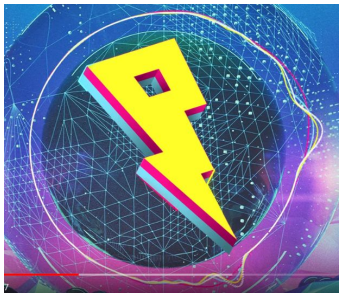
JAW dropping Visual Effects via Audio Spectral Analysis

Julian Chacon, Aaron Wubshet, Wings Yeung

1. Introduction

1.1 Abstract




This project is an audio based visual effect generator. By extracting frequencies and other features from an input audio, effects which alter the input image's color, shape, and position will be generated and superimposed onto the input image to create intriguing visuals. The inspiration comes from music videos that create spectrum analyzers wrapped in a circle through mapping frequencies from an input audio source. The images above demonstrate examples of the general effects we hope to implement.

1.2 Goals

Baseline goals for our system include 3 modules running independently:

- Module 1 (Julian)
    - Color change via pulsation of detected edge
    - Amplitude transformation line algorithm testing in MatLab
    - FFT energy bin calculations
- Module 2 (Aaron)
    - Audio data stored on SD card or sampled via microphone
    - Read audio into BRAM or FIFO
    - Apply FFT on audio without post processing
    - Pass audio through to PWM output of the Nexys4
- Module 3 (Wings)
    - Edge detection code in MATLAB
    - Separate a pixel-wide edge into equal length bins

For the expected goal, we hope to deliver 2 different demonstrations:

1. Color contour transformation on contours defined by edge detection in Verilog
2. Amplitude transformation applied to a predefined square

From these expected demonstrations, our system will include:

- Amplitude transformation line algorithm on FPGA
- Varying edge length for color change transformation
- Audio passed through selectable filters such as low pass, high pass, or bandpass
- FFT output processed to find the magnitude associated with each frequency bin
- Edge detection through Verilog and erosion of edge into a pixel-wide edge
- Camera input for picture

The stretch goals include:

- Simultaneous amplitude and color transformation implemented
- Image negation through amplitude envelope
- Rotation with combination of transformations listed above
- Normal vector and line estimation implemented
- Separation of edges using corner detection
- Edge detection on a live video

1.3 Results

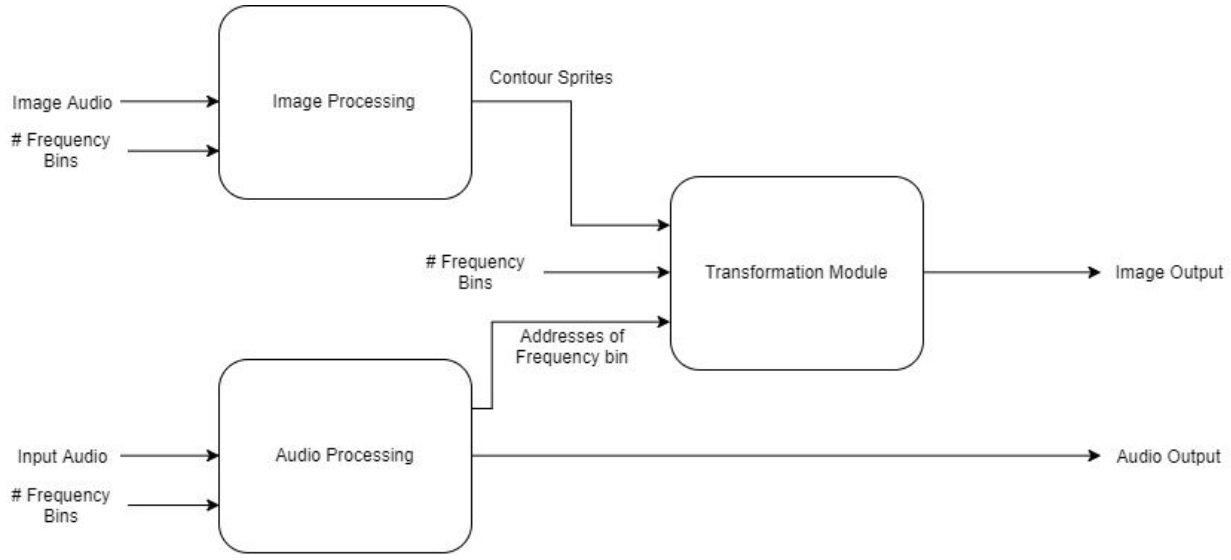
We accomplished all of our baseline goals and most of our expected goals. We are able to demonstrate color transformation based on filtered audio onto contours defined by edge detection. In terms of audio processing, we are able to output the audio loaded onto an SD card, pre-process the audio using multiple selectable filters, and apply FFT onto the incoming audio. For image processing, we are able to detect a pixel-wide edge and separate the edges into equal length sections on either a preloaded image or an image taken from a camera Lastly, for image transformations, we are able to demonstrate color transformations synced with the FFT output onto the separated edge contours.

Of the expected goals, all the modules are working except for the amplitude transformation line algorithm, and we will describe the challenges with this module in the Challenges section below. The verilog for the completed portions can be found in the Appendix.

2. Design and Dataflow

2.1 Block Diagram

There are three major modules in the project. The image and the audio needs to be retrieved and processed. Finally, a suitable mapping of transformations must be designed and implemented to the image before outputting the image on the monitor. Aaron will work on audio retrieval and processing. Wings will handle preliminary image processing and edge detection. Julian will use the processed audio and image to implement image transformations based on the filtered audio input mappings. Below is a block diagram of how our modules will interface.



2.2 Dataflow Overview

Each major module is instantiated in the top level. In addition, several IP cores: the clock and 3 BRAMs are instantiated in the top level.

The first BRAM, named frame_buffer, stores the RGB values of each pixel. This BRAM is 12 bits wide and 307200 deep. The VGA screen is 640 x 480, so each address in the BRAM refers to a single pixel. At each address, there's a 12 bit RGB value associated with

that pixel, with 4 red bits, 4 green bits, and 4 blue bits. This BRAM is used in image preprocessing.

The second BRAM, named xy_edge, stores the segment value of the edge. This BRAM is 3 bits wide, allowing for 7 different segments, and 307200 deep. Like the frame_buffer BRAM, every address in the BRAMM refers to a single pixel. This BRAM is shared between image preprcessing and image transformation module.

The third BRAM, named fft_amplitude_buffer, stores FFT amplitude values. The raw FFT output produces a complex valued number which after som processing turns into a magnitude value that is 16 bits wide and 1024 bits deep (with a window size of 1024 on the FFT). After the FFT completes writing to this BRAM, the mapping and transformations module takes over control.

2.3 Detail

The image module will read an image, either through a preloaded image or a frame taken by the camera. The image is saved in the frame_buffer BRAM. A preloaded image is inputted into the BRAM through a COE file loaded into the init file of the BRAM. If the user wants to use the camera, they first initialize the camera using the center button and then captures a frame using the right button. The information from the camera will be loaded into the BRAM, and therefore it overwrites the preloaded image. Afterwards, the image module extracts a pixel-wide edge and writes into the xy_edge BRAM. After the done signal is toggled by the user, the audio module and the image transformation modules will start.

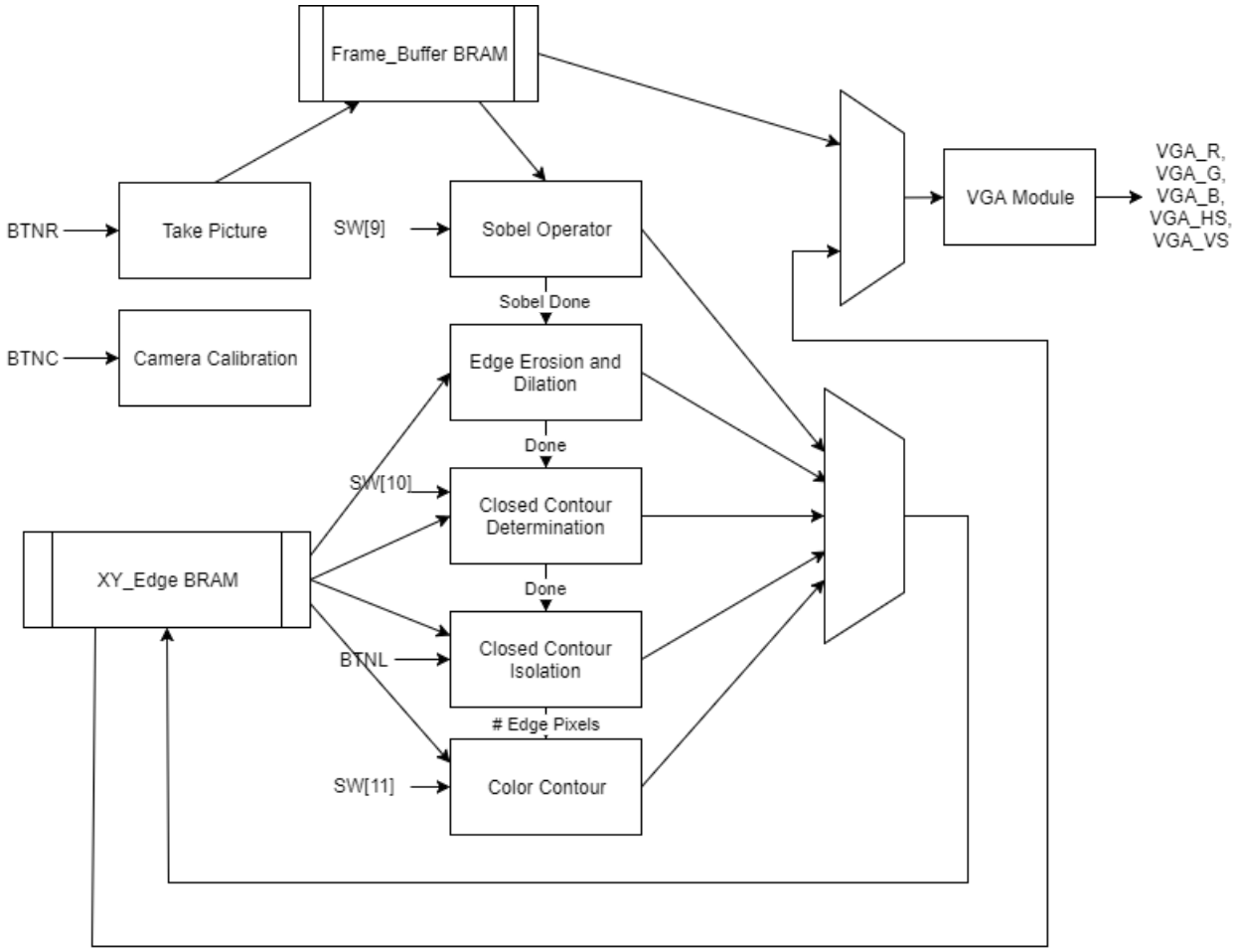
The audio processing module waits for the done signal from the image processing module and then reads in an audio sample from the SD card that is preloaded with appropriately formatted audio (WAV file, 32 kHz sample rate, 8 bit unsigned). The audio is then filtered and analyzed using an FFT. The filters are selected by the user (high pass, low pass, bandpass, or passthrough) using the switches. The FFT output is normalized and the magnitude is found before being written to the fft_amplitude_buffer. Each time (or nearly each time depending on the sensitivity desired) a new window of 1024 FFT values are ready and the audio processing module is no longer writing into the BRAM shared by the image transformation module, a done signal is asserted to indicate that the next mapping of FFT values to image transformations should occur.

HIGH LEVEL IMAGE TRANSFORMATION

3. Image Processing (Wings)

3.1 Block Diagram

`



The camera interface code is taken from Weston's code. The center button calibrates and resets the camera, and the right button takes the picture.

The user indicates that image preprocessing is finished by toggling SW[12], switch 12.

3.2 Sobel Edge Detection

The sobel edge detection modules uses the sobel operator to detect the edges of an image. The inputs into the module are the RGB values for a pixel taken from the

frame_buffer BRAM, and the output is a binary indicator, which will load into the xy_edge BRAM on whether that pixel is an edge. The sobel edge detection module is triggered with the toggling of SW[9], switch 9. When the sobel edge detection module finishes, it outputs a done signal.

The sobel edge detection is a differentiation operator which approximates the gradient of the image using two 3x3 kernels convolved with a black and white version of the original image. The first kernel calculates the horizontal gradient and the second kernel calculates the vertical gradient. We then calculate the magnitude of our gradient using the pythagorean theorem. If the magnitude is greater than a certain threshold, this pixel is an edge. If A is the source image, and Gx and Gy are the horizontal and vertical gradient, the computations are:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

Since the sobel operator use s the grayscale value of the pixel, I converted the pixel's RGB value into grayscale. Using the RGB to YCbCr conversion algorithm, the grayscale of a value is defined as 0.299R + 0.587G + 0.114B, which I approximated as ¼R + ⅝G + ⅛B. At each point, the module keeps a record of 9 grayscale pixel values. These 9 pixels represent a 3x3 square where the center pixel is the one analyzed. This is a rolling 3x3 square, so the module only needs to load 3 new pixels each time. Through testing, I found that the best threshold to obtain a smooth edge is a threshold of 0.

There is a limitation with the sobel operator. The sobel operator cannot distinguish between the edge of the object desired and noise from the camera. Since the camera being used has a low resolution, there are distinguishable rings with any photo taken, which the sobel operator will detect as an edge. Therefore, the image taken from the camera should be on a black surface. This eliminates the rings of the camera, which solves the issue of edges created by camera noise.

3.3 Edge Erosion and Isolation

These several modules reads from and writes to the xy_edge BRAM.

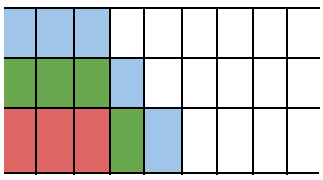
The sobel algorithm outputs a rough outline of the edge; however, the edges have variable thicknesses. However, for the color contour separation step, we need the edge to be a pixel-wide thick. The next step was to implement edge erosion in order to obtain a pixel-wide thick. Normal erosion techniques include classifying a pixel as an edge only if all 8 pixels surrounding it were also classified as an edge in a previous stage. Because the sobel algorithm outputs edges with varying pixel thicknesses, implementing the typical erosion technique will be unreliable and can lead to bumpy edges (if not eroded enough) or edges with gaps (if eroded too much).

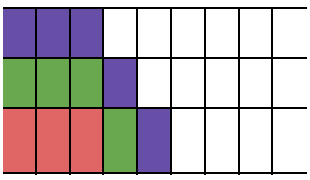
To gain a pixel-width edge, I dilated the edges created by sobel twice, each time updating the xy_edge BRAM. The first dilation creates a choppy pixel-wide and skinnier edge. The input is from the sobel edge, if 1 is read, this is an edge detected by sobel. The module then writes 2 to the pixel above, below, to the left, and to the right of this pixel if that location is 0 (not an edge as detected by sobel). A second dilation is then performed to smooth out the edge. Here, if the module reades a 2, the module writes 3 to the pixel above, below, to the left, and to the right of this pixel if that location is 0. Afterwards, the xy_edge BRAM is populated by 1 if the pixel is an edge detected by sobel, 2 if the pixel is the first dilated edge, 3 if the pixel is the second dilated edge, and 0 if the pixel is not an edge.

