @ (posedge clk_in) begin
        max <= ((a_3 >= a_1) && (a_3 >= a_2)) ? a_3 :
        ((a_2 >= a_1) && (a_2 >= a_3)) ? a_2 : a_1;
        min <= ((a_1 <= a_3) && (a_1 <= a_2)) ? a_1 :
        ((a_2 <= a_1) && (a_2 <= a_3)) ? a_2 : a_3;
        med <= (((a_1 == max) && (a_2 == min)) ||
        ((a_2 == max) && (a_1 == min))) ? a_3 :
        (((a_2 == max) && (a_3 == min)) ||
        ((a_3 == max) && (a_2 == min))) ? a_1 : a_2;
    end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j. abel
//
// Module Name: overlay
// Description: controls display of text-based overlay
//
//////////////////////////////////////////////////////////////////////////////////

module overlay(
    input clk_in,
    input [9:0] hcount,        //horizontal pixel position
    input [9:0] vcount,        //vertical pixel position
    input [2:0] adj_state,     //filter states
    input [2:0] conv_state,
    input overlay,             //overlay active flag
```

```
input [7:0] y_avg,         //average frame brightness
output wire is_text        //states whether current pixel text or not
);

//convolution filter names
parameter SOBEL = 3'd1;
parameter NOISE = 3'd2;
parameter SHARP = 3'd3;
parameter SUN_F   = 3'd4;

//adjustment filter level names
parameter BRIGHT3 = 3'd3;
parameter BRIGHT2 = 3'd2;
parameter BRIGHT1 = 3'd1;
parameter DARK1 = 3'd7;
parameter DARK2 = 3'd6;
parameter DARK3 = 3'd5;
parameter AUTO = 3'd4;

//character memory addresses
parameter A = 0;
parameter E = 1;
parameter MINUS = 5;
parameter N = 6;
parameter ONE = 8;
parameter PLUS = 9;
parameter SUN = 12;
parameter THREE = 13;
parameter TWO = 14;
parameter ZERO = 15;
parameter S = 16;
parameter BLANK = 17;

//digit indices
parameter AVG1 = 1;
parameter AVG2 = 2;
parameter CONV = 3;
parameter SIGN = 4;
parameter LEVL = 5;

//digit locations
parameter X_AVG1 = 10'd30;
parameter X_AVG2 = 10'd75;
parameter X_CONV = 10'd475;
parameter X_SIGN = 10'd530;
parameter X_LEVL = 10'd575;
parameter Y_LOC = 9'd20;
parameter WIDTH = 32;

//address of character assignment for each digit
reg [4:0] char_conv, char_sign, char_levl;
wire [4:0] char_avg1, char_avg2;
```

```
reg [2:0] digit;    //digit at current pixel location
reg [4:0] char_in;   //address of character at current digit
wire [8:0] y_loc = Y_LOC;
reg [9:0] x_loc;    //top left corner location of current digit
wire is_pxl;

assign is_text = is_pxl && overlay;  //overlay activation flag
//calculate characters for average value (as percentage)
avg_value(.clk_in(clk_in), .y_avg(y_avg), .char_tens(char_avg1),
          .char_ones(char_avg2));

//gets font for pixel in current digit
font_reader get_txt(clk_in, hcount, vcount, x_loc, y_loc, char_in,
                    is_pxl);

always @ (*) begin
    //current digit selection - based on pixel location
    digit = (hcount >= X_AVG1 && hcount <= X_AVG1 + WIDTH) ? AVG1:
            (hcount >= X_AVG2 && hcount <= X_AVG2 + WIDTH) ? AVG2:
            (hcount >= X_CONV && hcount <= X_CONV + WIDTH) ? CONV:
            (hcount >= X_SIGN && hcount <= X_SIGN + WIDTH) ? SIGN:
            (hcount >= X_LEVL && hcount <= X_LEVL + WIDTH) ? LEVL: 0;
    //location and character assignment - based on current digit
    case (digit)
        AVG1: begin; x_loc = X_AVG1; char_in = char_avg1; end
        AVG2: begin; x_loc = X_AVG2; char_in = char_avg2; end
        CONV: begin; x_loc = X_CONV; char_in = char_conv; end
        SIGN: begin; x_loc = X_SIGN; char_in = char_sign; end
        LEVL: begin; x_loc = X_LEVL; char_in = char_levl; end
        default: begin; x_loc = 0;   char_in = BLANK; end
    Endcase
    //level adjustment character selection
    case (adj_state)
        BRIGHT3: begin char_sign = PLUS; char_levl = THREE; end
        BRIGHT2: begin char_sign = PLUS; char_levl = TWO; end
        BRIGHT1: begin char_sign = PLUS; char_levl = ONE; end
        DARK1:   begin char_sign = MINUS; char_levl = ONE; end
        DARK2:   begin char_sign = MINUS; char_levl = TWO; end
        DARK3:   begin char_sign = MINUS; char_levl = THREE; end
        AUTO:    begin char_sign = BLANK; char_levl = A; end
        default: begin char_sign = BLANK; char_levl = ZERO; end
    endcase
    //convolution filter character selection
    case (conv_state)
        SOBEL: begin char_conv = E; end
        NOISE: begin char_conv = N; end
        SHARP: begin char_conv = S; end
        SUN_F: begin char_conv = SUN; end
        default: begin char_conv = BLANK; end
    endcase
end
```

```
endmodule

//////////////////////////////////////////////////////////////////////////////
// Engineer:   jt. mcguire
//
// Module Name: ram_manager2
// Description: controls flow of data in/out of ZBT ram
//
//////////////////////////////////////////////////////////////////////////////

module ram_manager2( clk, rst, out_ready, read_x, read_y, out_x, new_out,
            in_ready, in_done, in_x, in_y, data_in, addr, word,
            data, adv, clk_en, chip_en, write_en, bwrite_en, highZ);

      input clk, rst;
      input out_ready, in_ready;
      output reg in_done;

      input [35:0] data_in;
      input [8:0] in_x, in_y, read_x, read_y;
      output[8:0] out_x;
      output reg [18:0] addr;
      output reg [35:0] word;
      inout [35:0] data;
      output adv, clk_en, chip_en;
      output write_en;
      output [3:0] bwrite_en;
      output new_out;

       output highZ;

       parameter FRAME_WIDTH=312;

       reg [17:0] mul, x_plus;
       reg [35:0] data_latch_in;

       assign chip_en = 1'b0;
       assign clk_en = 1'b0;
       assign adv = 1'b0;
       assign data = (highZ==1'b1) ? {36{1'bz}} : data_latch_in;

       reg [5:0] or_shift, id_shift;
       reg [2:0] flag0;
       reg [53:0] x0r;
       reg [8:0] inx0;
       reg [8:0] y;
       reg flag;

       assign highZ = ~id_shift[3];
       assign write_en = ~(id_shift[1] & ~flag0[0]);
       assign bwrite_en = {write_en,write_en,write_en,write_en};
```