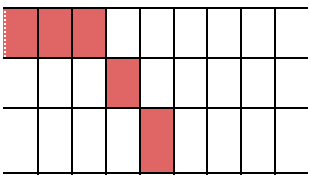
Afterwards, we want to identify a single closed-contour edge. For this, the module scans all the pixels from left to right, top to down, until it detects a pixel labeled 3. This is the start of our closed contour. The then follow this edge, only moving to pixels labeled 3 until we reach the pixel of the starting pixel. At every point, we relabel that pixel as 7. Afterwards, our xy_edge BRAM is populated by 1, 2, 3, 0, and 7, where 7 is the single closed contour edge we desire.

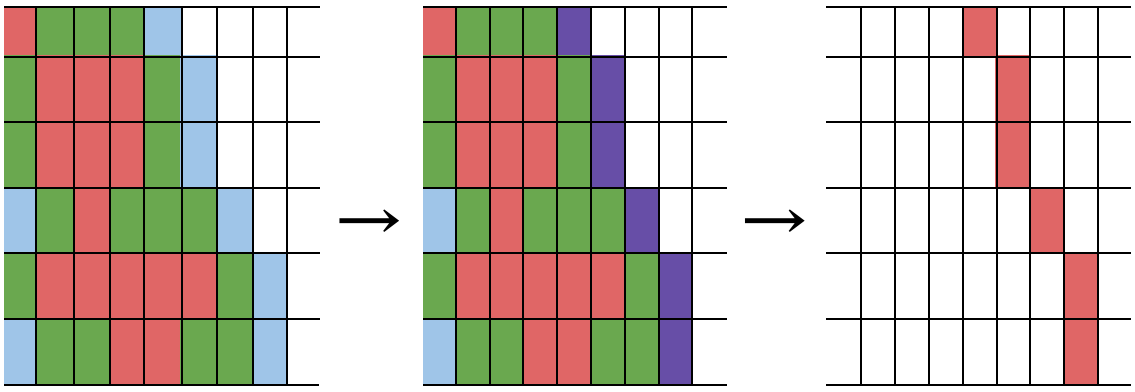
Lastly, we want to isolate this closed contour. So, we scan through the xy_edge BRAM. We rewrite the 7's to 1's, and anything else to 0. The module also outputs a count on how many pixels are in the final closed contour. Now, the xy_edge BRAM is populated only by 0 and 1, where 1 is the closed contour.

Below is an example of the edge erosion and isolation described. In these pictures, a 1 is represented as red, 2 is represented as green, 3 is represented as blue, 7 is represented as purple, and 0 is represented as white. The first is the image of the sobel with 2 dilations. The second is the image of finding a section of the closed contour. Lastly, the third image isolates only one edge in the xy_edge BRAM.









The dilation module is immediately activated by the done signal from the sobel module. Finding and isolating the closed contour is activated by SW[10]. There are some limitations with finding the closed contour. Because the image from a camera is noisy, there are times where a closed contour cannot be found, and therefore, contour isolation never occurs. To mitigate this, if the system was to be stuck on finding a closed contour, the user can press the left button to activate the contour isolation step.

3.4 Color Contour Separation

This module reads from and writes to the xy_edge BRAM.

After finding an isolated contour, the next step is to segment the contour into bins, which will later be matched to frequency bins. Given the count of edge pixels, outputted from the contour isolation module and the number of bins desired, the module uses the division IP core to calculate the number of pixels in each segment.

The color contour separation occurs similarly to finding the single-closed contour module. The module first scans until it finds the beginning of an edge. Then it follows the edge. If that section of the edge belongs to the first segment, a 1 is written to that address of the xy_edge BRAM. If it belongs to the second segment, a 2 is written. This continues all the way to the seventh segment.

3.5 VGA Module

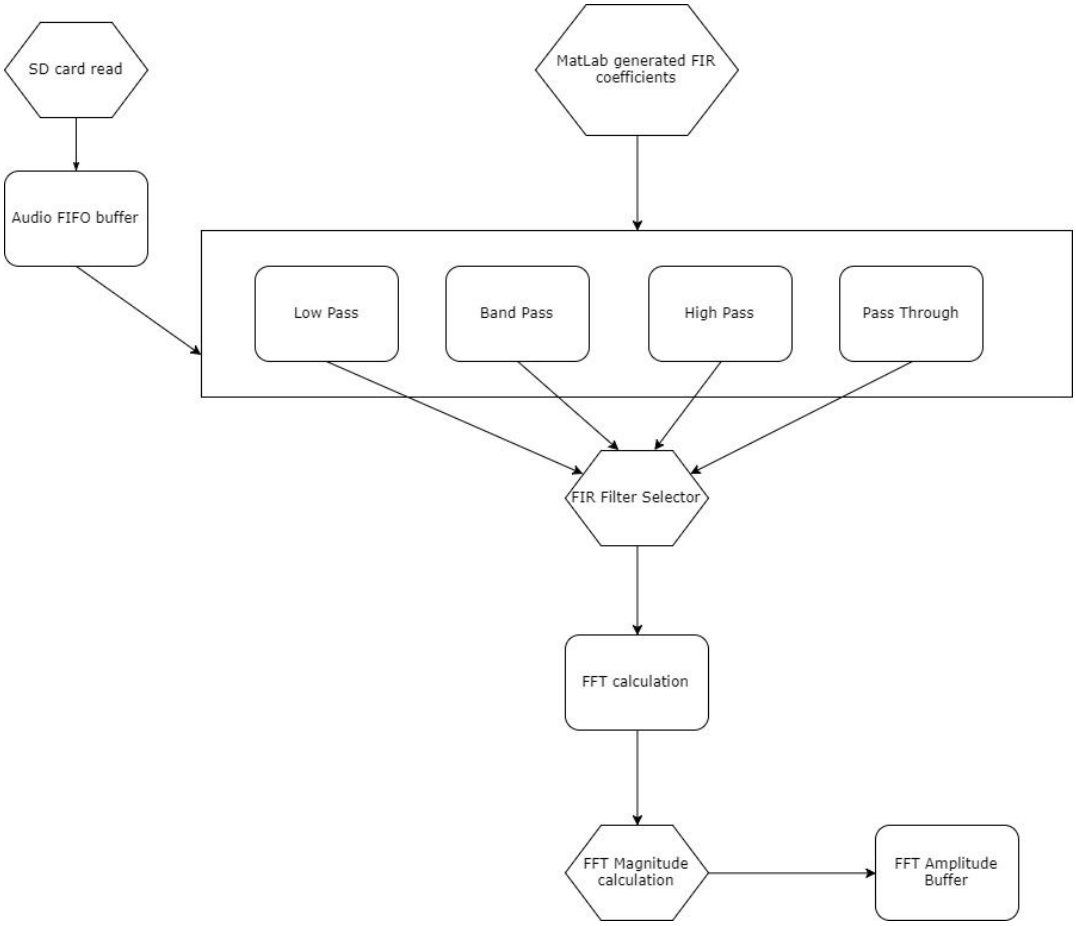
This VGA module is based off of Weston's VGA module in his camera implementation. This module allows for the user to visualize each stage of image processing. If a 0 is written to that pixel in the xy_edge BRAM, the output is the RGB value of the same pixel from the frame_buffer BRAM. Otherwise, the module outputs different,

preset colors, depending on the number written to that pixel in the xy_edge BRAM. For example, 1 is red, 2 is green, and 3 is blue.

4. Audio Processing (Aaron)

4.1 Summary

The audio portion is a three step process: retrieve the audio data, apply the necessary filters, and apply an FFT. Retrieving the audio data occurs via the SD card. After the audio data is accessed by the FPGA, a multitude of filters (low pass, bandpass, and high pass) will be applied to the data to make the mapping of frequencies to transformations as easy as possible. The coefficients for implementing an FIR filters were generated in Matlab and the structure and complexity will be similar to that of lab 5 except with different cutoff frequencies designed to give smooth transitions between bins or to generate interesting frequency effects. Finally, applying an FFT using the IP core will be necessary as the input to the transformations and mappings module.

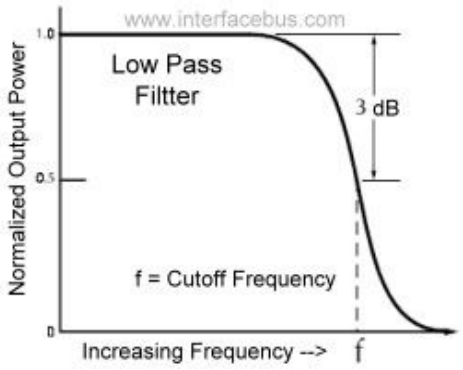


4.2 Data Retrieval

The audio data must be stored on the SD card before running the project in order for success. The SD reading protocol is already implemented in a given verilog file, so the user only has to assert a read at the appropriate time in order to get the data from the SD card. The audio processing required the data be buffered, so I created a FIFO, however a BRAM would have also worked. The advantage of the FIFO is the lack of read latency. With a BRAM there would have been at least a one clock cycle latency between wanting the value and actually having the value. An important consideration when playing audio data from the SD card on the Nexys4 is the data type. The audio must be an 8 bit unsigned WAV file at a 32 kHz sample rate. A program like HxD can be used to put the audio onto the SD card before putting it in the FPGA. The FIFO module I used was provided by Gim and had empty, full, and overflow signals which help with the FSM and control logic flow when using the FIFO as an audio buffer. The way data is read from the SD card is in 512 byte chunks. Thus each read while result in 512 bytes being pushed out from the SD card into the FIFO. After audio samples are in the FIFO the next step is filtering the audio and playing back that filtered audio.

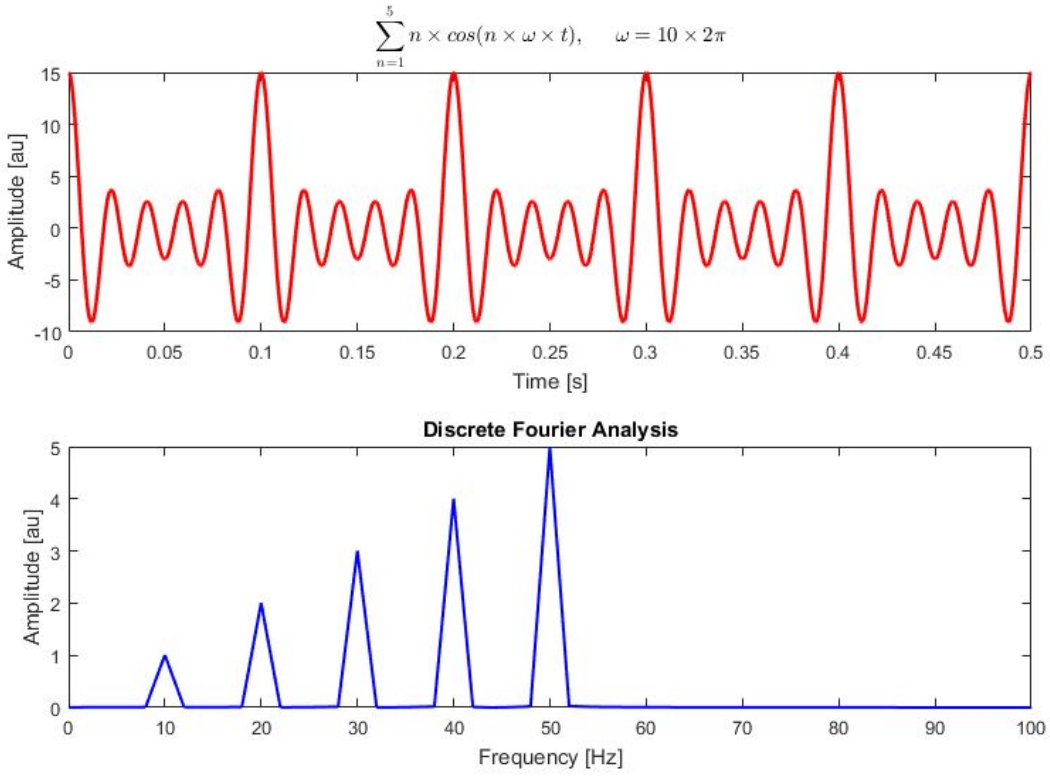
4.3 Audio Filtering and Playback

Once audio samples are in the FIFO buffer the next step in the pipeline is filter and playback. The filters implemented are identical to those used in the lab 5 audio lab. The 31 coefficients for each separate filter (31 tap FIR filter) were generated in MatLab specifying the type and frequency range desired. A total of three filters were used along with one passthrough. The three filters were a low pass, high pass, and bandpass filter. The low pass was at 320 Hz, the high pass at 640 Hz, and the bandpass covering the range in between 320 Hz and 640 Hz. Each sample being sent to a filter comes back 32 clock cycles later since that is the delay of the FIR filter module. The filter output is routed to two different locations. One is the audio playback route via the audioPWM module. The other is an internal BRAM shared with the FFT module. The audioPWM module essentially creates a ramp counter that toggles the audio output everytime it crosses the threshold of the audio sample value at that time step. The FFT module needs a certain number of sample to begin calculating the FFT. In this case the value was set at 1024 with the window shifting by 512 each time. Since a dual port BRAM was used the filter module simply continually incremented the address with no worry about the FFT module. It simply needed to indicate when the first 1024 samples had been filtered and every time the following 512 sample had filtered from then on.



4.4 FFT Implementation

Whenever a new 1024 window of filtered samples are ready (with a shifting window scale of 512 samples) the FFT goes to work. A frame is sent to the FFT IP core which after approximately 2000 clock cycles a value corresponding to each of the 1024 values is ready. These values are complex and must be processed by taking the magnitude of each one before being written into a BRAM shared by the image transformations module.



The purpose of taking the FFT in the first place is to extract useful information about the frequency content of the audio signal being read from the SD card post filtering. This allows for

different bins to be defined by splitting each output of 1024 samples into up to 7 bins which map to different types and intensities of transformations. Addition, multiplication, and square root IP cores are used to take the raw FFT output and generate a useful magnitude to store in the BRAM. However, the FFT changes very quickly which would result in imperceptibly quick changes on the image, and so, depending on the desired sensitivity, the number of frames that are ignored can increase or decrease.

4.5 FSM Considerations

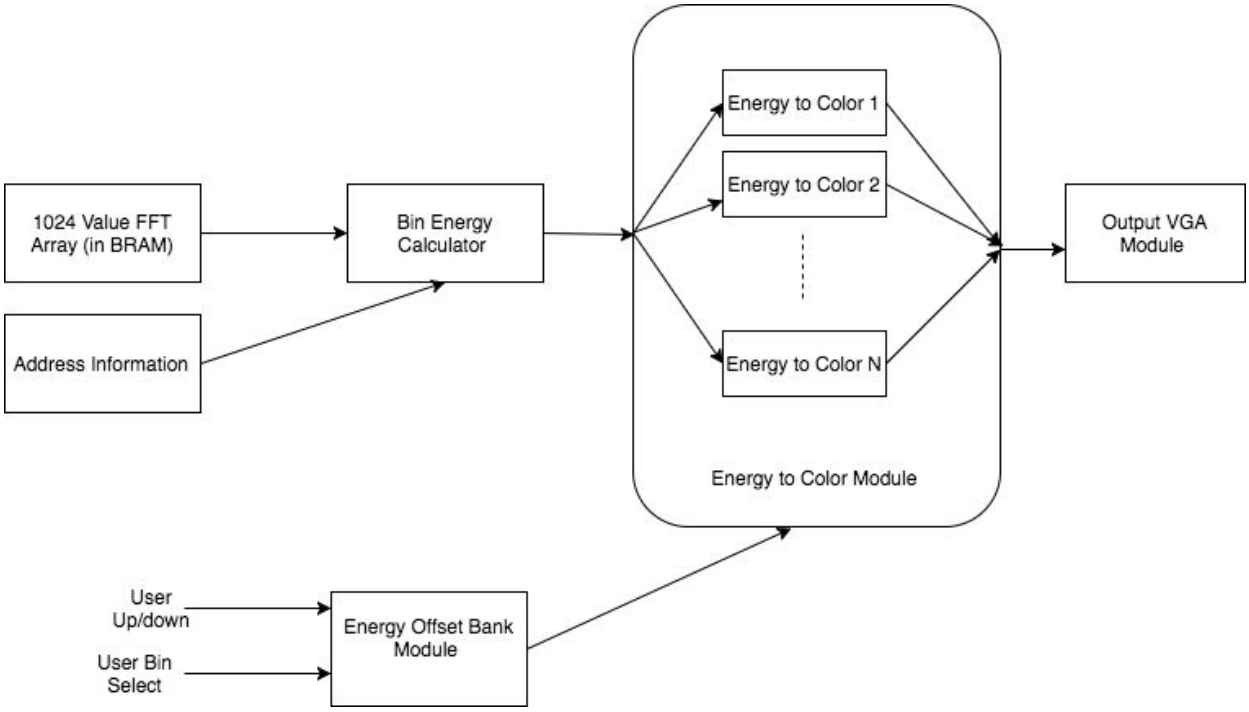
The pipeline is laid out as an FSM that reads from the SD when there is adequate space in the FIFO and reads from the FIFO when the filter ready and the rising edge of the sample rate clock (32 kHz) occurs. After values are filtered they are stored into the BRAM and read out after the appropriate number of samples are in that BRAM. The FSM relies heavily on ready and done signals from the filters, FFT, SD card, and FIFO.

5. Image Transformations (Julian)

5.1 Summary

This portion of the project was focused on combining the FFT spectral analysis of audio, and image processing, to result in interesting and perceivable visual effects in response to music. The focus for the basic project was to use the FFT and apply this spectral analysis in the form of interesting color variations of the image displayed out by the image processing module. In order to get some perceivable changes, the metric that was chosen was the energy in different frequency bins of the resulting FFT. The energy calculated per frequency bin, per window of audio data, would then be mapped to a value that would be used to display a change in color on the output image. A block diagram of this is described below.

Block Diagram



5.2 FFT Energy Motivation

The reason for the choice of energy as the metric for this color transformation was the thought that the energy might be a more stable value over the course of consecutive FFT calculations, than something like the average value of the FFT for those same windows. The equation used for this calculation is derived from Parseval's relation between the energy in a signal, with the squared magnitude of its corresponding DFT, shown below.

$$\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2$$

x[n] is our windowed time audio signal and X[k] are FFT samples.

Since what was used in our implementation was a time-dependent DFT (using FFT algorithm), then the energy value resulting from the calculation above would change between different FFT calculations. This metric was split up between the different frequency bins to represent changes in instrumentation of a song. An example would be to calculate the DFT

“energy” in a certain frequency band where a recognizable instrument’s frequency resides in. For example, this could be the kick drum in a song. Typically the kick drum in a song has most of its energy residing in the frequency band from 100 Hz - 300 HZ. Since the FFT represents samples of the Discrete Time Fourier Transform, then we would want to get the energy in this frequency band by choosing the correct samples representing these frequencies*. After properly choosing the samples of the FFT that represent this frequency band, we would then expect large changes in the energy between places in the audio where the kick drum was and wasn’t present. These large changes in FFT energy would then be visibly seen on the VGA display, as a large change in color intensity/variation for that specific edge.

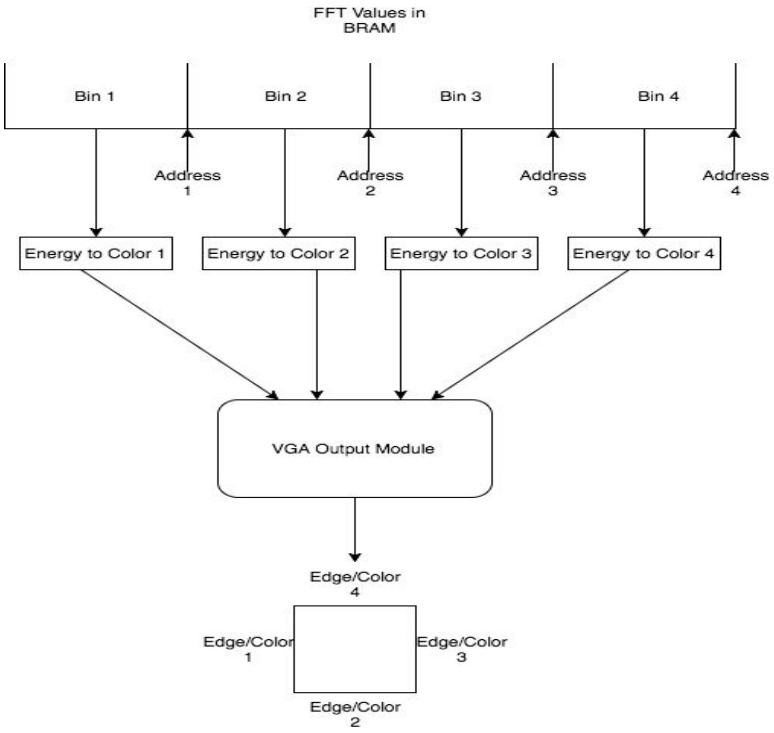
*Note - Half of the 1024 FFT array returned represents the frequencies ranging from 0 to half our sampling frequency, so only using half of this array would have been meaningful, since the other half represents the same values due to the fact that the incoming audio signal is real. This is simply a result of some of the properties of the Fourier transform for a real signal.

5.3 - FFT Energy Calculating and Mapping

The inputs to this module are the FFT values for the current window of audio (in a BRAM), and the addresses for this BRAM that correspond to the different frequency bins. Using these two, the module would take the first address, and take in FFT values from the BRAM, square them, and add them to an accumulated value. Once the module reaches the the first address, it would then pass this accumulated value to the color mapping module for the first color, reset the accumulated value, and repeat the process until it reached the second address and so on. This would produce energy values, per frequency bin, to then be mapped to a change in color, taken care of by the mapping module.

The mapping modules take the energy value and convert this over to a perceivable change in color. Given that we had planned for 7 different frequency bins, we would want up to 7 portions of the displayed edge to vary in color according to the incoming audio signal. We considered that the most perceivable changes in color, would be changes such as a lighter version of the color, to darker, or at least some change that would not be extremely drastic. Due to constraints in the 12 bit color of the Nexys 4 VGA module, and the need for 7 different color variations that were somewhat relatable, this meant that every bin’s energy value would then be mapped to a 4 bit value. The way this was implemented was to take 4 consecutive bits of the energy value, with an offset choosing which 4 consecutive bits this would be per color bin. This 4 bit value would then be used to increase or decrease RGB values per bin, in such a way that there was a perceivable color change. The offset was made to be reprogrammable by the user, even at runtime. The reason for this was that we were unsure what the energy values per bin would be and only taking 4 consecutive bits of the energy value might mean loss of information, such that no color changes are perceived. We would then want to change which 4 bits of the energy value are used for color changes, without having to recompile each time.

After each FFT calculation, the corresponding color per frequency bin would then be output to the VGA module. The VGA module then uses the xy-edge bram from the image processing module as a lookup table of sorts. Whenever the VGA module sees a number in the pixel index in the BRAM, it uses that number to choose one of the seven colors output by the color transformation module. The seven colors change per FFT calculation, and so the colors output on each portion of the displayed edge also change per FFT calculation. I show an example of this process for the case of a square (4 frequency bins), for a specific FFT calculation.



6. Challenges

The major source of uncertainty is the unbounded difficulty of the image modulation in a smooth manner. Color change and coarse image alterations seem manageable, but creating smooth transitions of a sprite based on the shape of an FFT may be too computationally intensive of the FPGA(and us). The main concern here are distortions due to resolution issues on the edge detection on the original image. In theory these transformations would look nice, but it will be difficult to tell until we reach that step.

Timing is also a big concern in terms of sampling audio and outputting to the VGA display correctly. This includes synchronization issues, as well as issues with time averaging the FFT in order to have changes that are perceivable on the VGA display.

Issues with the audio processing arose from a couple sources. The main theme however was that you should always declare your wire and register and make sure they are declared with right bit width. However, the first specific issue was the SD card. Understanding how to properly read data for audio playback was very difficult due to a difference between the PowerPoint slide describing how to use the SD card module and comments in the SD card module verilog file. The appropriate way is to wait for the SD card to assert a ready signal high and then respond by assert the read signal. This will produce 512 bytes of audio data on each rising edge of the byte available port which must be read into FIFO. The fact that the audio must be at a 32 kHz sample is mysterious requirement when loading the SD card from a computer. The next issue was using the FFT IP core using the block diagram utility. The learning curve for using the IP cores and block diagram was much steeper than anticipated. The final major issue was make sure that a done signal was sent to the image transformation module whenever a new frame of FFT data was ready. However, it turned out that the best results actually occured when FFT frames were only changed every 5-10 times new FFT data is ready.

There was also an issue with the amplitude transformation portion of the project. Our expected goal was to be able to have each portion of the displayed edge to present a spectrum analyzer for its corresponding frequency bin. There was a lot of work done towards having an individual module that could do this, assuming only one FFT calculation and no color modulation, but integration of the modules proved to be too difficult towards the end, given our time constraints. The plan was to have a frame read/write buffer, where a module would write FFT amplitude values to the one buffer, to modulate the displayed edge, while the VGA module read from the other buffer. At the end of each frame the buffers would be toggled between the FFT amplitude and VGA modules. This would then need to also synchronize with the output of the FFT processing module, which was the main issue faced towards the end. These synchronization issues lead to missed frames, or completely static frames. As such, we decided to place our energy towards ensuring that our color transformation module worked, and also towards improving it as well.

7. Conclusion

A common theme when creating and debugging this project was that declaring wires and registers (at the appropriate bit width) in between modules is crucial to ensuring that things work when integrated into a final project. While the final product did not include images moving or amplitudes being mapped around an image in response to the FFT, the color change module did work as expected with some fine tuning. The rate at which the FFT values changed simply needed to be slowed down.

An important lesson we learned came from wrestling with integration of our separate modules. We all had working modules for the color transformation early on, but integrating proved to be difficult due to scheduling issues between the three of us. We learned that it's almost detrimental to work on integration without all three members, since on multiple occasions, integration was attempted without the full team and resulted in hours of debugging that were easily fixed when the entire team was present. This was more of a logistical issue on our part, but still gives us valuable lessons for group projects going forward.