```
        assign new_out = (~flag0[2] & or_shift[4]);
        assign out_x = x0r[44:36];

        always @(posedge clk) begin
             // RESET BEHAVIOR
             if(rst)begin
                   x0r[53:0]<=54'h0;
                   or_shift[5:0]<=6'b0;
                   id_shift[5:0]<=6'b0;
                   addr<=19'b0;
                   in_done<=1'b0;
                   mul<=18'b0;
                   y<=9'b0;
                   inx0<=9'b0;
                   x_plus<=18'b0;
                   word<=36'b0;
                   data_latch_in<=36'b0;
                   flag0<=3'b0;

             end else begin
                   id_shift[5:1]<=id_shift[4:0];
                   id_shift[0]<=in_done;

                   flag0[2:1]<=flag0[1:0];
                   flag0[0]<=flag;

                   x0r[8:0]<=read_x;
                   x0r[53:9]<=x0r[44:0];

                   or_shift[5:1]<=or_shift[4:0];
                   or_shift[0]<=out_ready;

                   inx0<=in_x;

                   mul<=y*FRAME_WIDTH;
                   x_plus<=(or_shift[0]) ? x0r[8:0] : inx0;
                   flag<=( (x0r[8:0]>=FRAME_WIDTH) || (inx0>=FRAME_WIDTH) );
                   addr <= x_plus + mul;

                   // Relatch Y on out_ready or in_ready
                   y<=out_ready ? read_y : (in_ready ? in_y : y);
                   in_done<=~out_ready&in_ready;

                   // I/O latch assignment
                   Data_latch_in <= (~out_ready&in_ready) ? data_in :
                                    data_latch_in;
                   word<= (~flag0[2] & or_shift[4]) ? data : word;
             end
        end

endmodule
```

```
//////////////////////////////////////////////////////////////////////////////
// Engineer:   jt. mcguire
//
// Module Name: **read_ram**
// Description: reads frame from ZBT memory, upscales to 1280x1024
//
//////////////////////////////////////////////////////////////////////////////

module read_ram(clk, rst, new_out,
      rx, ry, out_ready, inx, word,
      h_sync, v_sync, blank, h_blank, v_blank,
      Cr, Cb, v_count_out,
      lum1, lum2, lum3, lum4, lum5,
      out_x, out_y);

      // 1280 x 1024 @ 60 Hz with 108 MHz pixel clock
      // RAM access speed: 108/4 = 27 MHz
      // HORIZ: 1688 total, 48 after, 112 sync, 248 before
      // VERT: 1066 total, 1 after, 3 sync, 38 before

      // System signals
      input clk, rst, new_out;
      // Luminance output lines (each one is 5 adjacent 8-bit pixels)
      output reg [39:0] lum1;
      output reg [39:0] lum2;
      output reg [39:0] lum3;
      output reg [39:0] lum4;
      output reg [39:0] lum5;
      // Color output signals
      output reg [7:0] Cr, Cb;
      // VGA control signals
      output h_sync, v_sync, blank;
      output [10:0] out_x, out_y;
      // DZBT RAM control and receive lines
      output reg out_ready;
      input [35:0] word;
      output reg [8:0] rx, ry;
      output [7:0] v_count_out;
      input [8:0] inx;

      output h_blank, v_blank;

      // Line buffer index
      reg [2:0] ind;

      // Counts for horiz and vertical output
      reg [11:0] h_count;
      reg [9:0] v_count, v_count0;

      parameter LINE_WIDTH=312;
      parameter HEIGHT=480;
```

```
// Line buffer outputs
wire [35:0] out1,out2,out3,out4,out5;

// Line buffer read address
wire [8:0] x_out;

// Line buffer write enables
reg we1, we2, we3, we4, we5;

// Output lum value registers
reg [15:0] val11, val21, val31, val41;
reg [15:0] val12, val22, val32, val42;
reg [15:0] val13, val23, val33, val43;

// Output color value shift registers
//    Less depth because only need center
reg [9:0] cr23, cr22, cr21, cr33, cr32, cr31;
reg [9:0] cb23, cb22, cb21, cb33, cb32, cb31;

// Registers for old h_count[0] and h_count[1]
reg hc0_0;
reg hc1_0;

// Registers for shifting following signals
//   h_sync, v_sync, h_blank, v_blank, half line, zero value line
reg [3:0] hs0, vs0, hbl0, vbl0, hl0, vz0;

// Fill line buffer with zeros if reading out of bounds lines
wire [35:0] win;
//assign win = vz0[2] ? 36'b0 : word;
assign win = word;

// Line buffers for saving memory output
LineBuf buf1 ( .addra(inx), .addrb(x_out), .clka(clk),
.clkb(clk), .dina(win), .doutb(out1), .wea(we1) );

LineBuf buf2 ( .addra(inx), .addrb(x_out), .clka(clk),
.clkb(clk), .dina(win), .doutb(out2), .wea(we2) );

LineBuf buf3 ( .addra(inx), .addrb(x_out), .clka(clk),
.clkb(clk), .dina(win), .doutb(out3), .wea(we3) );

LineBuf buf4 ( .addra(inx), .addrb(x_out), .clka(clk),
.clkb(clk), .dina(win), .doutb(out4), .wea(we4) );

LineBuf buf5 ( .addra(inx), .addrb(x_out), .clka(clk),
.clkb(clk), .dina(win), .doutb(out5), .wea(we5) );

// Wires to calulate half pixel and half line values
wire [7:0] avg13,avg23,avg33, a13_12,a23_22,a33_32,
           avg12,avg22,avg32, a12_11,a22_21,a32_31;
```

```
wire [7:0] a12_22_0, a12_22_1, a13_23_0, a13_23_1;
wire [7:0] a22_32_0, a22_32_1, a23_33_0, a23_33_1;
wire [7:0] a32_42_0, a32_42_1, a33_43_0, a33_43_1;
wire [7:0] sq1, sq2, sq3, sq4, sq5, sq6, sq7, sq8, sq9, sq10, sq11,
           sq12;

// Function to compute the average of two values
function automatic [7:0] avg;
      input [7:0] v1, v2;
      reg [8:0] sum;
      begin
            sum=v1+v2;
            avg=sum[8:1];
      end
endfunction

// Calculate half pixel and half line values for upscaling by two
assign avg13=avg(val13[15:8],val13[7:0]);
assign a13_12=avg(val13[7:0],val12[15:8]);
assign avg12=avg(val12[15:8],val12[7:0]);
assign a12_11=avg(val12[7:0],val11[15:8]);
assign avg23=avg(val23[15:8],val23[7:0]);
assign a23_22=avg(val23[7:0],val22[15:8]);
assign avg22=avg(val22[15:8],val22[7:0]);
assign a22_21=avg(val22[7:0],val21[15:8]);
assign avg33=avg(val33[15:8],val33[7:0]);
assign a33_32=avg(val33[7:0],val32[15:8]);
assign avg32=avg(val32[15:8],val32[7:0]);
assign a32_31=avg(val32[7:0],val31[15:8]);

assign a12_22_0=avg(val12[15:8],val22[15:8]);
assign a22_32_0=avg(val22[15:8],val32[15:8]);
assign a32_42_0=avg(val32[15:8],val42[15:8]);
assign a12_22_1=avg(val12[7:0],val22[7:0]);
assign a22_32_1=avg(val22[7:0],val32[7:0]);
assign a32_42_1=avg(val32[7:0],val42[7:0]);
assign a13_23_0=avg(val13[15:8],val23[15:8]);
assign a23_33_0=avg(val23[15:8],val33[15:8]);
assign a33_43_0=avg(val33[15:8],val43[15:8]);
assign a13_23_1=avg(val13[7:0],val23[7:0]);
assign a23_33_1=avg(val23[7:0],val33[7:0]);
assign a33_43_1=avg(val33[7:0],val43[7:0]);

assign sq1=avg(avg13,avg23); assign sq2=avg(a13_12,a23_22);
assign sq3=avg(avg12,avg22); assign sq4=avg(a12_11,a22_21);
assign sq5=avg(avg23,avg33); assign sq6=avg(a23_22,a33_32);
assign sq7=avg(avg22,avg32); assign sq8=avg(a22_21,a32_31);
assign sq9=avg(avg(val43[15:8], val43[7:0]),avg33);
assign sq10=avg(avg(val43[7:0], val42[15:8]),a33_32);
assign sq11=avg(avg(val42[15:8],val42[7:0]),avg32);
assign sq12=avg(avg(val42[7:0],val41[15:8]),a32_31);
```

```
// Wires to calulate half pixel and hlaf line colors
wire [7:0] cr_23_1, cr_23_22, cr_22_0, cr_a22, cr_23_33_1, cr_sq6,
           cr_22_32_0, cr_sq7;
wire [7:0] cb_23_1, cb_23_22, cb_22_0, cb_a22, cb_23_33_1, cb_sq6,
           cb_22_32_0, cb_sq7;

// Calculate the color values for center four target pixels in all
// cases
assign cr_23_1 = {cr23[4:0], 3'b0};
assign cr_23_22 = avg( {cr23[4:0], 3'b0}, {cr22[9:5],3'b0} );
assign cr_22_0 = {cr22[9:5], 3'b0};
assign cr_a22 = avg( {cr22[9:5], 3'b0}, {cr22[4:0], 3'b0} );
assign cr_23_33_1 = avg( {cr23[4:0], 3'b0}, {cr33[4:0], 3'b0} );
assign cr_sq6 = avg( cr_23_22, avg( {cr33[4:0], 3'b0}, {cr32[9:5],
                    3'b0} ) );
assign cr_22_32_0 = avg( {cr22[9:5], 3'b0}, {cr32[9:5], 3'b0} );
assign cr_sq7 = avg( cr_a22, avg( {cr32[4:0], 3'b0}, {cr32[9:5],
                    3'b0} ) );

assign cb_23_1 = {cb23[4:0], 3'b0};
assign cb_23_22 = avg( {cb23[4:0], 3'b0}, {cb22[9:5],3'b0} );
assign cb_22_0 = {cb22[9:5], 3'b0};
assign cb_a22 = avg( {cb22[9:5], 3'b0}, {cb22[4:0], 3'b0} );
assign cb_23_33_1 = avg( {cb23[4:0], 3'b0}, {cb33[4:0], 3'b0} );
assign cb_sq6 = avg( cb_23_22, avg( {cb33[4:0], 3'b0}, {cb32[9:5],
                    3'b0} ) );
assign cb_22_32_0 = avg( {cb22[9:5], 3'b0}, {cb32[9:5], 3'b0} );
assign cb_sq7 = avg( cb_a22, avg( {cb32[4:0], 3'b0}, {cb32[9:5],
                    3'b0} ) );

// Wires for instantaneous sync, blank, and half-pixel/half-line
// controls
wire vs,hs,bl,hp,op,hl,hlx;
reg hz0;

// Vertical sync when v_count in range+2 (because of 2 line delay in
// buffers)
assign vs = (v_count>=515 && v_count<=516);
// Horizantal sync when h_count in range for normal and half lines
assign hs = (h_count>=3016 && h_count<3128) || (h_count>=1328 &&
            h_count<1440);
// Horizantal blank when h_count exceeds limits for zero and half
// lines
assign hbl = (h_count>=2968 || (h_count>=1280 && h_count<1688));
// Line outside bounds detection
assign vzero = (v_count>=480);
// V blank signal when v_count in range+2 (2 line delay)
assign vbl = (v_count>=514 || v_count<=2);
// Half pixel detection from 1 cycle delay hcount[0]
assign hp = hc0_0 & ~hbl0[0] & ~vbl0[0];
// Odd pixel detection from 1 delay hcount[0]
```

```
assign op = hc1_0 & ~hbl0[0] & ~vbl0[0];
// Half line instantaneous detection
assign hlx = (h_count>=1687);
// Actual half line signal (properly delayed)
assign hl = hl0[0] & ~hbl0[0] & ~vbl0[0];

// Actual output sync and blank signals from delayed lines
assign h_sync = hs0[2];
assign v_sync = vs0[2];
assign blank = hbl0[2] | vbl0[2];
assign h_blank = hbl0[2];
assign v_blank = vbl0[2];

assign v_count_out = v_count[9:2];

always @* begin
      // Assign write enable signals to proper write buffer (5th
      // line)
      we1 = (ind==3'd0) & inx<LINE_WIDTH;
      we2 = (ind==3'd1) & inx<LINE_WIDTH;
      we3 = (ind==3'd2) & inx<LINE_WIDTH;
      we4 = (ind==3'd3) & inx<LINE_WIDTH;
      we5 = (ind==3'd4) & inx<LINE_WIDTH;

      // Assign value lines based on circular buffer structure
      //    val1X is oldest (highest line) >>>> val4X is newest
      //    (lowest line)
      //    Also yank out corresponding color signals for center two
      //    rows
      if(hbl0[0] | hz0)begin
            val11 = 16'b0;
            val21 = 16'b0;
            val31 = 16'b0;
            val41 = 16'b0;
            cr21 = 10'b0;
            cb21 = 10'b0;
            cr31 = 10'b0;
            cb31 = 10'b0;
      end else if(ind==3'd0)begin
            val11 = {out2[35:28] , out2[17:10]};
            val21 = {out3[35:28] , out3[17:10]};
            val31 = {out4[35:28] , out4[17:10]};
            val41 = {out5[35:28] , out5[17:10]};
            cr21 = {out3[27:23] , out3[9:5]};
            cb21 = {out3[22:18] , out3[4:0]};
            cr31 = {out4[27:23] , out4[9:5]};
            cb31 = {out4[22:18] , out4[4:0]};
      end else if(ind==3'd1)begin
            val11 = {out3[35:28] , out3[17:10]};
            val21 = {out4[35:28] , out4[17:10]};
            val31 = {out5[35:28] , out5[17:10]};
            val41 = {out1[35:28] , out1[17:10]};
```

```
      cr21 = {out4[27:23] , out4[9:5]};
      cb21 = {out4[22:18] , out4[4:0]};
      cr31 = {out5[27:23] , out5[9:5]};
      cb31 = {out5[22:18] , out5[4:0]};
end else if(ind==3'd2)begin
      val11 = {out4[35:28] , out4[17:10]};
      val21 = {out5[35:28] , out5[17:10]};
      val31 = {out1[35:28] , out1[17:10]};
      val41 = {out2[35:28] , out2[17:10]};
      cr21 = {out5[27:23] , out5[9:5]};
      cb21 = {out5[22:18] , out5[4:0]};
      cr31 = {out1[27:23] , out1[9:5]};
      cb31 = {out1[22:18] , out1[4:0]};
end else if(ind==3'd3)begin
      val11 = {out5[35:28] , out5[17:10]};
      val21 = {out1[35:28] , out1[17:10]};
      val31 = {out2[35:28] , out2[17:10]};
      val41 = {out3[35:28] , out3[17:10]};
      cr21 = {out1[27:23] , out1[9:5]};
      cb21 = {out1[22:18] , out1[4:0]};
      cr31 = {out2[27:23] , out2[9:5]};
      cb31 = {out2[22:18] , out2[4:0]};
end else if(ind==3'd4)begin
      val11 = {out1[35:28] , out1[17:10]};
      val21 = {out2[35:28] , out2[17:10]};
      val31 = {out3[35:28] , out3[17:10]};
      val41 = {out4[35:28] , out4[17:10]};
      cr21 = {out2[27:23] , out2[9:5]};
      cb21 = {out2[22:18] , out2[4:0]};
      cr31 = {out3[27:23] , out3[9:5]};
      cb31 = {out3[22:18] , out3[4:0]};
end else begin
      val11 = 16'b0;
      val21 = 16'b0;
      val31 = 16'b0;
      val41 = 16'b0;
      cr21 = 10'b0;
      cb21 = 10'b0;
      cr31 = 10'b0;
      cb31 = 10'b0;
end