8. Appendix of Verilog

```
`timescale 1ns / 1ps

// Audio PWM module.


module audio_PWM(
    input clk,                          // 100MHz clock.
    input reset,                // Reset assertion.
    input [7:0] music_data,       // 8-bit music sample
    output reg PWM_out                      // PWM output. Connect this to ampPWM.
    );




    reg [7:0] pwm_counter = 8'd0;           // counts up to 255 clock cycles per pwm period



    always @(posedge clk) begin
        if(reset) begin
            pwm_counter <= 0;
```

```
            PWM_out <= 0;

        end
        else begin

            pwm_counter <= pwm_counter + 1;

            if(pwm_counter >= music_data)
            begin
            PWM_out <= 0;
            end
            else
            begin
                PWM_out <= 1;
            end
        end
    end
endmodule
// 320 to 640 hz bandpass filter

module bpf1_coeffs(
  input wire [4:0] index,
  output reg signed [9:0] coeff
);
  // tools will turn this into a 31x10 ROM
```

```
always @(index)
  case (index)
    5'd0:  coeff = 10'sd1;
    5'd1:  coeff = 10'sd2;
    5'd2:  coeff = 10'sd3;
    5'd3:  coeff = 10'sd6;
    5'd4:  coeff = 10'sd10;
    5'd5:  coeff = 10'sd15;
    5'd6:  coeff = 10'sd22;
    5'd7:  coeff = 10'sd30;
    5'd8:  coeff = 10'sd39;
    5'd9:  coeff = 10'sd48;
    5'd10: coeff = 10'sd58;
    5'd11: coeff = 10'sd66;
    5'd12: coeff = 10'sd74;
    5'd13: coeff = 10'sd79;
    5'd14: coeff = 10'sd83;
    5'd15: coeff = 10'sd84;
    5'd16: coeff = 10'sd83;
    5'd17: coeff = 10'sd79;
    5'd18: coeff = 10'sd74;
    5'd19: coeff = 10'sd66;
    5'd20: coeff = 10'sd58;
    5'd21: coeff = 10'sd48;
    5'd22: coeff = 10'sd39;
```

```
      5'd23: coeff = 10'sd30;
      5'd24: coeff = 10'sd22;
      5'd25: coeff = 10'sd16;
      5'd26: coeff = 10'sd10;
      5'd27: coeff = 10'sd6;
      5'd28: coeff = 10'sd3;
      5'd29: coeff = 10'sd2;
      5'd30: coeff = 10'sd1;
      default: coeff = 10'hXXX;
    endcase
endmodule
// 500 to 1000 hz bandpass filter

module bpf2_coeffs(
  input wire [4:0] index,
  output reg signed [9:0] coeff
);
  // tools will turn this into a 31x10 ROM
  always @(index)
    case (index)
      5'd0:  coeff = -10'sd5;
      5'd1:  coeff = -10'sd4;
      5'd2:  coeff = -10'sd4;
      5'd3:  coeff = -10'sd3;
      5'd4:  coeff = -10'sd1;
```

```
5'd5:  coeff = 10'sd4;
5'd6:  coeff = 10'sd11;
5'd7:  coeff = 10'sd21;
5'd8:  coeff = 10'sd33;
5'd9:  coeff = 10'sd47;
5'd10: coeff = 10'sd62;
5'd11: coeff = 10'sd77;
5'd12: coeff = 10'sd90;
5'd13: coeff = 10'sd100;
5'd14: coeff = 10'sd106;
5'd15: coeff = 10'sd109;
5'd16: coeff = 10'sd106;
5'd17: coeff = 10'sd100;
5'd18: coeff = 10'sd90;
5'd19: coeff = 10'sd77;
5'd20: coeff = 10'sd62;
5'd21: coeff = 10'sd47;
5'd22: coeff = 10'sd33;
5'd23: coeff = 10'sd21;
5'd24: coeff = 10'sd11;
5'd25: coeff = 10'sd4;
5'd26: coeff = -10'sd1;
5'd27: coeff = -10'sd3;
5'd28: coeff = -10'sd4;
5'd29: coeff = -10'sd4;
```

```
      5'd30: coeff = -10'sd5;
      default: coeff = 10'hXXX;
    endcase
endmodule
`timescale 1ns / 1ps


module camera_address_gen(
    input wire camera_clk,
    input wire camera_pixel_valid,
    input wire camera_frame_done,
    input wire capture_frame,
    input wire [15:0] camera_pixel,
    output reg [11:0] memory_data,
    output wire [18:0] memory_addr,
    output reg memory_we
    );


    parameter VCOUNT_MAX = 479;
    parameter HCOUNT_MAX = 639;

    reg [11:0] vcount = 0;
    reg [11:0] hcount = 0;
    reg capture_frame_latched = 0;
```

```
    assign memory_addr = hcount + vcount * (HCOUNT_MAX+1);


    always@(posedge camera_clk) begin


        capture_frame_latched <= capture_frame ? 1 : camera_frame_done ? 0 :
capture_frame_latched;

        if(camera_frame_done) begin //set frame done

            vcount <= 0;

            hcount <= 0;

            memory_we <= 0;

        end


        else begin

            hcount <= camera_pixel_valid ? (hcount >= HCOUNT_MAX) ? 0 : hcount + 1 : hcount;

            vcount <= camera_pixel_valid & (hcount >= HCOUNT_MAX) ? vcount + 1 : vcount;

            memory_we <= capture_frame_latched ? camera_pixel_valid : 0;

            memory_data <= {camera_pixel[15:12], camera_pixel[10:7], camera_pixel[4:1]};
//convert camera RGB:565 to RGB:444

        end

    end

endmodule

`timescale 1ns / 1ps


module camera_configure

    #(
```

```
parameter CLK_FREQ=25000000
)
(
input wire clk,
input wire start,
output wire sioc,
output wire siod,
output wire done
);

wire [7:0] rom_addr;
wire [15:0] rom_dout;
wire [7:0] SCCB_addr;
wire [7:0] SCCB_data;
wire SCCB_start;
wire SCCB_ready;
wire SCCB_SIOC_oe;
wire SCCB_SIOD_oe;

assign sioc = SCCB_SIOC_oe ? 1'b0 : 1'bZ;
assign siod = SCCB_SIOD_oe ? 1'b0 : 1'bZ;

OV7670_config_rom rom1(
    .clk(clk),
    .addr(rom_addr),
```

```
    .dout(rom_dout)
    );

OV7670_config #(.CLK_FREQ(CLK_FREQ)) config_1(
    .clk(clk),
    .SCCB_interface_ready(SCCB_ready),
    .rom_data(rom_dout),
    .start(start),
    .rom_addr(rom_addr),
    .done(done),
    .SCCB_interface_addr(SCCB_addr),
    .SCCB_interface_data(SCCB_data),
    .SCCB_interface_start(SCCB_start)
    );

SCCB_interface #( .CLK_FREQ(CLK_FREQ)) SCCB1(
    .clk(clk),
    .start(SCCB_start),
    .address(SCCB_addr),
    .data(SCCB_data),
    .ready(SCCB_ready),
    .SIOC_oe(SCCB_SIOC_oe),
    .SIOD_oe(SCCB_SIOD_oe)
    );
```

```
endmodule
`timescale 1ns / 1ps

module camera_read(
        input wire p_clock,
        input wire vsync,
        input wire href,
        input wire [7:0] p_data,
        output reg [15:0] pixel_data =0,
        output reg pixel_valid = 0,
        output reg frame_done = 0

//      inout wire [1:0] SD_DAT,
//      output wire SD_CMD, SD_SCK
    );


//      //SD card memory
//      SD_card_mem SD_card_mem_2 (
//          .cs(SD_DAT[3]),// Connect to SD_DAT[3].
//          .mosi(SD_CMD), // Connect to SD_CMD.
//          .miso(SD_DAT[0]),// Connect to SD_DAT[0].
//          .sclk(SD_SCK), // Connect to SD_SCK.
////                          For SPI mode, SD_DAT[2] and SD_DAT[1] should be held HIGH.
////                          SD_RESET should be held LOW.
```

```
//          .rd(), // Read-enable. When [ready] is HIGH, asseting [rd] will
//                              // begin a 512-byte READ operation at [address].
//                              // [byte_available] will transition HIGH as a new byte has been
//                              // read from the SD card. The byte is presented on [dout].
//          .dout(), // Data output for READ operation.
//          .byte_available(), // A new byte has been presented on [dout].
//          .wr(1'b1), // Write-enable. When [ready] is HIGH, asserting [wr] will
//                              // begin a 512-byte WRITE operation at [address].
//                              // [ready_for_next_byte] will transition HIGH to request that
//                              // the next byte to be written should be presentaed on [din].
//          .din(p_data), // Data input for WRITE operation.
//          .ready_for_next_byte(), // A new byte should be presented on [din].
//          .reset(1'b0),// Resets controller on assertion.
//          .ready(), // HIGH if the SD card is ready for a read or write operation.
//          .address(addr), // Memory address for read/write operation. This MUST
//                                      // be a multiple of 512 bytes, due to SD sectoring.
//          .clk(p_clock), // 25 MHz clock.
//          .status() // For debug purposes: Current state of controller.

//        );
          reg [31:0] addr = 0;   //



          reg [1:0] FSM_state = 0;
```

```
reg pixel_half = 0;

        localparam WAIT_FRAME_START = 0;
        localparam ROW_CAPTURE = 1;




        always@(posedge p_clock) begin

     case(FSM_state)

        WAIT_FRAME_START: begin //wait for VSYNC
          FSM_state <= (!vsync) ? ROW_CAPTURE : WAIT_FRAME_START;
          frame_done <= 0;
          pixel_half <= 0;

          addr <= 0; //
        end

        ROW_CAPTURE: begin
          FSM_state <= vsync ? WAIT_FRAME_START : ROW_CAPTURE;
          frame_done <= vsync ? 1 : 0;
          pixel_valid <= (href && pixel_half) ? 1 : 0;
          if (href) begin
              pixel_half <= ~ pixel_half;
```

```
                if (pixel_half) pixel_data[7:0] <= p_data;
                else pixel_data[15:8] <= p_data;
            end

            addr <= addr + 1; //
        end


    endcase
        end

endmodule
module clock_divider(input clk_in, input [31:0] divider, output reg clk_out = 0, input reset);
    // divider const max = 2^32 - 1

    reg [31:0] counter = 0;

    always @(posedge clk_in) begin
        if (reset == 1) begin
            counter <= 0;
            clk_out <= 0;
        end else if (counter == divider - 1) begin
            counter <= 0;
            clk_out <= ~clk_out;
        end else begin
```

```
            counter <= counter + 1;
            clk_out <= clk_out;
        end

    end // always @

Endmodule
`timescale 1ns / 1ps



module color_contour(
    input wire clk,
    input wire [11:0] num_pixels,
    input wire [2:0] num_bins,
    output reg done, //AARON CHANGED THIS TO NOT HAVE THE EQUAL SIGN
    input wire start,

    //writing to the BRAM of the edges
    input wire [2:0] bram_read,
    output reg [2:0] bram_write,
    output reg [18:0] edge_addr_read,
    output reg [18:0] edge_addr_write
    );
```

```
    initial done = 0; //AARON ADDED THIS LINE SO THAT DONE IS ONLY INITLALY 0 BUT EVENTUALLY COULD GO UP TO 1

    parameter WIDTH = 640;

    parameter HEIGHT = 480;



    parameter STATE_SETUP = 0;

    parameter STATE_WAIT = 1;

    parameter STATE_EXPLORE = 2;

    parameter STATE_IS_IT_EDGE = 3;

    parameter STATE_WAIT_TWO = 4;

    parameter STATE_GET_FIRST = 6;


    reg [2:0] explore_dir = 0;

    reg [2:0] max_explore_dir = 3'b111;

    parameter DIR_GET_RIGHT = 0;

    parameter DIR_GET_DOWNRIGHT = 1;

    parameter DIR_GET_DOWN = 2;

    parameter DIR_GET_DOWNLEFT = 3;

    parameter DIR_GET_LEFT = 4;

    parameter DIR_GET_UPLEFT = 5;

    parameter DIR_GET_UP = 6;

    parameter DIR_GET_UPRIGHT = 7;


    reg [2:0] state = STATE_SETUP;
```

```
reg [2:0] next_state;

reg [18:0] addr_start;
reg [11:0] pixel_count;
reg [2:0] bin_in;



reg [9:0] x_prev = 0;
reg [8:0] y_prev = 0;
reg [18:0] addr_prev = 0;
reg [9:0] x_curr;
reg [8:0] y_curr;
reg [18:0] addr_curr = 0;

reg [9:0] x_explore;
reg [8:0] y_explore;
reg [18:0] addr_explore = 0;

reg [11:0] pixel_per_bin;
wire [23:0] divide_out;
wire divide_valid;
reg divisor_ready;
reg dividend_ready;
reg [7:0] divisor_in;
reg [15:0] dividend_in;
```

```
reg [2:0] count;




div_gen_0 your_instance_name (
  .aclk(clk),                                  // input wire aclk
  .s_axis_divisor_tvalid(divisor_ready),    // input wire s_axis_divisor_tvalid
  .s_axis_divisor_tdata(divisor_in),      // input wire [7 : 0] s_axis_divisor_tdata
  .s_axis_dividend_tvalid(dividend_ready),  // input wire s_axis_dividend_tvalid
  .s_axis_dividend_tdata(dividend_in),    // input wire [15 : 0] s_axis_dividend_tdata
  .m_axis_dout_tvalid(divide_valid),          // output wire m_axis_dout_tvalid
  .m_axis_dout_tdata(divide_out)            // output wire [23 : 0] m_axis_dout_tdata
);



reg [11:0] manual_pixel_bin = 12'h162;
reg first = 1;

always @(posedge clk) begin
    if (~start) begin
        state <= STATE_SETUP;
        done <= 0;
        bin_in <= 3'b001;
        pixel_count <= 0;
```

```
            divisor_ready <= 0;
            dividend_ready <= 0;
            count <= 0;
//            num_pixels <= 0;
        end
        else if (~done) begin


        case (state)
            STATE_SETUP: begin
                bram_write <= 3'b000;
                edge_addr_read <= 19'd0; //64000; //0;
                edge_addr_write <= 19'd0; //64000; //0;
                x_curr <= 0;
                y_curr <= 100;

                addr_prev <= 0;

                divisor_ready <= 1;
                dividend_ready <= 1;
                divisor_in <= {5'h00, num_bins};
                dividend_in <= {4'h0, num_pixels};

                pixel_count <= 0;
```

```
            if (divide_valid) begin
                pixel_per_bin <= divide_out[19:8] + 1;

                next_state <= STATE_GET_FIRST;
                state <= STATE_WAIT;
            end


//              next_state <= STATE_GET_FIRST;
//              state <= STATE_WAIT;
        end

        STATE_GET_FIRST: begin
            if (bram_read != 0 && count == 5) begin
                addr_start <= edge_addr_read;

                bram_write <= bin_in;
                edge_addr_write <= edge_addr_read;

                addr_curr <= edge_addr_read;

                state <= STATE_EXPLORE;
            end
            else begin
                if (bram_read != 0) begin
```

```
            count <= count + 1;
        end
        edge_addr_read <= edge_addr_read + 1;
        if (x_curr == WIDTH - 1) begin
            x_curr <= 0;
            y_curr <= y_curr + 1;
        end else begin
            x_curr <= x_curr + 1;
        end

        state <= STATE_WAIT;
        next_state <= STATE_GET_FIRST;

    end
end




STATE_WAIT: begin
    state <= STATE_WAIT_TWO;
end

STATE_WAIT_TWO: begin
    state <= next_state;
end
```

```
STATE_EXPLORE: begin
    next_state <= STATE_IS_IT_EDGE;
    state <= STATE_WAIT;

    case (explore_dir)
        DIR_GET_RIGHT: begin
            if (addr_curr + 1 == addr_prev) begin
                state <= STATE_EXPLORE;
                explore_dir <= explore_dir + 1;
            end else begin
                edge_addr_read <= addr_curr + 1;
            end
        end

        DIR_GET_DOWNRIGHT: begin
            if (addr_curr + 640 + 1 == addr_prev) begin
                state <= STATE_EXPLORE;
                explore_dir <= explore_dir + 1;
            end else begin
                edge_addr_read <= addr_curr + 640 + 1;
            end
        end

        DIR_GET_DOWN: begin
```

```
    if (addr_curr + 640 == addr_prev) begin
        state <= STATE_EXPLORE;
        explore_dir <= explore_dir + 1;
    end else begin
        edge_addr_read <= addr_curr + 640;
    end
end

DIR_GET_DOWNLEFT: begin
    if (addr_curr + 640 - 1 == addr_prev) begin
        state <= STATE_EXPLORE;
        explore_dir <= explore_dir + 1;
    end else begin
        edge_addr_read <= addr_curr + 640 - 1;
    end
end

DIR_GET_LEFT: begin
    if (addr_curr - 1 == addr_prev) begin
        state <= STATE_EXPLORE;
        explore_dir <= explore_dir + 1;
    end else begin
        edge_addr_read <= addr_curr - 1;
    end
end
```

```
DIR_GET_UPLEFT: begin
    if (addr_curr - 640 - 1 == addr_prev) begin
        state <= STATE_EXPLORE;
        explore_dir <= explore_dir + 1;
    end else begin
        edge_addr_read <= addr_curr - 640 - 1;
    end
end

DIR_GET_UP: begin
    if (addr_curr - 640 == addr_prev) begin
        state <= STATE_EXPLORE;
        explore_dir <= explore_dir + 1;
    end else begin
        edge_addr_read <= addr_curr - 640;
    end
end

DIR_GET_UPRIGHT: begin
    if (addr_curr - 640 + 1 == addr_prev) begin
        state <= STATE_EXPLORE;
        explore_dir <= explore_dir + 1;
    end else begin
        edge_addr_read <= addr_curr - 640 + 1;
```

```
            end
        end
    endcase
end

STATE_IS_IT_EDGE: begin
    state <= STATE_EXPLORE;

    if (edge_addr_read == addr_start) begin
        done <= 1;
        state <= STATE_WAIT;
    end
    else begin
        if (bram_read != 0) begin //3'b001) begin

            //update our memory of current and past pixels
            addr_prev <= addr_curr;
            addr_curr <= edge_addr_read;

            explore_dir <= 0;

            //write that bin into the bram location
            edge_addr_write <= edge_addr_read;

            bram_write <= bin_in;
```

```
                pixel_count <= pixel_count + 1;
                if (pixel_count == pixel_per_bin) begin
                    if (bin_in == 3'b111) begin
                        bin_in <= 1;
                    end
                    else begin
                        bin_in <= bin_in + 1;
                    end
                    pixel_count <= 0;
                end
            end
            else begin
                explore_dir <= explore_dir + 1;
                if (explore_dir == max_explore_dir)begin
                    done <= 1;
//                      if (pixel_count < 2) begin
////                        pixel_count <= 0;
////                        edge_addr_read <= edge_addr_read + 2;
//                      state <= STATE_WAIT;
//                      next_state <= STATE_GET_FIRST;
////                        edge_addr_write <= 0;

//                      bram_write <= 3'b000;
//                      edge_addr_read <= edge_addr_read + 2; //0;
```

```
//                          edge_addr_write <= edge_addr_read + 2; //0;


//                          if (x_curr == WIDTH - 1) begin
//                              x_curr <= 0;
//                              y_curr <= y_curr + 1;
//                          end else begin
//                              x_curr <= x_curr + 1;
//                          end
//                      end
//                      else begin
//                          done <= 1;
//                      end
                    end
                end
            end
        end




        endcase
        end
    end
endmodule
`timescale 1ns / 1ps
```

```
module Color_offst(
        input clock,
        input wire reset,
        input wire [15:0] SW,
        input wire up,
        input wire down,
        output wire adjusting,
        output reg [3:0] off1,
        output reg [3:0] off2,
        output reg [3:0] off3,
        output reg [3:0] off4,
        output reg [3:0] off5,
        output reg [3:0] off6,
        output reg [3:0] off7

    );

    always@(posedge clock)begin
        if(reset)begin
            off1 <= 0;
            off2 <= 0;
            off3 <= 0;
            off4 <= 0;
            off5 <= 0;
            off6 <= 0;
```

```
    off7 <= 0;
end

else if(adjusting) begin
    if (SW[0]) begin
        if(up)begin
            if(off1 == 15)begin
                off1 <= off1;
            end
            else begin
                off1 <= off1 + 1;
            end

        end

        else if (down)begin
            if(off1 == 0) begin
                off1 <= off1;
            end
            else begin
                off1 <= off1 - 1;
            end
        end
    end
```

```
if (SW[1]) begin
    if(up)begin
        if(off2 == 15)begin
            off2 <= off2;
        end
        else begin
            off2 <= off2 + 1;
        end

    end

    else if (down)begin
        if(off2 == 0) begin
            off2 <= off2;
        end
        else begin
            off2 <= off2 - 1;
        end
    end
```

```
end

if (SW[2]) begin
    if(up)begin
        if(off3 == 15)begin
            off3 <= off3;
        end
        else begin
            off3 <= off3 + 1;
        end

    end

    else if (down)begin
        if(off3 == 0) begin
            off3 <= off3;
        end
        else begin
            off3 <= off3 - 1;
        end
```

```
    end
end




if (SW[3]) begin
    if(up)begin
        if(off4 == 15)begin
            off4 <= off4;
        end
        else begin
            off4 <= off4 + 1;
        end

    end

    else if (down)begin
        if(off4 == 0) begin
            off4 <= off4;
        end
        else begin
            off4 <= off4 - 1;
        end
    end
```

```
end



if (SW[4]) begin
    if(up)begin
        if(off5 == 15)begin
            off5 <= off5;
        end
        else begin
            off5 <= off5 + 1;
        end

    end

    else if (down)begin
        if(off5 == 0) begin
            off5 <= off5;
        end
        else begin
            off5 <= off5 - 1;
        end
    end
end
```

```
if (SW[5]) begin
    if(up)begin
        if(off6 == 15)begin
            off6 <= off6;
        end
        else begin
            off6 <= off6 + 1;
        end

    end

    else if (down)begin
        if(off6 == 0) begin
            off6 <= off6;
        end
        else begin
            off6 <= off6 - 1;
        end
    end
end


if (SW[6]) begin
    if(up)begin
```

```
            if(off7 == 15)begin
                off7 <= off7;
            end
            else begin
                off7 <= off7 + 1;
            end

        end

        else if (down)begin
            if(off7 == 0) begin
                off7 <= off7;
            end
            else begin
                off7 <= off7 - 1;
            end
        end
    end

end
```

```
    end
assign adjusting = (SW[6:0] == 0) ? 0: 1;
endmodule
`timescale 1ns / 1ps



module Color_output(
    input wire [2:0] bin_data,
    input wire ready,
    input wire [83:0] FFT_COLOR,
    input wire video_clk,
    output wire [18:0] memory_addr,
    output reg vsync,
    output reg hsync,
    output wire [11:0] video_out
    );



    // horizontal: 800 pixels total
    // display 640 pixels per line
    reg hblank,vblank;
    wire hsyncon,hsyncoff,hreset,hblankon;
    reg [11:0] hcount = 0;
```

```
reg [11:0] vcount = 0;
reg blank;
//kludges to fix frame alignment due to memory access time
reg blank_delay;
reg blank_delay_2;
reg hsync_pre_delay;
reg hsync_pre_delay_2;
reg vsync_pre_delay;
reg vsync_pre_delay_2;

reg at_display_area;

/*assign pixel out value based on BRAM data*/

assign video_out = ~at_display_area ? 0:
                        bin_data == 3'b001 ? FFT_COLOR[11:0] :
                        bin_data == 3'b010 ? FFT_COLOR[23:12] :
      bin_data == 3'b011 ? FFT_COLOR[35:24] :
                        bin_data == 3'b100 ? FFT_COLOR[47:36]:
                        bin_data == 3'b101 ? FFT_COLOR[59:48]:
                        bin_data == 3'b110 ? FFT_COLOR[71:60]:
                        bin_data == 3'b111 ? FFT_COLOR[83:72]:
                        bin_data == 3'b000 ? 0: 0;

assign hblankon = (hcount == 639);   //blank after display width
```

```
    assign hsyncon = (hcount == 655);  // active video + front porch
    assign hsyncoff = (hcount == 751); //active video + front portch + sync
    assign hreset = (hcount == 799); //plus back porch

    // vertical:  525 lines total
    // display 480 lines
    wire vsyncon,vsyncoff,vreset,vblankon;
    assign vblankon = hreset & (vcount == 479);
    assign vsyncon = hreset & (vcount == 489);
    assign vsyncoff = hreset & (vcount == 491);
    assign vreset = (hreset & (vcount == 524));

    // sync and blanking
    wire next_hblank,next_vblank;
    assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
    assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;


    assign memory_addr = ready ? hcount+(vcount*640) : 0;                    //don't output address
until can start reading from BRAM

    always @(posedge video_clk) begin
        blank_delay <= blank;
        blank_delay_2 <= blank_delay;
        hsync_pre_delay_2 <= hsync_pre_delay;
```

```
        vsync_pre_delay_2 <= vsync_pre_delay;
        vsync <= vsync_pre_delay_2;
        hsync <= hsync_pre_delay_2;
         //hcount
         hcount <= hreset ? 0 : hcount + 1;
         hblank <= next_hblank;
         hsync_pre_delay <= hsyncon ? 0 : hsyncoff ? 1 : hsync_pre_delay;  // active low

         vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
         vblank <= next_vblank;
         vsync_pre_delay <= vsyncon ? 0 : vsyncoff ? 1 : vsync_pre_delay;  // active low

         blank <= next_vblank | (next_hblank & ~hreset);

         at_display_area <= ((hcount >= 0) && (hcount < 640) && (vcount >= 0) && (vcount <
480));
    end
endmodule
`timescale 1ns / 1ps

module Color_transform(
       input CLK100MHZ,
            input clock_25mhz,
                input reset, //added master reset instead of directly using switch 15
       input[15:0] SW,
```

```
input BTNC, BTNU, BTNL, BTNR, BTND,
output[3:0] VGA_R,
output[3:0] VGA_B,
output[3:0] VGA_G,
output VGA_HS,
output VGA_VS,
output [9:0] addr2_b,
input [15:0] dout2_b,

output [18:0] addr_b,
input [2:0] dout_b,

input wings_done,
input wubs_done,

input [59:0] addresses,
input [2:0] bin_num,

output FFT_done,
output [31:0] data_disp,
output [9:0] julian_test,
output wire start
);
```

```
wire up_puls;
lev_to_pulse pulse_up(.clock(CLK100MHZ),.reset(reset),.lev(BTNU),.pulse(up_puls));

wire down_puls;
lev_to_pulse pulse_down(.clock(CLK100MHZ),.reset(reset),.lev(BTND),.pulse(down_puls));



wire [83:0] color_FFT;
wire [83:0] color = 83'hFFF_FF0_F0F_00F_0F0_D08_0FF;
wire [83:0] color_test;

wire [3:0] off1;
wire [3:0] off2;
wire [3:0] off3;
wire [3:0] off4;
wire [3:0] off5;
wire [3:0] off6;
wire [3:0] off7;
wire [3:0] off8;
wire adjusting;

wire [3:0] c_map_test;

wire [23:0] data_test;
```

```
    Color_offst
test_off(.clock(CLK100MHZ),.reset(SW[14]),.SW(SW),.up(up_puls),.down(down_puls),.adjusting
(adjusting), .off1(off1), .off2(off2),
                    .off3(off3), .off4(off4), .off5(off5), .off6(off6), .off7(off7));


    //add a pause signal somewhere to change the offset???


    FFT_energy
test_FFT(.clock(CLK100MHZ),.ready(wubs_done),.bin_num(bin_num),.addresses(addresses),.
data(dout2_b),.bram_addr(addr2_b),
                    .off1(off1), .off2(off2), .off3(off3), .off4(off4), .off5(off5), .off6(off6), .off7(off7),

.color(color_FFT),.test_out(data_test),.done(FFT_done),.adjusting(adjusting),.reset(reset),.c_ma
p_test(c_map_test), .julian_test(julian_test),
                    .start(start));


    assign color_test = color_FFT;
    //wire [83:0] color = 84'h00F_00C_00A_008_006_004_003;
    wire [11:0] rgb;
    wire hsync, vsync;


    //assign data_disp[7:0] = {off2,off1};
    assign data_disp = {16'b0, off4,off3,off2,off1};
    //assign data_disp[0] = julian_test;
```

```
    Color_output
test_color(.bin_data(dout_b),.ready(wings_done),.FFT_COLOR(color_test),.video_clk(clock_25
mhz),.memory_addr(addr_b),.vsync(vsync), .hsync(hsync),

                        .video_out(rgb));









    ////////////////////////////////////////////////////////////////////////////////

    // sample Verilog to generate color bars

    //    vga vga1(.vga_clock(clock_25mhz),.hcount(hcount),.vcount(vcount),

    //          .hsync(hsync),.vsync(vsync),.at_display_area(at_display_area));


        assign VGA_R = rgb[11:8];

        assign VGA_G = rgb[7:4];

        assign VGA_B = rgb[3:0];


    //    assign VGA_R =  10;

    //    assign VGA_G =  8;

    //    assign VGA_B =  1;

        assign VGA_HS = hsync;

        assign VGA_VS = vsync;



endmodule

// Switch Debounce Module
```

```
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce #(parameter DELAY=1000000)   // .01 sec with a 100Mhz clock
                 (input reset, clock, noisy,
                  output reg clean);

  reg [19:0] count;
  reg new_;

  always @(posedge clock)
    if (reset)
      begin
            count <= 0;
            new_ <= noisy;
            clean <= noisy;
      end
    else if (noisy != new_)
      begin
            new_ <= noisy;
            count <= 0;
      end
    else if (count == DELAY)
      clean <= new_;
    else
      count <= count+1;
```

```
Endmodule
`timescale 1ns / 1ps

module display_8hex(
    input clk,                  // system clock
    input [31:0] data,          // 8 hex numbers, msb first
    output reg [6:0] seg,       // seven segment display output
    output reg [7:0] strobe     // digit strobe
    );

    localparam bits = 13;

    reg [bits:0] counter = 0;  // clear on power up

    wire [6:0] segments[15:0]; // 16 7 bit memorys
    assign segments[0]  = 7'b100_0000;
    assign segments[1]  = 7'b111_1001;
    assign segments[2]  = 7'b010_0100;
    assign segments[3]  = 7'b011_0000;
    assign segments[4]  = 7'b001_1001;
    assign segments[5]  = 7'b001_0010;
    assign segments[6]  = 7'b000_0010;
    assign segments[7]  = 7'b111_1000;
    assign segments[8]  = 7'b000_0000;
```

```
assign segments[9]  = 7'b001_1000;
assign segments[10] = 7'b000_1000;
assign segments[11] = 7'b000_0011;
assign segments[12] = 7'b010_0111;
assign segments[13] = 7'b010_0001;
assign segments[14] = 7'b000_0110;
assign segments[15] = 7'b000_1110;

always @(posedge clk) begin
  counter <= counter + 1;
  case (counter[bits:bits-2])
     3'b000: begin
            seg <= segments[data[31:28]];
            strobe <= 8'b0111_1111 ;
           end

     3'b001: begin
            seg <= segments[data[27:24]];
            strobe <= 8'b1011_1111 ;