//cr_23_1, cr_23_22, cr_22_0, cr_a22, cr_23_33_1, cr_sq6,
// cr_22_32_0, cr_sq7;

// Select the proper output values based on the target frame
case ({op,hp,hl})
      3'b000: begin
                        lum1={val13[15:8], avg13, val13[7:0],
                              a13_12, val12[15:8]};
                        lum2={a13_23_0, sq1, a13_23_1, sq2,
                              a12_22_0};
```

```
                    lum3={val23[15:8], avg23, val23[7:0],
                         a23_22, val22[15:8]};
                    lum4={a23_33_0, sq5, a23_33_1, sq6,
                         a22_32_0};
                    lum5={val33[15:8], avg33, val33[7:0],
                         a33_32, val32[15:8]};
                    Cr=cr_23_1; Cb=cb_23_1;
                end
            3'b001: begin
                    lum1={a13_23_0, sq1, a13_23_1, sq2,
                         a12_22_0};
                    lum2={val23[15:8], avg23, val23[7:0],
                         a23_22, val22[15:8]};
                    lum3={a23_33_0, sq5, a23_33_1, sq6,
                         a22_32_0};
                    lum4={val33[15:8], avg33, val33[7:0],
                         a33_32, val32[15:8]};
                    lum5={a33_43_0, sq9, a33_43_1, sq10,
                         a32_42_0};
                    Cr=cr_23_33_1; Cb=cb_23_33_1;
                end
            3'b010: begin
                    lum1={avg13, val13[7:0], a13_12,
                         val12[15:8], avg12};
                    lum2={sq1, a13_23_1, sq2, a12_22_0,
                         sq3};
                    lum3={avg23, val23[7:0], a23_22,
                         val22[15:8], avg22};
                    lum4={sq5, a23_33_1, sq6, a22_32_0,
                         sq7};
                    lum5={avg33, val33[7:0], a33_32,
                         val32[15:8], avg32};
                    Cr=cr_23_22; Cb=cb_23_22;
                end
            3'b011: begin
                    lum1={sq1, a13_23_1, sq2, a12_22_0,
                         sq3};
                    lum2={avg23, val23[7:0], a23_22,
                         val22[15:8], avg22};
                    lum3={sq5, a23_33_1, sq6, a22_32_0,
                         sq7};
                    lum4={avg33, val33[7:0], a33_32,
                         val32[15:8], avg32};
                    lum5={sq9, a33_43_1, sq10, a32_42_0,
                         sq11};
                    Cr=cr_sq6; Cb=cb_sq6;
                end
            3'b100: begin
                    lum1={val13[7:0], a13_12, val12[15:8],
                         avg12, val12[7:0]};
                    lum2={a13_23_1, sq2, a12_22_0, sq3,
                         a12_22_1};
```

```
                                lum3={val23[7:0], a23_22, val22[15:8],
                                     avg22, val22[7:0]};
                                lum4={a23_33_1, sq6, a22_32_0, sq7,
                                     a22_32_1};
                                lum5={val33[7:0], a33_32, val32[15:8],
                                     avg32, val32[7:0]};
                                Cr=cr_22_0; Cb=cb_22_0;
                            end
                3'b101: begin
                                lum1={a13_23_1, sq2, a12_22_0, sq3,
                                     a12_22_1};
                                lum2={val23[7:0], a23_22, val22[15:8],
                                     avg22, val22[7:0]};
                                lum3={a23_33_1, sq6, a22_32_0, sq7,
                                     a22_32_1};
                                lum4={val33[7:0], a33_32, val32[15:8],
                                     avg32, val32[7:0]};
                                lum5={a33_43_1, sq10, a32_42_0, sq11,
                                     a32_42_1};
                                Cr=cr_22_32_0; Cb=cb_22_32_0;
                            end
                3'b110: begin
                                lum1={a13_12, val12[15:8], avg12,
                                     val12[7:0], a12_11};
                                lum2={sq2, a12_22_0, sq3, a12_22_1,
                                     sq4};
                                lum3={a23_22, val22[15:8], avg22,
                                     val22[7:0], a22_21};
                                lum4={sq6, a22_32_0, sq7, a22_32_1,
                                     sq8};
                                lum5={a33_32, val32[15:8], avg32,
                                     val32[7:0], a32_31};
                                Cr=cr_a22; Cb=cb_a22;
                            end
                3'b111: begin
                                lum1={sq2, a12_22_0, sq3, a12_22_1,
                                     sq4};
                                lum2={a23_22, val22[15:8], avg22,
                                     val22[7:0], a22_21};
                                lum3={sq6, a22_32_0, sq7, a22_32_1,
                                     sq8};
                                lum4={a33_32, val32[15:8], avg32,
                                     val32[7:0], a32_31};
                                lum5={sq10, a32_42_0, sq11, a32_42_1,
                                     sq12};
                                Cr=cr_sq7; Cb=cb_sq7;
                            end
        endcase

end

// Define an h_count with 1 line subtracted away
```

```
wire [11:0] h_count_sub;
assign h_count_sub = (h_count-12'd1688);
      // Assign the proper output address as 1/4th of half line
      // corrected h_count value
//          Rescales h_count to range 0-312
assign x_out = (h_count>=1688) ? h_count_sub[10:2] : h_count[10:2];

assign out_x = ((h_count>=1688) ? h_count_sub : h_count) - 11'd9;
assign out_y = (v_count0<3) ? 11'b0 : {v_count0-10'd3,
               (h_count>1600)};

// System clock edged behavior
always @(posedge clk) begin
      // Zero behavior
      if(rst)begin
            // Zero the hcount shift regs
            hc0_0<=1'b0;
            hc1_0<=1'b0;
            hz0<=1'b0;
            // Zero the count registers
            rx<=9'b0;
            ry<=9'b0;
            ind<=3'b0;
            h_count<=12'b0;
            v_count<=10'b0;
            // Set luminance shift values to zero
            val12<=16'b0;
            val22<=16'b0;
            val32<=16'b0;
            val42<=16'b0;
            val13<=16'b0;
            val23<=16'b0;
            val33<=16'b0;
            val43<=16'b0;
            // Set color shift values to zero
            cr22<=10'b0;
            cb22<=10'b0;
            cr32<=10'b0;
            cb32<=10'b0;
            cr23<=10'b0;
            cb23<=10'b0;
            cr33<=10'b0;
            cb33<=10'b0;
            // Set the Hsync, Vsync, zero detect, and blank shift
            //regs to zero
            vz0[2:0]<=3'b0;
            hs0[2:0]<=3'b0;
            vs0[2:0]<=3'b0;
            hbl0[2:0]<=3'b0;
            vbl0[2:0]<=3'b0;
            hl0[2:0]<=3'b0;
            v_count0<=10'b0;
```

```
end else begin
      // Shift back h_count[0] and [1] to match line buf output
      // delay
      hc0_0<=h_count[0];
      hc1_0<=h_count[1];
      v_count0<=v_count;

      // Shift back output signals to match line buf output
      // delay
      hs0[0]<=hs;
      vs0[0]<=vs;
      hbl0[0]<=hbl;
      vbl0[0]<=vbl;
      hl0[0]<=hlx;
      vz0[0]<=vzero;
      hz0<=(x_out>9'd311);
      // Shift zero detection back to match ZBT RAM output
      vz0[2:1]<=vz0[1:0];
      // Trigger the ZBT RAM read on every eighth h_count when
      // in bounds
      Out_ready <= (h_count[2:0]==3'b0) &&
                   (h_count[11:3]<LINE_WIDTH); // & ~vzero
      // Assign ZBT RAM address lines
      rx<=h_count[11:3];
      ry<=v_count;

      // Received a frame refresh signal
      if(v_count>533)begin
            // Zero everything and the index
            h_count<=12'b0;
            v_count<=10'b0;
            ind<=3'b0;
      // H count exceeded two-line output count
      end else if(h_count>=3375)begin
            // Zero h count
            h_count<=12'b0;
            // Increment v count
            v_count<=v_count+1;
            // Increment index and wrap if needed
            ind<= (ind>=4) ? 3'b0 : ind+1;
      end else begin
            // Increment h_count
            h_count<=h_count+1;
            // Recap v_count
            v_count<=v_count;
            // Recap the line buffer index
            ind<=ind;
      end

      if(hc1_0 & ~h_count[1])begin
            // Shift h,v,blank,hl back
```

```
                        hs0[3:1]<=hs0[2:0];
                        vs0[3:1]<=vs0[2:0];
                        hbl0[3:1]<=hbl0[2:0];
                        vbl0[3:1]<=vbl0[2:0];
                        hl0[3:1]<=hl0[2:0];
                        // Shift zeros if h blanking detected, otherwise
                        // choose values
                        val12<=val11;
                        val22<=val21;
                        val32<=val31;
                        val42<=val41;
                        val13<=val12;
                        val23<=val22;
                        val33<=val32;
                        val43<=val42;
                        // Shift colors back (0 if h blanking detected)
                        cr22<=cr21;
                        cb22<=cb21;
                        cr32<=cr31;
                        cb32<=cb31;
                        cr23<=cr22;
                        cb23<=cb22;
                        cr33<=cr32;
                        cb33<=cb32;
                    end

                    // 1280 x 1024 @ 60 Hz with 108 MHz pixel clock
                    // RAM access speed: 108/4 = 27 MHz
                    // HORIZ: 1688 total, 48 after, 112 sync, 248 before
                    // VERT: 1066 total, 1 after, 3 sync, 38 before
                end
        end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   jt.mcguire
//
// Module Name: read_ram2
// Description: simplification of above module, outputs 640x480 frame with
//              no upscaling
//////////////////////////////////////////////////////////////////////////////////

module read_ram2(clk, sysclk, rst, new_out,
        rx, ry, out_ready, inx, word,
        h_sync, v_sync, blank,
        Cr, Cb, lum0, lum1, lum2, lum3, lum4, lum5, lum6, lum7, lum8,
        out_x, out_y);

        input clk, sysclk, rst;

        input new_out;
```

```
	input [8:0] inx;
	input [35:0] word;

	output h_sync, v_sync, blank;
	output [9:0] out_x, out_y;
	output reg [7:0] Cr, Cb, lum0, lum1, lum2, lum3, lum4, lum5, lum6,
lum7, lum8;

	output [8:0] rx;
	output reg [8:0] ry;
	output out_ready;

	reg [9:0] h_count, v_count;

//    Visible area 640   25.422045680238
//    Front porch  16    0.63555114200596
//    Sync pulse   96    3.8133068520357
//    Back porch   48    1.9066534260179
//    Whole line   800   31.777557100298

//Visible area    480   15.253227408143
//Front porch     10    0.31777557100298
//Sync pulse 2    0.063555114200596
//Back porch 33   1.0486593843098
//Whole frame     525   16.683217477656

	assign h_sync = (h_count>655) && (h_count<=751);
	assign v_sync = (v_count>489) && (v_count<=491);
	assign blank = h_count>=640 || v_count>=480;

	wire [8:0] oldh;
	wire [35:0] buf_out1, buf_out2, buf_out3, buf_out4;
	reg we1, we2, we3, we4;

	wire [35:0] word_in;
	assign word_in = inx<480 ? word : 35'b0;

	LineBuf buf1 (.addra(inx), .addrb(oldh), .clka(clk),    .clkb(clk),
	.dina(word_in),    .doutb(buf_out1), .wea(we1));

	LineBuf buf2 (.addra(inx), .addrb(oldh), .clka(clk),    .clkb(clk),
	.dina(word_in),    .doutb(buf_out2), .wea(we2));

	LineBuf buf3 (.addra(inx), .addrb(oldh), .clka(clk),    .clkb(clk),
	.dina(word_in),    .doutb(buf_out3), .wea(we3));

	LineBuf buf4 (.addra(inx), .addrb(oldh), .clka(clk),    .clkb(clk),
	.dina(word_in),    .doutb(buf_out4), .wea(we4));

	reg [1:0] ind;
	reg we10,we20,we30,we40;
	reg [35:0] word0,word00,word1,word10,word2,word20;
```

```
reg out_r, out_r0, out_r00;
assign rx = h_count[9:1];
always @* begin
      if(v_count==524)begin
            ry=9'd1;
            out_r = (h_count[9:1]<312) & ~h_count[0];
      end else if(v_count==523)begin
            ry=9'd0;
            out_r = (h_count[9:1]<312) & ~h_count[0];
      end else if(v_count<480)begin
            ry=v_count+9'd2;
            out_r = (h_count[9:1]<312) & ~h_count[0];
      end else begin
            ry=9'd0;
            out_r = 1'b0;
      end

      case (ind)
            2'b00:      begin
                              word0=buf_out1;
                              word1=buf_out2;
                              word2=buf_out3;
                              we40=(inx<312);
                              we30=1'b0;
                              we20=1'b0;
                              we10=1'b0;
                              end
            2'b01:begin
                              word0=buf_out2;
                              word1=buf_out3;
                              word2=buf_out4;
                              we40=1'b0;
                              we30=1'b0;
                              we20=1'b0;
                              we10=(inx<312);
                              end
            2'b10:begin
                              word0=buf_out3;
                              word1=buf_out4;
                              word2=buf_out1;
                              we40=1'b0;
                              we30=1'b0;
                              we20=(inx<312);
                              we10=1'b0;
                              end
            2'b11:begin
                              word0=buf_out4;
                              word1=buf_out1;
                              word2=buf_out2;
                              we40=1'b0;
                              we30=(inx<312);
```

```
                                    we20=1'b0;
                                    we10=1'b0;
                                    end
      endcase
end

assign out_ready = out_r0&~out_r00;

always @(posedge sysclk)begin
      out_r0<=out_r;
      out_r00<=out_r0;
      we4<=we40;
      we3<=we30;
      we2<=we20;
      we1<=we10;
end