           end

     3'b010: begin
             seg <= segments[data[23:20]];
             strobe <= 8'b1101_1111 ;
            end
```

```
3'b011: begin
        seg <= segments[data[19:16]];
        strobe <= 8'b1110_1111;
       end
3'b100: begin
        seg <= segments[data[15:12]];
        strobe <= 8'b1111_0111;
       end

3'b101: begin
        seg <= segments[data[11:8]];
        strobe <= 8'b1111_1011;
       end

3'b110: begin
         seg <= segments[data[7:4]];
         strobe <= 8'b1111_1101;
       end
3'b111: begin
        seg <= segments[data[3:0]];
        strobe <= 8'b1111_1110;
       end
```

```
        endcase
       end

endmodule
`timescale 1ns / 1ps



module edge_to_display(
    input wire [2:0] bin_data,
    input wire video_clk,
    output wire [18:0] memory_addr,
    output reg vsync,
    output reg hsync,
    output wire [11:0] video_out
    );



    // horizontal: 800 pixels total
    // display 640 pixels per line
    reg hblank,vblank;
    wire hsyncon,hsyncoff,hreset,hblankon;
    reg [11:0] hcount = 0;
    reg [11:0] vcount = 0;
    reg blank;
```

```
    //kludges to fix frame alignment due to memory access time
    reg blank_delay;
    reg blank_delay_2;
    reg hsync_pre_delay;
    reg hsync_pre_delay_2;
    reg vsync_pre_delay;
    reg vsync_pre_delay_2;

    reg at_display_area;
//    assign video_out = 12'hF0F; //blank_delay_2 ? 12'b0 : (bin_data != 3'b000) ? 12'hFFF : 12'b0;
//    assign video_out = at_display_area ? 12'hF0f: 0;
//    assign video_out = at_display_area ? ((bin_data != 3'b000) ? 12'h000 : 12'hFFF): 0;//(12'h0F0) : 0;
    assign video_out = ~at_display_area ? 0: bin_data == 3'b001 ? 12'hF00 : bin_data == 3'b010 ? 12'h0F0 :
                       bin_data == 3'b011 ? 12'h00F : bin_data == 3'b100 ? 12'hF0F : 12'hFFF;

    assign hblankon = (hcount == 639);   //blank after display width
    assign hsyncon = (hcount == 655);  // active video + front porch
    assign hsyncoff = (hcount == 751); //active video + front portch + sync
    assign hreset = (hcount == 799); //plus back porch

    // vertical:  525 lines total
    // display 480 lines
    wire vsyncon,vsyncoff,vreset,vblankon;
```

```
assign vblankon = hreset & (vcount == 479);
assign vsyncon = hreset & (vcount == 489);
assign vsyncoff = hreset & (vcount == 491);
assign vreset = (hreset & (vcount == 524));

// sync and blanking
wire next_hblank,next_vblank;
assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;

assign memory_addr = hcount+vcount*640;

always @(posedge video_clk) begin
    blank_delay <= blank;
    blank_delay_2 <= blank_delay;
    hsync_pre_delay_2 <= hsync_pre_delay;
    vsync_pre_delay_2 <= vsync_pre_delay;
    vsync <= vsync_pre_delay_2;
    hsync <= hsync_pre_delay_2;
     //hcount
     hcount <= hreset ? 0 : hcount + 1;
     hblank <= next_hblank;
     hsync_pre_delay <= hsyncon ? 0 : hsyncoff ? 1 : hsync_pre_delay;  // active low

     vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
```

```
        vblank <= next_vblank;
        vsync_pre_delay <= vsyncon ? 0 : vsyncoff ? 1 : vsync_pre_delay;  // active low

        blank <= next_vblank | (next_hblank & ~hreset);

        at_display_area <= ((hcount >= 0) && (hcount < 640) && (vcount >= 0) && (vcount <
480));
    end
endmodule
`timescale 1ns / 1ps


module edge_pixel_width(
    input wire clk,
    input wire start,
    output reg done,

    //writing to the BRAM of the edges
    input wire [2:0] bram_read,
    output reg [2:0] bram_write,
    output reg [18:0] edge_addr_read,
    output reg [18:0] edge_addr_write
    );

    reg [2:0] n = 1;
```

```
reg [3:0] pixel0, pixel1, pixel2, pixel3, pixel4, pixel5, pixel6, pixel7, pixel8;
reg [3:0] pixel_load0, pixel_load1, pixel_load2;

parameter STATE_SETUP = 0;
parameter STATE_WAIT = 1;
parameter STATE_WAIT2 = 2;
parameter STATE_GET_9 = 3;

parameter STATE_SHIFTWINDOW = 4;
parameter STATE_MIDDLE_EDGE = 5;
parameter STATE_UP = 6;
parameter STATE_DOWN = 7;
parameter STATE_RIGHT = 8;
parameter STATE_LEFT = 9;


reg [4:0] state = STATE_SETUP;
reg [4:0] next_state;

parameter WIDTH = 640;
parameter HEIGHT = 480;

reg [9:0] x;
reg [8:0] y;
```

```
reg [18:0] middle_addr;



reg [3:0] i = 0;
reg [5:0] old_SW = 16'hFFFF;



always @(posedge clk) begin
    if (~start) begin
        state <= STATE_SETUP;
        n <= 1;
        done <= 0;
    end
    else if (~done) begin
    case (state)
        STATE_SETUP: begin
            i <= 0;
            x <= 1;
            y <= 1;
            edge_addr_read <= 0;
            middle_addr <= 640 + 1;
            done <= 0;

            if (start) begin
```

```
        state <= STATE_WAIT;
        next_state <= STATE_GET_9;
    end
end

STATE_WAIT: state <= STATE_WAIT2;

STATE_WAIT2: state <= next_state;

STATE_GET_9: begin
    i <= i + 1;
    state <= STATE_WAIT;
    next_state <= STATE_GET_9;

    case (i)
        0: begin
            pixel0 = bram_read;
            edge_addr_read <= edge_addr_read + 1;
        end
        1: begin
            pixel1 = bram_read;
            edge_addr_read <= edge_addr_read + 1;
        end
        2: begin
            pixel2 = bram_read;
```

```
    edge_addr_read <= edge_addr_read + WIDTH - 2;
end
3: begin
    pixel3 = bram_read;
    edge_addr_read <= edge_addr_read + 1;
end
4: begin
    pixel4 = bram_read;
    edge_addr_read <= edge_addr_read + 1;
end
5: begin
    pixel5 = bram_read;
    edge_addr_read <= edge_addr_read + WIDTH - 2;
end
6: begin
    pixel6 = bram_read;
    edge_addr_read <= edge_addr_read + 1;
end
7: begin
    pixel7 = bram_read;
    edge_addr_read <= edge_addr_read + 1;
end
8: begin
    pixel8 = bram_read;
    edge_addr_read <= edge_addr_read - WIDTH - WIDTH + 1;
```

```
            next_state <= STATE_MIDDLE_EDGE;
        end
    endcase
end

STATE_MIDDLE_EDGE: begin    //decides whether or not middle pixel is an edge
    if (pixel4 == n) begin //1) begin
        state <= STATE_UP;
    end else begin
        middle_addr <= middle_addr + 1;
        state <= STATE_SHIFTWINDOW;
        i <= 0;

    end
end

STATE_UP: begin
    if (pixel1 == 0) begin
        bram_write <= n + 1; //3'b010;
        edge_addr_write <= middle_addr - 640;
    end
    state <= STATE_RIGHT;
end

STATE_RIGHT: begin
```

```
    if (pixel5 == 0) begin
        bram_write <= n + 1; //3'b010;
        edge_addr_write <= middle_addr + 1;
    end
    state <= STATE_DOWN;
end

STATE_DOWN: begin
    if (pixel7 == 0) begin
        bram_write <= n + 1; //3'b010;
        edge_addr_write <= middle_addr + 640;
    end
    state <= STATE_LEFT;
end

STATE_LEFT: begin
    if (pixel3 == 0) begin
        bram_write <= n + 1; //3'b010;
        edge_addr_write <= middle_addr - 1;
    end
    middle_addr <= middle_addr + 1;
    state <= STATE_SHIFTWINDOW;
    i <= 0;
end
```

```
STATE_SHIFTWINDOW: begin
    i <= i + 1;
    state <= STATE_WAIT;
    next_state <= STATE_SHIFTWINDOW;

    case(i)
        0: begin
            pixel_load0 = bram_read;
            edge_addr_read <= edge_addr_read + WIDTH;
        end

        1: begin
            pixel_load1 = bram_read;
            edge_addr_read <= edge_addr_read + WIDTH;
        end

        2: begin
            pixel_load2 = bram_read;
            edge_addr_read <= edge_addr_read - WIDTH - WIDTH + 1;
        end

        3: begin
            pixel0 <= pixel1;
```

```
                pixel1 <= pixel2;
                pixel2 <= pixel_load0;
                pixel3 <= pixel4;
                pixel4 <= pixel5;
                pixel5 <= pixel_load1;
                pixel6 <= pixel7;
                pixel7 <= pixel8;
                pixel8 <= pixel_load2;

                if (x == WIDTH - 1) begin
                    x <= 0;
                    y <= y + 1;
                end
                else begin
                    x <= x + 1;
                end

                if (middle_addr >= 307200 - 640 - 640) begin// ( x == WIDTH - 1) && (y ==
HEIGHT - 2)) begin
                    if (n == 2) begin
                        done <= 1;
                    end else begin
                        state <= STATE_SETUP;
                        n <= n + 1;
                    end
```

```
                end else begin
                    state <= STATE_MIDDLE_EDGE;
                end
            end
        endcase
    end

    endcase
    end
  end
endmodule
`timescale 1ns / 1ps


module FFT_energy(
        input clock,
        input ready,
        input reset,
        input wire [2:0] bin_num,
        input wire [59:0] addresses,
        input wire [15:0] data,
        input wire [3:0] off1,
        input wire [3:0] off2,
        input wire [3:0] off3,
        input wire [3:0] off4,
```

```
        input wire [3:0] off5,
        input wire [3:0] off6,
        input wire [3:0] off7,
        input wire adjusting,
        output wire [11:0] acc,
        output reg [9:0] bram_addr,
        output reg done,
        output reg [3:0]  c_map_test,
        output wire [83:0] color,
        output reg [23:0] test_out,     //debugging output
        output reg test,
        output reg  [9:0] julian_test,
        output reg start
    );

reg [59:0] addr_hold;                   //holds
reg [9:0] max_addr;
reg [2:0] bin_count;
reg [2:0] start_count;
reg addr_count;
reg start = 0;
reg [41:0] bin_acc;                 //make sure this can hold our maximum value! - 42 bits

/*individual color registers*/
reg [11:0] sev;
```

```
reg [11:0] six;
reg [11:0] fiv;
reg [11:0] four;
reg [11:0] three;
reg [11:0] two;
reg [11:0] one;

/*individual color map wires*/
wire [3:0] c_map_1;
wire [3:0] c_map_2;
wire [3:0] c_map_3;
wire [3:0] c_map_4;
wire [3:0] c_map_5;
wire [3:0] c_map_6;
wire [3:0] c_map_7;



reg first_time;

initial first_time = 0;
 //reg julian_test;
 reg done_delay;
 reg [5:0] count;
 initial done_delay = 0;
```

```
initial done = 1; //AARON ADDED THIS LINE SO THAT INITIALLY JULIAN OUTPUTS DONE

   always@(posedge clock) begin

      //AARON ADDED A SEPARATE CONDITIONAL THAT TRIGGERS IF AND ONLY IF
THERE IS A RESET

      if(reset) begin

         done <= 1;




         sev = 12'hFFF;

         six = 12'hD2A;

         fiv = 12'h000;

         four = 12'h00F;

         three =12'h6FF;

         two = 12'hF00;

         one = 12'h0F0;

      end

      /*ready from wubs, while I can read*/

      //THIS SET UP STUFF CAN'T HAPPEN AT RESET, IT NEEDS TO HAPPEN THE FIRST
CLOCK CYCLE IN WHICH READY

      //IS ASSERT FROM AARON

      if (done)

      begin

         done_delay <= 1;

      end

      else

      begin
```

```
            done_delay <= 0;
        end
        if((ready && !start && done_delay)) begin
            start <= 1'b1;                  //start in taking data and calculating
            addr_hold <= addresses;
            bram_addr <= 10'd511;
            max_addr <= addresses[9:0];
            //max_addr <= 10'd1023;
            bin_count <= 0;
            start_count <= 0;
            done <= 1'b0;
            //test_out <= 0;
            bin_acc <= 0;

//            sev = 12'hFFF;
//            six = 12'hD2A;
//            fiv = 12'h000;
//            four = 12'h00F;
//            three =12'h6FF;
//            two = 12'hF00;
//            one = 12'h0F0;

        end

        if(start) begin
```

```
if(bin_count < bin_num) begin
/*data available, start calculation */
    if(start_count >= 2) begin
        start_count <= 0;
         if(bram_addr < max_addr) begin
             bin_acc <= data*data + bin_acc;                  //update our bin energy
             test_out[15:0] <= data;
             bram_addr <= bram_addr + 1;                       //move on to next bram address
             //start_count <= 0;
             //test_out <= 1;
         end

         else if (bram_addr >= max_addr) begin
         /****write code to transfer bin_acc value to color value****/
            case(bin_count)
                3'b000: begin
                    if(adjusting) begin
                        one <= one;             //load in previous value
                    end

                    else begin                  //otherwise update
                        if(c_map_1 >= 15) begin
                            one <= 12'h0FF;
```

```
        end

        else begin
            one <= {4'h0,4'hF - c_map_1, 4'hF};
        end

    end
end

3'b001: begin
    c_map_test <=c_map_2;
    if(adjusting) begin
        two <= two;
    end

    else begin
        if(c_map_2 > 11)begin
            two <= {4'h0,4'h8,4'h0};
        end

        if(c_map_2 <= 4) begin
            two <=  {4'h0, 4'hF, (4'h4 - c_map_2)};
        end

        else if((c_map_2 > 4) && (c_map_2 <= 11)) begin
```

```
            two <= {4'h0,4'hF - (c_map_2 - 4'h4),4'h0};
        end

    end

end

3'b010:begin

    if(!first_time)begin
        julian_test <= max_addr;
        first_time <= 1;
    end
    if(adjusting) begin
        three <= three;
    end

    else begin
        if(c_map_3 >= 12) begin
            three <= {4'hB,4'h0, 4'h0};
        end

        if(c_map_3 < 7) begin
            three <= {4'hF,4'h7-c_map_3,4'h0};
        end
```

```
        else if ((c_map_3 >= 7) && (c_map_3 < 12)) begin
            three <= {4'hF - (c_map_3 - 4'h7),4'h0,4'h0};
        end

    end
end

3'b011: begin
    if(adjusting) begin
        four <= four;
    end

    else begin
        if(c_map_4 >= 12) begin
            four <= {4'hF,4'h0,4'h3};
        end

        if(c_map_4 < 2)begin
            four <= {4'hF,4'h2 - c_map_4,4'hD};
        end

        else if ((c_map_4 >= 2) && (c_map_4 <12)) begin
            four <= {4'hF,4'h0,4'hD - (c_map_4 - 4'h2)};
        end
```

```
    end




end
3'b100:begin
    if(adjusting)begin
        fiv <= fiv;
    end

    else begin
        if(c_map_5 > 10) begin
            fiv <= {4'h3,4'h0,4'hF};
        end

        else begin
            fiv <= {4'hD - c_map_5, 4'h0,4'hF};
        end
    end

end



3'b101: begin
```

```
    if(adjusting) begin
        six <= six;
    end

    else begin
        if(c_map_6 > 13) begin
            six <= {4'h3,4'h0,4'hF};
        end

        else begin
            six <= {4'hF - c_map_6,4'h0,4'h0};
        end

    end
end




3'b110:begin
    if(adjusting)begin
        sev <= sev;
    end

    else begin
```

```