// Assign the read adcdress to the output brams
assign oldh = h_count<624 ? h_count[9:1] : 9'b0;

always @(posedge clk)begin
      if(rst)begin
            h_count<=10'b0;
            v_count<=10'b0;
            ind<=2'b0;
      end else begin
            word00<=h_count[0] ? word0 : word00;
            word10<=h_count[0] ? word1 : word10;
            word20<=h_count[0] ? word2 : word20;
            if(h_count>=799)begin
                  h_count<=10'b0;
                  ind<=ind+1;
                  if(v_count>=524)begin
                        v_count<=10'b0;
                  end else begin
                        v_count<=v_count+1;
                  end
            end else begin
                  h_count<=h_count+1;
                  v_count<=v_count;
                  ind<=ind;
            end

            if(v_count>=2 && v_count<=481 && h_count>=2 &&
               h_count<=626)begin
                  if(~h_count[0])begin
                        lum0<=word00[35:28];
                        lum1<=word00[17:10];
                        lum2<=word0[35:28];
                        lum3<=word10[35:28];
                        lum4<=word10[17:10];
                        lum5<=word1[35:28];
```

```
                                lum6<=word20[35:28];
                                lum7<=word20[17:10];
                                lum8<=word2[35:28];
                                Cr<={word10[9:5],3'b0};
                                Cb<={word10[4:0],3'b0};
                          end else begin
                                lum0<=word00[17:10];
                                lum1<=word0[35:28];
                                lum2<=word0[17:10];
                                lum3<=word10[17:10];
                                lum4<=word1[35:28];
                                lum5<=word1[17:10];
                                lum6<=word20[17:10];
                                lum7<=word2[35:28];
                                lum8<=word2[17:10];
                                Cr<={word1[27:23],3'b0};
                                Cb<={word1[22:18],3'b0};
                          end
                    end else begin
                          lum0<=8'b0;
                          lum1<=8'b0;
                          lum2<=8'b0;
                          lum3<=8'b0;
                          lum4<=8'b0;
                          lum5<=8'b0;
                          lum6<=8'b0;
                          lum7<=8'b0;
                          lum8<=8'b0;
                          Cr<=8'b0;
                          Cb<=8'b0;
                    end
              end
        end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j.abel
//
// Module Name: sharpen_filt
// Description: 3x3 sharpening filter
//
//////////////////////////////////////////////////////////////////////////////////

module sharpen_filt(
    input clk_in,
    input [71:0] y_in,
    output reg [7:0] y_out
    );

    parameter BLACK = 8'd16;
    parameter WHITE = 8'd235;
```

```
    // positive and negative sums in kernel
    reg [9:0] y_pos;
    reg [9:0] y_neg;

    always @(*) begin
    //output assignment with clipping
        y_out = (y_neg > y_pos) ? BLACK  : ((y_pos-y_neg) > WHITE ) ? WHITE
                : y_pos - y_neg;
    end

    always @ (posedge clk_in) begin
        // convolution arithmetic
        y_pos <= y_in[39:32]*3;
        y_neg <= (y_in[63:56]+y_in[47:40]+y_in[31:24]+y_in[15:8])>>1;
    end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j.abel
//
// Module Name: ** sharpen_filt_5 **
// Description: 5x5 sharpening filter (unsharp mask) - not implemented
//
//////////////////////////////////////////////////////////////////////////////////

module sharpen_filt(
    input clk_in,
    input [199:0] y_in,        //5x5 convolution kernel
    output reg [7:0] y_out
    );

    parameter BLACK = 8'd16;
    parameter WHITE = 8'd235;

    // positive and negative sums in kernel
    reg [16:0] y_pos;
    reg [16:0] y_neg;

    // pixel assignment
    wire [7:0] y_1 = y_in[199:192];
    wire [7:0] y_2 = y_in[191:184];
    wire [7:0] y_3 = y_in[183:176];
    wire [7:0] y_4 = y_in[175:168];
    wire [7:0] y_5 = y_in[167:160];
    wire [7:0] y_6 = y_in[159:152];
    wire [7:0] y_7 = y_in[151:144];
    wire [7:0] y_8 = y_in[143:136];
    wire [7:0] y_9 = y_in[135:128];
    wire [7:0] y_10 = y_in[127:120];
    wire [7:0] y_11 = y_in[119:112];
```

```
    wire [7:0] y_12 = y_in[111:104];
    wire [7:0] y_13 = y_in[103:96];
    wire [7:0] y_14 = y_in[95:88];
    wire [7:0] y_15 = y_in[87:80];
    wire [7:0] y_16 = y_in[79:72];
    wire [7:0] y_17 = y_in[71:64];
    wire [7:0] y_18 = y_in[63:56];
    wire [7:0] y_19 = y_in[55:48];
    wire [7:0] y_20 = y_in[47:40];
    wire [7:0] y_21 = y_in[39:32];
    wire [7:0] y_22 = y_in[31:24];
    wire [7:0] y_23 = y_in[23:16];
    wire [7:0] y_24 = y_in[15:8];
    wire [7:0] y_25 = y_in[7:0];

    always @ (*) begin
        //clipped output assignment
        y_out = (y_neg > y_pos) ? BLACK : ((y_pos-y_neg) > WHITE) ? WHITE :
                 y_pos-y_neg;
    end

    always @ (posedge clk_in) begin
        //convolution filter
        y_neg <= (y_1 + y_5 + y_21 + y_25 +
                 + (y_2 + y_4 + y_6 + y_10 + y_16 + y_20 + y_22 + y_24)*4
                 + (y_7 + y_9 + y_17 + y_19)*16
                 + (y_3 + y_11 + y_15 + y_23)*6
                 + (y_8 + y_12 + y_14 + y_18)*24) >> 8;
        y_pos <= (y_13 * 476) >> 8;
    end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j.abel
//
// Module Name: sun_filt
// Description: sun blocking filter
//
//////////////////////////////////////////////////////////////////////////////////

module sun_filt(
    input clk_in,
    input [71:0] y_in,          //3x3 kernel - change to [199:0] for 5x5
    input [2:0] lvl_state,       //state of level adjustment filter
    input [7:0] y_avg,
    output reg [7:0] y_out
    );

    parameter BLACK = 8'd16;

    //pixel assignment
```

```
    wire [7:0] y_1 = y_in[71:64];
    wire [7:0] y_2 = y_in[63:56];
    wire [7:0] y_3 = y_in[55:48];
    wire [7:0] y_4 = y_in[47:40];
    wire [7:0] y_5 = y_in[39:32];
    wire [7:0] y_6 = y_in[31:24];
    wire [7:0] y_7 = y_in[23:16];
    wire [7:0] y_8 = y_in[15:8];
    wire [7:0] y_9 = y_in[7:0];

    wire [8:0] s;              //outputs of threshold operation
    wire [3:0] count;         //number of pixels above the threshold

    //thresholding operations for each pixel
    thresholder t1 (.y_in(y_1), .lvl_state(lvl_state), .y_avg(y_avg),
                    .p_out(s[0]));
    thresholder t2 (.y_in(y_2), .lvl_state(lvl_state), .y_avg(y_avg),
                    .p_out(s[1]));
    thresholder t3 (.y_in(y_3), .lvl_state(lvl_state), .y_avg(y_avg),
                    .p_out(s[2]));
    thresholder t4 (.y_in(y_4), .lvl_state(lvl_state), .y_avg(y_avg),
                    .p_out(s[3]));
    thresholder t5 (.y_in(y_5), .lvl_state(lvl_state), .y_avg(y_avg),
                    .p_out(s[4]));
    thresholder t6 (.y_in(y_6), .lvl_state(lvl_state), .y_avg(y_avg),
                    .p_out(s[5]));
    thresholder t7 (.y_in(y_7), .lvl_state(lvl_state), .y_avg(y_avg),
                    .p_out(s[6]));
    thresholder t8 (.y_in(y_8), .lvl_state(lvl_state), .y_avg(y_avg),
                    .p_out(s[7]));
    thresholder t9 (.y_in(y_9), .lvl_state(lvl_state), .y_avg(y_avg),
                    .p_out(s[8]));

    //count darkened pixels, assign output
    assign count = s[0]+s[1]+s[2]+s[3]+s[4]+s[5]+s[6]+s[7]+s[8];
    assign y_out = (count >= 5) ? BLACK : y_5;

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j.abel
//
// Module Name: thresholder
// Description: determines whether pixel above threshold - submodule of
//              sun_filt
//////////////////////////////////////////////////////////////////////////////////

module thresholder(
    input [7:0] y_in,
    [2:0] lvl_state,
    [7:0] y_avg,
```

```
    output reg p_out
    );

    //level adjustment filter states (only those that make pixels brighter)
    parameter BRIGHT3 = 3'd3;
    parameter BRIGHT2 = 3'd2;
    parameter BRIGHT1 = 3'd1;
    parameter AUTO = 3'd4;

    always @ (*) begin
        case (lvl_state)
            //thresholding with automatic level adjustment
            AUTO: begin p_out = ((y_in >= 8'd191 && y_avg < 8'd72) ||
                                 (y_in >= 8'd234 && y_avg >= 8'd72)) ?
                                 1 : 0; end
            //thresholding with manual level adjustment
            BRIGHT3: begin p_out = (y_in >= 8'd125) ? 1 :0; end
            BRIGHT2: begin p_out = (y_in >= 8'd162) ? 1 :0; end
            BRIGHT1: begin p_out = (y_in >= 8'd191) ? 1 :0; end
            default: begin p_out = (y_in >= 8'd234) ? 1 :0; end
        endcase
    end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   jt.mcguire
//
// Module Name: TV_i2c
// Description: I2C controller to control the ADV7185 video decoder
//
//////////////////////////////////////////////////////////////////////////////////

module TV_i2c(
      rst, clk,
      i2c_data, i2c_clock,
      isAcked, sending,
      byteAddr, dout  );

       parameter CNT = 10000;
       parameter DAT_TRIG = CNT/4;
       parameter CLK_HIGH = CNT/2;
       parameter TOTAL_BYTES = 9;   // MUST BE A MULTIPLE OF 3

       inout i2c_data;
       inout i2c_clock;
       input clk, rst;
       output reg isAcked;
       output sending;
       input [7:0] dout;
       output reg [3:0] byteAddr;
```

```
reg dat, clock;

reg [13:0] count;

assign i2c_data = dat ? 1'bZ : 1'b0;
assign i2c_clock = clock ? 1'b1 : 1'b0;

reg start, stop;
reg ack;
reg [2:0] bitAddr;
reg datTrig, clkHigh, clkLow;

assign sending = ~(byteAddr>=TOTAL_BYTES && stop && dat && clock);

// System clock triggered behavior
always @(posedge clk) begin
     if(rst)begin
           // Zero all triggers and counts
           datTrig<=1'b0;
           clkHigh<=1'b0;
           clkLow<=1'b0;
           count<=14'b0;
     end else begin
           // Data trigger happens first
           if(count==DAT_TRIG)begin
                 // Pulse data trigger line
                 datTrig<=1'b1;
                 clkLow<=1'b0;
                 clkHigh<=1'b0;
                 count<=count+1;
           // Clock pulled low after data triggered
           end else if(count==CLK_HIGH)begin
                 // Pulse the clock low line
                 clkLow<=1'b0;
                 datTrig<=1'b0;
                 clkHigh<=1'b1;
                 count<=count+1;
           // Clock pulled high, shifts data in on slave
           end else if(count>=CNT)begin
                 // Pulse the clock high line
                 clkHigh<=1'b0;
                 datTrig<=1'b0;
                 clkLow<=1'b1;
                 count<=14'b0;
           // Increment the counter and end the pulses
           end else begin
                 clkHigh<=1'b0;
                 clkLow<=1'b0;
                 datTrig<=1'b0;
                 count<=count+1;
           end
     end
```

```
        end

        reg hold;
        reg [1:0] dead;
        wire complete;
        wire div3;
        assign complete = (byteAddr>(TOTAL_BYTES-1));
        assign div3 = (byteAddr==4'd3 || byteAddr==4'd6 || byteAddr==4'd9 ||
byteAddr==4'd12 || byteAddr==4'd15);
        wire all_trig;
        BUFG buf2 (.I(clkHigh | clkLow | datTrig | (rst&clk)), .O(all_trig) );

        // Triggered behavior
        always @(posedge all_trig) begin
             if(rst)begin
                   bitAddr<=3'b111;
                   byteAddr<=4'b0;
                   ack<=1'b0;
                   isAcked<=1'b0;
                   stop<=1'b0;
                   clock<=1'b1;
                   dat<=1'b1;
                   start<=1'b1;
                   hold<=1'b1;
                   dead<=2'b0;
             end else if(start)begin
                   bitAddr<=3'b111;
                   byteAddr<=byteAddr;
                   ack<=1'b0;
                   isAcked<=1'b0;
                   stop<=1'b0;
                   clock<=~clkLow;
                   dat<=datTrig ? 1'b0 : dat;
                   start<=~clkLow;
                   hold<=1'b1;
                   dead<=2'b0;

             // CLOCK CHANGE BEHAVIOR
             end else if(clkHigh | clkLow)begin
                   dat<=dat;
                   clock<=stop ? 1'b1 : clkHigh;
                   // Recap anything else
                   bitAddr<=bitAddr;
                   byteAddr<=byteAddr;
                   ack<=ack;
                   isAcked<=(clkHigh & ack) ? i2c_data : isAcked;
                   stop<=stop;
                   start<=start;
                   hold<=1'b0;
                   dead<=dead;

             // DATA TRIGGER BEHAVIOR
```

```
end else begin
	// Recap clock
	clock<=clock;
	isAcked<=isAcked;
	if(bitAddr==3'b0)begin
		byteAddr<=byteAddr+1;
		// Next bit address is 7
		bitAddr<=3'b111;
		// Indicate an ack bit is next
		ack<=1'b1;
		// Pull data high (lets slave communicate)
		dat<=1'b1;
		stop<=stop;
		hold<=hold;
		start<=start;
	// Ack bit next
	end else if(ack)begin
		// Zero the ack and bit address
		bitAddr<=3'b111;
		ack<=1'b0;
		// Recap all else
		byteAddr<=byteAddr;
		if(complete | div3)begin
			stop<=1'b1;
			dat<=1'b0;
		end else begin
			stop<=1'b0;
			// Set data to next ROM value
			dat<=dout[bitAddr];
		end
		hold<=hold;
		start<=start;
	end else begin
		// Stop indicated
		if(hold)begin
			bitAddr<=3'b111;
			dat<=dout[bitAddr];
			stop<=stop;
			hold<=hold;
			start<=start;
		end else if(stop)begin
			bitAddr<=bitAddr;
			dat<=(dead<=2'b10) | complete;
			if(dead==2'b11 && ~complete)begin
				stop<=1'b0;
				hold<=1'b1;
				start<=1'b1;
				dead<=2'b0;
			end else begin
				dead<=dead+1;
				stop<=1'b1;
				hold<=1'b0;
```