            if(c_map_7 > 13) begin
                sev <= {4'h3,4'h0,4'h0};
            end

            else begin
                sev <={4'h0,4'h0,4'hF - c_map_7 };
            end
        end
    end




    3'b111: begin


    end


endcase



//test_out <= bin_acc;
bin_count <= bin_count + 1;
bin_acc <= 0;
addr_hold <= addr_hold >>10;
```

```
            max_addr <= addr_hold>>10;     //get next max bin address
//              start_count <= 0;
          end


    end



    else begin
        start_count <= start_count + 1;
    end

end


else begin
    done <= 1'b1;
    start <= 0;
    count <= count + 1;
    if (count > 10)
    begin
        count <= 0;
        //julian_test <= 1;
    end
    //possibly wait a clock cycle
    //test_out <= 4;
    //bin_acc <= 1;
```

```
            end


        end


//        else if (done) begin
//            start <= 1'b0;
//        end
    end
 assign acc = bin_acc;
 assign c_map_1 = {bin_acc[off1 + 3],bin_acc[off1 + 2],bin_acc[off1 + 1],bin_acc[off1]} ;
//offset bits of accumulation register
 assign c_map_2 = {bin_acc[off2 + 3],bin_acc[off2 + 2],bin_acc[off2 + 1],bin_acc[off2]};
//offset bits of accumulation register
 assign c_map_3 = {bin_acc[off3 + 3],bin_acc[off3 + 2],bin_acc[off3 + 1],bin_acc[off3]} ;
//offset bits of accumulation register
 assign c_map_4 = {bin_acc[off4 + 3],bin_acc[off4 + 2],bin_acc[off4 + 1],bin_acc[off4]} ;
//offset bits of accumulation register
 assign c_map_5 = {bin_acc[off5 + 3],bin_acc[off5 + 2],bin_acc[off5 + 1],bin_acc[off5]} ;
//offset bits of accumulation register
 assign c_map_6 = {bin_acc[off6 + 3],bin_acc[off6 + 2],bin_acc[off6 + 1],bin_acc[off6]} ;
//offset bits of accumulation register
 assign c_map_7 = {bin_acc[off7 + 3],bin_acc[off7 + 2],bin_acc[off7 + 1],bin_acc[off7]} ;
//offset bits of accumulation register
 assign color = {sev,six,fiv,four,three,two,one};
endmodule
// a simple synchronous FIFO (first-in first-out) buffer
// Parameters:
//    LOGSIZE  (parameter) FIFO has 1<<LOGSIZE elements
```

```
//    WIDTH    (parameter) each element has WIDTH bits
// Ports:
//    clk      (input) all actions triggered on rising edge
//    reset    (input) synchronously empties fifo
//    din      (input, WIDTH bits) data to be stored
//    wr       (input) when asserted, store new data
//    full     (output) asserted when FIFO is full
//    dout     (output, WIDTH bits) data read from FIFO
//    rd       (input) when asserted, removes first element
//    empty    (output) asserted when fifo is empty
//    overflow (output) asserted when WR but no room, cleared on next RD
module fifo #(parameter  LOGSIZE = 2,   // default size is 4 elements
                         WIDTH = 4)     // default width is 4 bits
              (input clk,reset,rd,wr, input [WIDTH-1:0] din,
             output full,empty,overflow, output [WIDTH-1:0] dout);
  parameter SIZE = 1 << LOGSIZE;  // compute size

  reg [WIDTH-1:0] fifo[SIZE-1:0];   // fifo data stored here
  reg overflow;                     // true if WR but no room, cleared on RD
  reg [LOGSIZE-1:0] wptr,rptr;  // fifo write and read pointers

  wire [LOGSIZE-1:0] wptr_inc = wptr + 1;

  assign empty = (wptr == rptr);
  assign full = (wptr_inc == rptr);
```

```
  assign dout = fifo[rptr];

  always @(posedge clk) begin
    if (reset) begin
      wptr <= 0;
      rptr <= 0;
      overflow <= 0;
    end
    else if (wr) begin
      // store new data into the fifo
      fifo[wptr] <= din;
      wptr <= wptr_inc;
      overflow <= overflow | (wptr_inc == rptr);
    end

    // bump read pointer if we're done with current value.
    // RD also resets the overflow indicator
    if (rd && (!empty || overflow)) begin
      rptr <= rptr + 1;
      overflow <= 0;
    end
  end
endmodule
`timescale 1ns / 1ps
```

```
module filter_control(
    input clock,
    input reset,
    input [1:0] switch,
    input ready,
    input [7:0] audio_in,
    output [7:0] audio_out,
    output done,
    output wire flag,
    output wire last_flag
    );



    wire no_filter;

    assign no_filter = (switch == 2'd0) ?
        0 :   (switch==2'd1) ?
      0 : (switch== 2'd2) ?
      0 : (switch == 2'd3) ?
      1 : 0;

    //low pass filter signals and instantiations
    wire signed [9:0] lpf_coeff;
    wire [4:0] lpf_idx;
```

```
    wire lpf_done;
    wire lpf_ready;
    assign lpf_ready = (switch == 2'd0) ? ready : 0;
    wire signed [17:0] lpf_filter_out;

    lpf_coeffs low_pass_coeffs(.index(lpf_idx), .coeff(lpf_coeff));

    fir31 lpf(.clock(clock),.reset(reset),.ready(lpf_ready), .x(audio_in), .coeff(lpf_coeff),
.idx(lpf_idx),
        .y(lpf_filter_out), .done(lpf_done), .no_filter(no_filter));

    //high pass filter signals and instantiations
    wire signed [9:0] hpf_coeff;
    wire [4:0] hpf_idx;
    wire hpf_done;
    wire hpf_ready;
    assign hpf_ready = (switch == 2'd1) ? ready: 0;
    wire signed [17:0] hpf_filter_out;

    hpf_coeffs high_pass_coeffs(.index(hpf_idx), .coeff(hpf_coeff));

    fir31 hpf(.clock(clock),.reset(reset),.ready(hpf_ready), .x(audio_in), .coeff(hpf_coeff),
.idx(hpf_idx),
        .y(hpf_filter_out), .done(hpf_done), .no_filter(no_filter));
```

```
    //band pass 1 filter signals and instations
    wire signed [9:0] bpf1_coeff;
    wire [4:0] bpf1_idx;
    wire bpf1_done;
    wire bpf1_ready;
    assign bpf1_ready = (switch == 2'd2) ? ready: 0;
    wire signed [17:0] bpf1_filter_out;

    bpf1_coeffs band_pass_1_coeffs(.index(bpf1_idx), .coeff(bpf1_coeff));

    fir31 bpf1(.clock(clock),.reset(reset),.ready(bpf1_ready), .x(audio_in), .coeff(bpf1_coeff),
.idx(bpf1_idx),
        .y(bpf1_filter_out), .done(bpf1_done), .no_filter(no_filter));

    //band pass 2 filter signals and instations
    //NOT REALLY BANDPASS JUST PASSTHROUGH
    wire signed [9:0] bpf2_coeff;
    wire [4:0] bpf2_idx;
    wire bpf2_done;
    wire bpf2_ready;
    assign bpf2_ready = (switch == 2'd3) ? ready: 0;
    wire signed [17:0] bpf2_filter_out;

    bpf2_coeffs band_pass_2_coeffs(.index(bpf2_idx), .coeff(bpf2_coeff));
```

```
    fir31 bpf2(.clock(clock),.reset(reset),.ready(bpf2_ready), .x(audio_in), .coeff(bpf2_coeff),
.idx(bpf2_idx),
        .y(bpf2_filter_out), .done(bpf2_done), .no_filter(no_filter));


    //output multiplexing
    assign done = (switch == 2'd0) ?
        lpf_done: (switch==2'd1) ?
        hpf_done : (switch== 2'd2) ?
        bpf1_done : (switch == 2'd3) ?
        bpf2_done : 0;
    assign audio_out = (switch == 2'd0) ?
        lpf_filter_out[17:10]: (switch==2'd1) ?
        hpf_filter_out[17:10]: (switch== 2'd2) ?
        bpf1_filter_out[17:10] : (switch == 2'd3) ?
        bpf2_filter_out[7:0] : 0;
endmodule
`timescale 1ns / 1ps



module final_project(
    //MASTER CLOCK
    input CLK100MHZ,


    //switches
    input [15:0] SW,
```

```
//buttons
input BTNC, BTNU, BTNL, BTNR, BTND,

//SD card stuff
input SD_CD,
output SD_SCK,
output SD_CMD,
inout [3:0] SD_DAT,
output SD_RESET,
//JA and JB PMOD Headers for Camera input
inout [7:0] JA,
inout [7:0] JB,

// video
output [3:0] VGA_R,
output [3:0] VGA_G,
output [3:0] VGA_B,
output VGA_HS,
output VGA_VS,

//LEDs
output LED16_B, LED16_G, LED16_R,
output LED17_B, LED17_G, LED17_R,
```

```
 output[15:0] LED,
 output [7:0] SEG,  // segments A-G (0-6), DP (7)
 output [7:0] AN,  // Display 0-7

 // audio
 output AUD_PWM,
 output AUD_SD // PWM audio enable
 );

 //master reset
 wire master_reset = SW[15];

//CLOCK SETUP
 wire clk_100mhz;
 wire clk_25mhz;
 wire locked = 1;
 clk_manager clocker (
    .clk_in1(CLK100MHZ),    // input clk_in1
    .clk_out1(clk_25mhz),
    .clk_out2(clk_100mhz),
    .reset(SW[15]), // input reset
    .locked(locked) // output locked
 );
```

```
// debounce buttons
wire btn_up, btn_down, btn_center, btn_left, btn_right;
debounce up(.reset(master_reset), .clock(clk_25mhz), .noisy(BTNU), .clean(btn_up));
debounce down(.reset(master_reset), .clock(clk_25mhz), .noisy(BTND), .clean(btn_down));
debounce center(.reset(master_reset), .clock(clk_25mhz), .noisy(BTNC), .clean(btn_center));
debounce left(.reset(master_reset), .clock(clk_25mhz), .noisy(BTNL), .clean(btn_left));
debounce right(.reset(master_reset), .clock(clk_25mhz), .noisy(BTNR), .clean(btn_right));



//8-HEX SETUP FOR TESTING
wire [31:0] data;
wire [6:0] segments;
display_8hex display(.clk(clk_25mhz),.data(data), .seg(segments), .strobe(AN));
assign SEG[6:0] = segments;
assign SEG[7] = 1'b1;



//SETTING UP THE SD CARD
wire [7:0] sd_read;
wire [7:0] sd_write;
//wire sd_we;  //0 = read, 1 = write
wire sd_read_available;
wire sd_write_available;
```

```
wire sd_write_ready = 0;

wire sd_reset = 0;

wire sd_ready;

wire [31:0] sd_addr;

wire [4:0] sd_status;


// set SPI mode

assign SD_DAT[2] = 1;

assign SD_DAT[1] = 1;

assign SD_RESET = 0;

wire sd_rd;

sd_controller sd_read_write (

    .cs (SD_DAT[3]),

    .mosi (SD_CMD),

    .miso (SD_DAT[0]),

    .sclk (SD_SCK),

    .rd (sd_rd), //input rd. The byte is presented on [dout].

    .dout (sd_read), // output reg [7:0] dout for READ operation.

    .byte_available (sd_read_available), // A new byte has been presented on [dout].

    .wr(0), //input wr. The next byte to be written should be presentaed on [din].

    .din(sd_write), // Data input for WRITE operation.

    .ready_for_next_byte (sd_write_available), // A new byte should be presented on [din].

    .reset(sd_reset), // input Resets controller on assertion.

    .ready (sd_ready), //output HIGH if the SD card is ready for a read or write operation.

    .address(sd_addr), //input [31:0] addres
```

```
    .clk(clk_25mhz), // 25 MHz clock.
    .status(sd_status) //output [4:0] status: Current state of controller.
);



wire [11:0] memory_read_data;
wire [11:0] memory_write_data;
wire [18:0] memory_read_addr;
wire [18:0] memory_write_addr;
wire memory_write_enable;

//frame buffer memory
frame_buffer frame_buffer_1 (
    .clka(clk_25mhz),
    .wea(memory_write_enable),
    .addra(memory_write_addr),
    .dina(memory_write_data),
    .clkb(clk_25mhz),
    .addrb(memory_read_addr),
    .doutb(memory_read_data)
);

wire [2:0] edge_bram_din;
wire [2:0] edge_bram_dout;
wire [18:0] edge_bram_addr;
```

```
wire edge_bram_we;

wire [2:0] edge_bram_dinb;
wire [2:0] edge_bram_doutb;
wire [18:0] edge_bram_addrb;

wire [18:0] image_edge_bram_addr;
wire [18:0] image_edge_bram_addrb;
wire [2:0] image_edge_bram_din;
wire image_done;

wire [18:0] image_memory_read_addr;
wire [18:0] vga_bram_addr;


assign edge_bram_addr = image_edge_bram_addr;
assign edge_bram_din = image_edge_bram_din;
assign memory_read_addr = ~image_done ? image_memory_read_addr : vga_bram_addr;
assign edge_bram_addrb = ~image_done ? image_edge_bram_addrb : vga_bram_addr;

//a = write, b = read
xy_bin xy_edge (
    .clka(clk_25mhz),    // input wire clka
    .wea(1),      // input wire [0 : 0] wea
    .addra(edge_bram_addr),  // input wire [18 : 0] addra
```

```
    .dina(edge_bram_din),    // input wire [2 : 0] dina
    .douta(edge_bram_dout),  // output wire [2 : 0] douta
    .clkb(clk_25mhz),
    .web(0),
    .addrb(edge_bram_addrb),
    .dinb(edge_bram_dinb),
    .doutb(edge_bram_doutb)
);




wire [3:0] VGA_R_w;
wire [3:0] VGA_B_w;
wire [3:0] VGA_G_w;
wire VGA_VS_w;
wire VGA_HS_w;

image_processing box(
    .clk_25mhz(clk_25mhz),
    .CLK_100M(clk_100mhz),
    .num_bins(3'b100),
    .done(LED[15]),
    .sobel_start(SW[9]),
    .edge_start(SW[10]),
```

```
        .color_start(SW[11]),

        //frame buffer
        .memory_read_data(memory_read_data),
        .memory_write_data(memory_write_data),
        .memory_read_addr(image_memory_read_addr),
        .memory_write_addr(memory_write_addr),
        .memory_write_enable(memory_write_enable),

        //xy bram
        .edge_bram_doutb(edge_bram_doutb),
        .edge_bram_addr(image_edge_bram_addr),
        .edge_bram_addrb(image_edge_bram_addrb),
        .edge_bram_din(image_edge_bram_din),

        .VGA_R(VGA_R_w), .VGA_B(VGA_B_w), .VGA_G(VGA_G_w), .VGA_HS(VGA_HS_w),
.VGA_VS(VGA_VS_w),

        .JA(JA), .JB(JB), .BTNC(btn_center), .BTNR(btn_right), .BTNL(btn_left)
    );

    wire [31:0] julian_debug;
    wire [9:0] julian_test;
    //wire adjusting;
```

```
    wire [3:0] VGA_R_j;
    wire [3:0] VGA_B_j;
    wire [3:0] VGA_G_j;
    wire VGA_VS_j;
    wire VGA_HS_j;
    assign VGA_R = SW[12] ? VGA_R_j : VGA_R_w;
    assign VGA_G = SW[12] ? VGA_G_j : VGA_G_w;
    assign VGA_B = SW[12] ? VGA_B_j : VGA_B_w;
    assign VGA_VS = SW[12] ? VGA_VS_j : VGA_VS_w;
    assign VGA_HS = SW[12] ? VGA_HS_j : VGA_HS_w;
    assign image_done = SW[12];



    /*julian debug statement*/
    wire [9:0] addrb;
    wire [15:0] doutb;
    wire [59:0] bin_addr;
    wire [2:0] bin;
    Color_transform julian_color(.CLK100MHZ(clk_100mhz), .reset(master_reset),
.clock_25mhz(clk_25mhz), .SW(SW[15:0]),
        .BTNC(btn_center), .BTNU(btn_up), .BTNL(btn_left), .BTNR(btn_right), .BTND(btn_down),
        .VGA_R(VGA_R_j), .VGA_B(VGA_B_j), .VGA_G(VGA_G_j), .VGA_HS(VGA_HS_j),
.VGA_VS(VGA_VS_j), .addr2_b(addrb),
        .dout2_b(doutb), .addr_b(vga_bram_addr), .dout_b(edge_bram_doutb),
.wings_done(image_done),
        .wubs_done(aaron_done), .addresses(bin_addr), .bin_num(bin), .FFT_done(julian_done),
```

```
    .data_disp(julian_debug), .julian_test(julian_test), .start(start));

//FFT amplitude output buffer read and write signals
wire fft_bram_out_write_enablea;
wire [9:0] fft_bram_out_addra;
wire [15:0] fft_bram_out_dina;
wire [15:0] fft_bram_out_douta;

bram_fft_output_buffer fft_amplitude_buffer (
    .clka(clk_100mhz),    // input wire clka
    .wea(fft_bram_out_write_enablea),      // input wire [0 : 0] wea
    .addra(fft_bram_out_addra),  // input wire [9 : 0] addra
    .dina(fft_bram_out_dina),    // input wire [15 : 0] dina
    .douta(fft_bram_out_douta),  // output wire [15 : 0] douta
    .clkb(clk_100mhz),    // input wire clkb
    .web(0),      // input wire [0 : 0] web
    .addrb(addrb),  // input wire [9 : 0] addrb
    .dinb(dinb),    // input wire [15 : 0] dinb
    .doutb(doutb)  // output wire [15 : 0] doutb
);


wire [2:0] state;
wire filter_flag;
wire filter_done;
```

```
    wire flag;

    wire last_flag;

    wire filter_ready;

    wire test;

    audio_processing audio_stuff(.clock(clk_100mhz), .clk_25mhz(clk_25mhz),