```
                                    start<=1'b0;
                              end
                        end else begin
                              // Increment bit address
                              bitAddr<=bitAddr-1;
                              // Set data to next ROM value
                              dat<=dout[bitAddr-1];
                              stop<=stop;
                              hold<=hold;
                              start<=start;
                        end
                        // Recap or reassign all else
                        byteAddr<=byteAddr;
                        ack<=1'b0;
                  end
            end
      end

endmodule

//////////////////////////////////////////////////////////////////////////////////
// Engineer:   jt.mcguire
//
// Module Name: tvInControl2
// Description: receives NTSC data from ADV7185, extracts oizel information
//
//////////////////////////////////////////////////////////////////////////////////

module tvInControl2( in_clk, clk, rst, lum_in, chrom_in,
in_ready, in_done, in_x, in_y, data_in, ovf, newFrame);

// NOTES:
//    Assuming 16 bit CCIR656 4:2:2 input at clock rate LLC2
//          This clock rate is 13.5 MHz
//          System clock rate is about 160 MHz
//          field_ord should be connected to a physical switch

parameter LAST_X = 623;

// Luminance, chrominance inputs (8 bits each)
input [7:0] lum_in, chrom_in;
// Clocking and reset
input in_clk, clk, rst;
// Output ready-to-store signal
output reg in_ready;
// Store complete signal
input in_done;
// clk synced x and y outputs
output [8:0] in_x, in_y;
// clk synced data output
output reg [35:0] data_in;
// Overflow indicator
```

```
output reg ovf;
output newFrame;

// Input shift registers
reg [7:0] lum_in0, lum_in1, lum_in2, Cb0, Cr0, Cb2;
// Sums of successive color signals
wire [8:0] Cb_sum, Cr_sum;
// New input flag
reg newIn;
// Input state storage
reg state;
// Pixel x and y values
reg [8:0] y;
reg [9:0] x;
// F,V,active
reg F,V,V0,active,active0;

assign newFrame = F;

// 625 line video, but last line will be truncated because
// it is not even, so 624 lines of output (0-623)

// Sum of last and current color values
assign Cb_sum = Cb0 + Cb2;
assign Cr_sum = Cr0 + chrom_in;

// in_clk synced logic
always @(posedge in_clk) begin
	// Luminance shift registers
	lum_in2<=lum_in;
	lum_in1<=lum_in2;
	lum_in0<=lum_in1;
	// Chrominance shift registers
	if(x==(LAST_X-1))begin
		Cb2<=Cr0;
	end else begin
		Cb2<=chrom_in;
	end
	Cr0<=Cb2;
	Cb0<=Cr0;

	// Data code detected on three clk old lines
	if( lum_in0==8'h00 && Cb0==8'hFF )begin
		// Save F, V, active signals
		F<=lum_in1[6];
		V<=lum_in1[5];
		active<=~lum_in1[4];
	end else begin
		// Recapture
		F<=F;
		V<=V;
		active<=active;
```

```
	end

	// Delay
	active0<=active;
	V0<=V;
	// F high, y to 1;  F low, y to 0
	// Falling edge on active signal
	if(rst) begin
		x<=10'b0; y<=9'b0; newIn<=1'b0;
	end else if(V0 && !V)begin
		if(F)begin
			y<=9'b1;
		end else begin
			y<=9'b0;
		end
		x<=10'b0;  newIn<=1'b0;
	end else if(!active && active0)begin
		// Zero x
		x<=10'b0;
		y<=y+2;
		newIn<=1'b0;
	// Active signal high
	end else if(active)begin
		// Increment x, recap y
		x<=x+1; y<=y;
		newIn <= ~x[0] && (x<=LAST_X) && ~V;
	end else begin
		// Zero x, recap y
		x<=10'b0; y<=y; newIn<=1'b0;
	end
end

// clk synced edge detector on trigger
reg newIn0;

// Reassign output
assign in_x = x[9:1];
assign in_y = y;

// clk synced logic
always @(posedge clk) begin
	if(rst)begin
		// Zero everything
		newIn0<=1'b0;
		in_ready<=1'b0;
		ovf<=1'b0;
		data_in<=36'b0;
	end else begin
		// Old value saves
		newIn0<=newIn;

		// Positive edge on newIn
```

```
            if(newIn && !newIn0)begin
                  // in_ready register still high
                  if(in_ready)begin
                        // Overflow detected
                        ovf<=1'b1;
                  end else begin
                        // No overflow
                        ovf<=1'b0;
                  end
                  // Raise the input ready line
                  in_ready<=1'b1;
                  // Latch out two pixels of info
                  data_in<={ lum_in0, Cr0[7:3], Cb0[7:3], lum_in1,
Cr_sum[8:4], Cb_sum[8:4] };
            end else begin
                  // Recapture output and overflow
                  data_in<=data_in;
                  ovf<=ovf;
                  // Input save completed
                  if(in_done)begin
                        // Zero the ready register
                        in_ready<=1'b0;
                  end else begin
                        // Recapture the ready register
                        in_ready<=in_ready;
                  end
            end
      end
end


endmodule


//////////////////////////////////////////////////////////////////////////////////
// Engineer:   j.abel
//
// Module Name: ycbcr_2_rgb
// Description: convert pixel from YCbCr to RGB color space
//
//////////////////////////////////////////////////////////////////////////////////

module ycbcr_2_rgb(
    input clk_in,
    input [7:0] y_in,
    input [7:0] cb_in,
    input [7:0] cr_in,
    output wire [23:0] rgb_out
    );

    //intermediate signed variables
    wire signed [17:0] y_8, cb_8, cr_8;
```

```
    reg signed [9:0] r_s, g_s, b_s;
    reg [7:0] red, green, blue;

    //input recentering
    assign y_8  = y_in-16;
    assign cb_8 = cb_in-128;
    assign cr_8 = cr_in-128;
    assign rgb_out = {red, green, blue};


    always @ (*) begin
       //clipped assignment
       red   = (r_s[9] == 1) ? 8'd0 :
               ((r_s[8] == 1) && (r_s[9] == 0)) ? 8'd255 : r_s[7:0];
       green = (g_s[9] == 1) ? 8'd0 :
               ((g_s[8] == 1) && (g_s[9] == 0)) ? 8'd255 : g_s[7:0];
       blue  = (b_s[9] == 1) ? 8'd0 :
               ((b_s[8] == 1) && (b_s[9] == 0)) ? 8'd255 : b_s[7:0];
    end

    always @ (posedge clk_in) begin
       //matrix multiplication
       r_s   <= ((298*y_8)+(409*cr_8))>>8;
       g_s   <= ((298*y_8)-(100*cb_8)-(208*cr_8))>>8;
       b_s   <= ((298*y_8)+(516*cb_8))>>8;
    end

endmodule
```