.reset(master_reset), .wings_done(image_done),

        .julian_done(julian_done), .sd_data(sd_read),.sd_ready(sd_ready),
.sd_read_available(sd_read_available),

        .AUD_PWM(AUD_PWM),.sd_addr(sd_addr), .AUD_SD(AUD_SD), .sd_rd(sd_rd),
.filter_control(SW[8:7]),

        .fft_bram_out_write_enablea(fft_bram_out_write_enablea),
.fft_bram_out_addra(fft_bram_out_addra),

        .fft_bram_out_dina(fft_bram_out_dina), .done(aaron_done),.bin(bin),.bin_addr(bin_addr),

        .state(state), .filter_flag(filter_flag), .filter_done(filter_done), .flag(flag), .last_flag(last_flag),
.filter_ready(filter_ready), .test(test));




    assign data[15:0] = julian_debug[15:0];

    assign data[31:16] = {6'b0, julian_test };




endmodule

//////////////////////////////////////////////////////////////////////////////////

//

// 31-tap FIR filter, 8-bit signed data, 10-bit signed coefficients.

// ready is asserted whenever there is a new sample on the X input,

// the Y output should also be sampled at the same time.  Assumes at
```

```
// least 32 clocks between ready assertions.  Note that since the
// coefficients have been scaled by 2**10, so has the output (it's
// expanded from 8 bits to 18 bits).  To get an 8-bit result from the
// filter just divide by 2**10, ie, use Y[17:10].
//
//////////////////////////////////////////////////////////////////////////////////

module fir31(
  input wire clock,reset,ready,
  input wire [7:0] x,
  input wire signed [9:0] coeff,
  output wire [4:0] idx,
  output reg signed [17:0] y,
  output done,
  input no_filter
);


  reg [7:0] sample [31:0];  // 32 element array each 8 bits wide
  reg signed [17:0] accumulator;
  reg [4:0] idx_reg;
  reg flag;
  reg last_flag;
  reg [4:0] offset;
  assign idx = idx_reg;
```

```
initial offset = 0;
initial flag = 1;
initial last_flag = 1;
initial idx_reg = 0;

initial sample[offset] = x;

assign done = (last_flag == 0 && flag==1) ? 1:0;
always @(posedge clock)
begin
//reset values each time output is updated
    last_flag <= flag;
            if(ready && flag)
            begin
                accumulator <= 0;
                sample[offset] <= x;
                flag <= 0;
                idx_reg <= 0;
                offset <= offset + 1;
            end
            if (idx_reg <32 && !flag)
            begin
                idx_reg <= idx_reg +1;
                if (offset-idx_reg <0)
```

```
                        begin
                        //make sure to cycle around for negative index values
                                accumulator <= accumulator + coeff*sample[(offset-idx_reg + 32)];
                        end
                        else
                        begin
                                accumulator <= accumulator + coeff*sample[(offset-idx_reg)];
                        end
                end
                if (idx_reg == 31)
                begin
                 if(no_filter)
                 begin
                    y <= x;
        end
        else
        begin
                        y <= accumulator;
        end
                        flag <=1;
                end
    end
endmodule
//640hz high pass filter coeffs
module hpf_coeffs(
```

```
  input wire [4:0] index,
  output reg signed [9:0] coeff
);
  // tools will turn this into a 31x10 ROM
  always @(index)
    case (index)
      5'd0:  coeff = -10'sd2;
      5'd1:  coeff = -10'sd2;
      5'd2:  coeff = -10'sd3;
      5'd3:  coeff = -10'sd5;
      5'd4:  coeff = -10'sd7;
      5'd5:  coeff = -10'sd10;
      5'd6:  coeff = -10'sd14;
      5'd7:  coeff = -10'sd17;
      5'd8:  coeff = -10'sd21;
      5'd9:  coeff = -10'sd25;
      5'd10: coeff = -10'sd29;
      5'd11: coeff = -10'sd33;
      5'd12: coeff = -10'sd36;
      5'd13: coeff = -10'sd39;
      5'd14: coeff = -10'sd40;
      5'd15: coeff = 10'sd982;
      5'd16: coeff = -10'sd40;
      5'd17: coeff = -10'sd39;
      5'd18: coeff = -10'sd36;
```

```
        5'd19: coeff = -10'sd33;
        5'd20: coeff = -10'sd29;
        5'd21: coeff = -10'sd25;
        5'd22: coeff = -10'sd21;
        5'd23: coeff = -10'sd17;
        5'd24: coeff = -10'sd13;
        5'd25: coeff = -10'sd10;
        5'd26: coeff = -10'sd5;
        5'd27: coeff = -10'sd2;
        5'd28: coeff = -10'sd2;
        5'd29: coeff = -10'sd2;
        5'd30: coeff = -10'sd2;
        default: coeff = 10'hXXX;
    endcase
endmodule
`timescale 1ns / 1ps



module image_processing(
    input wire clk_25mhz,
    input wire CLK_100M,
    input wire [5:0] SW,
    input wire [2:0] num_bins,
    output wire done,
    input wire sobel_start,
```

```
input wire edge_start,
input wire color_start,

//frame buffer
input wire [11:0] memory_read_data,
output wire [11:0] memory_write_data,
output wire [18:0] memory_read_addr,
output wire [18:0] memory_write_addr,
output wire memory_write_enable,

//xy bram
input wire [2:0] edge_bram_doutb,
output wire [18:0] edge_bram_addr,
output wire [18:0] edge_bram_addrb,
output wire [2:0] edge_bram_din,

//display
output wire [3:0] VGA_R,
output wire [3:0] VGA_B,
output wire [3:0] VGA_G,
output wire VGA_HS,
output wire VGA_VS,

//camera
inout wire [7:0] JA,
```

```
    inout wire [7:0] JB,
    input wire BTNC,
    input wire BTNR,
    input wire BTNL


);
    //camera signals
wire camera_pwdn;
wire camera_clk_out;
wire [7:0] camera_dout;
wire camera_scl, camera_sda;
wire camera_vsync, camera_hsync;
wire [15:0] camera_pixel;
wire camera_pixel_valid;
wire camera_reset;
wire camera_frame_done;

assign camera_pwdn = 0;
assign camera_reset = 1;

//assign camera outputs
assign JA[0] = camera_pwdn;
assign camera_dout[0] = JA[1];
assign camera_dout[2] = JA[2];
```

```
assign camera_dout[4] = JA[3];
assign JA[4] = camera_reset;
assign camera_dout[1] = JA[5];
assign camera_dout[3] = JA[6];
assign camera_dout[5] = JA[7];

assign camera_dout[6] = JB[0];
assign JB[1] = clk_25mhz;
assign camera_hsync = JB[2];
//assign JB[3]= camera_sda;
assign camera_dout[7] = JB[4];
assign camera_clk_out = JB[7];
assign camera_vsync = JB[5];
//assign JB[7] = camera_scl;



//camera configuration module
camera_configure camera_configure_1 (
    .clk(clk_25mhz),
    .start(BTNC),
    .sioc(JB[6]),
    .siod(JB[3]),
    .done()
    );
```

```
//camera interface
camera_read camera_read_1 (
    .p_clock(camera_clk_out),
    .vsync(camera_vsync),
    .href(camera_hsync),
    .p_data(camera_dout),
    .pixel_data(camera_pixel),
    .pixel_valid(camera_pixel_valid),
    .frame_done(camera_frame_done)
    );

//write camera data to frame buffer
camera_address_gen camera_address_gen_1 (
    .camera_clk(camera_clk_out),
    .camera_pixel_valid(camera_pixel_valid),
    .camera_frame_done(camera_frame_done),
    .capture_frame(BTNR),
    .camera_pixel(camera_pixel),
    .memory_data(memory_write_data),
    .memory_addr(memory_write_addr_cam),
    .memory_we(memory_write_enable_cam)
    );

wire [18:0] memory_write_addr_cam;
```

```
wire memory_write_enable_cam;


assign memory_write_addr = memory_write_addr_cam; //image_start ? 0 :
memory_write_addr_cam;

assign memory_write_enable = memory_write_enable_cam; //image_start ? 0 :
memory_write_enable_cam;




    wire sobel_done;

    wire [18:0] sobel_edge_bram_addr;

    wire [18:0] sobel_rgb_bram_addr;

    wire [2:0] sobel_edge_bram_din;


    //sobel edge detection

    sobel edge_detection(

        .clk(CLK_100M),

        .start(sobel_start),

        .done(sobel_done),

        .SW(SW),


        .pixel_data(memory_read_data),

        .pic_memory_addr(sobel_rgb_bram_addr),


        .is_edge(sobel_edge_bram_din),
```

```
    .edge_memory_addr(sobel_edge_bram_addr)
);




wire erosion_done;
wire [18:0] erosion_edge_bram_addrb;
wire [2:0] erosion_edge_bram_din;
wire [18:0] erosion_edge_bram_addr;
wire [2:0] erosion_edge_bram_doutb;

edge_pixel_width skinny_edge(
    .clk(clk_25mhz),
    .start(sobel_done),
    .done(erosion_done),

    .bram_read(edge_bram_doutb),
    .bram_write(erosion_edge_bram_din),
    .edge_addr_read(erosion_edge_bram_addrb),
    .edge_addr_write(erosion_edge_bram_addr)

);

wire [9:0] x_start;
```

```
wire [8:0] y_start;
wire [18:0] addr_start;
wire [11:0] num_edge_pixels;
wire one_edge_done;
wire [2:0] one_edge_bram_din;
wire [18:0] one_edge_addrb;
wire [18:0] one_edge_addr;

one_edge isolate(
    .clk(clk_25mhz),
    .num_pixels(num_edge_pixels),
    .done(one_edge_done),
    .start(edge_start),
    .bram_read(edge_bram_doutb),
    .bram_write(one_edge_bram_din),
    .edge_addr_read(one_edge_addrb),
    .edge_addr_write(one_edge_addr),
    .clear(BTNL)
);

wire color_contour_done;
wire [18:0] color_contour_addr;
wire [18:0] color_contour_addrb;
wire [2:0] color_contour_bram_din;
wire color_contour_we;
```

```
    color_contour separate(
        .clk(clk_25mhz),
        .num_pixels(num_edge_pixels),
        .num_bins(num_bins),
        .done(color_contour_done),
        .start(color_start),
        .bram_read(edge_bram_doutb),
        .bram_write(color_contour_bram_din),
        .edge_addr_read(color_contour_addrb),
        .edge_addr_write(color_contour_addr)
    );

//    wire padding_done;
//    wire [18:0] padding_edge_bram_addrb;
//    wire [2:0] padding_edge_bram_din;
//    wire [18:0] padding_edge_bram_addr;
//    padding pad(
//        .clk(clk_25mhz),
//        .start(color_contour_done),
//        .done(padding_done),

//        .bram_read(edge_bram_doutb),
//        .bram_write(padding_edge_bram_din),
```

```
//        .edge_addr_read(padding_edge_bram_addrb),
//        .edge_addr_write(padding_edge_bram_addr)

//    );

    wire [3:0] R, G, B;
    wire [18:0] vga_bram_addr;
    video_playback video_playback_1 (
        .pixel_data(edge_bram_doutb),
        .rgb_data(memory_read_data),
        .video_clk(clk_25mhz),
        .memory_addr(vga_bram_addr),
        .vsync(VGA_VS),
        .hsync(VGA_HS),
        .video_out({R, G, B}), //{VGA_R, VGA_G, VGA_B}), //{R, G, B}), //{
        .contour(sobel_start)
        );

    assign VGA_R = done ? R : 0;
    assign VGA_G = done ? G : 0;
    assign VGA_B = done ? B : 0;

    wire [18:0] edge_bram_addrb_temp;
    wire [18:0] memory_read_addr_temp;
```

```
    assign edge_bram_addr = ~sobel_done ? sobel_edge_bram_addr : ~erosion_done ?
erosion_edge_bram_addr :

                        ~one_edge_done ? one_edge_addr : color_contour_addr;

    assign edge_bram_addrb_temp = ~erosion_done ? erosion_edge_bram_addrb :
~one_edge_done ? one_edge_addrb :

                        color_contour_addrb;

    assign memory_read_addr_temp = sobel_rgb_bram_addr;

    assign edge_bram_din = ~sobel_done ? sobel_edge_bram_din : ~erosion_done ?
erosion_edge_bram_din :

                        ~one_edge_done? one_edge_bram_din :

                        color_contour_bram_din;




    assign done = (sobel_start && edge_start && color_start) ? color_contour_done:
//padding_done : //color_contour_done :

                    (sobel_start && edge_start) ? one_edge_done :

                    (sobel_start) ? erosion_done : 1;




    assign memory_read_addr = ~sobel_start ? vga_bram_addr : done ? vga_bram_addr :
memory_read_addr_temp;

    assign edge_bram_addrb = ~sobel_start ? vga_bram_addr : done? vga_bram_addr :
edge_bram_addrb_temp;




endmodule

`timescale 1ns / 1ps
```

```
module lev_to_pulse(input clock, input reset, input lev,
                output reg pulse
    );



reg lev_prev;



    always@(posedge clock)begin
        lev_prev <= lev;

        if((!lev_prev) && lev)begin
            pulse <= 1;
        end

        else begin
            pulse <= 0;
        end

    end
endmodule
//////////////////////////////////////////////////////////////////////////////////
//
```

```
// 320hz low pass
//
////////////////////////////////////////////////////////////////////////////////


module lpf_coeffs(
  input wire [4:0] index,
  output reg signed [9:0] coeff
);
  // tools will turn this into a 31x10 ROM
  always @(index)
    case (index)
      5'd0:  coeff = 10'sd4;
      5'd1:  coeff = 10'sd5;
      5'd2:  coeff = 10'sd7;
      5'd3:  coeff = 10'sd10;
      5'd4:  coeff = 10'sd14;
      5'd5:  coeff = 10'sd19;
      5'd6:  coeff = 10'sd24;
      5'd7:  coeff = 10'sd30;
      5'd8:  coeff = 10'sd37;
      5'd9:  coeff = 10'sd43;
      5'd10: coeff = 10'sd49;
      5'd11: coeff = 10'sd54;
      5'd12: coeff = 10'sd58;
      5'd13: coeff = 10'sd62;
```

```
      5'd14: coeff = 10'sd64;
      5'd15: coeff = 10'sd64;
      5'd16: coeff = 10'sd64;
      5'd17: coeff = 10'sd62;
      5'd18: coeff = 10'sd58;
      5'd19: coeff = 10'sd54;
      5'd20: coeff = 10'sd49;
      5'd21: coeff = 10'sd43;
      5'd22: coeff = 10'sd37;
      5'd23: coeff = 10'sd30;
      5'd24: coeff = 10'sd24;
      5'd25: coeff = 10'sd19;
      5'd26: coeff = 10'sd14;
      5'd27: coeff = 10'sd10;
      5'd28: coeff = 10'sd7;
      5'd29: coeff = 10'sd5;
      5'd30: coeff = 10'sd4;
      default: coeff = 10'hXXX;

    endcase
endmodule
`timescale 1ns / 1ps



module one_edge(
```

```
    input clk,
//    output reg [9:0] x_start,
//    output reg [8:0] y_start,
//    output reg [18:0] addr_start,
    output reg [11:0] num_pixels,
    output reg done = 0,
    input start,

    //writing to the BRAM of the edges
    input wire [2:0] bram_read,
    output reg [2:0] bram_write,
    output reg [18:0] edge_addr_read,
    output reg [18:0] edge_addr_write,

    input wire clear

    );

    parameter WIDTH = 640;
    parameter HEIGHT = 480;


    parameter STATE_SETUP = 0;
    parameter STATE_WAIT = 1;
    parameter STATE_EXPLORE = 2;
```

```
parameter STATE_IS_IT_EDGE = 3;
parameter STATE_WAIT_TWO = 4;
parameter STATE_CLEAR_REST = 5;
parameter STATE_GET_FIRST = 6;

reg [2:0] explore_dir = 0;
reg [2:0] max_explore_dir = 3'b111;
parameter DIR_GET_RIGHT = 0;
parameter DIR_GET_DOWNRIGHT = 1;
parameter DIR_GET_DOWN = 2;
parameter DIR_GET_DOWNLEFT = 3;
parameter DIR_GET_LEFT = 4;
parameter DIR_GET_UPLEFT = 5;
parameter DIR_GET_UP = 6;
parameter DIR_GET_UPRIGHT = 7;

reg [2:0] state = STATE_SETUP;
reg [2:0] next_state;


reg [9:0] x_prev = 0;
reg [8:0] y_prev = 0;
reg [18:0] addr_prev = 0;
reg [9:0] x_curr;
reg [8:0] y_curr;
```

```
reg [18:0] addr_curr = 0;

reg [9:0] x_explore;
reg [8:0] y_explore;
reg [18:0] addr_explore = 0;

reg [18:0] addr_start;




always @(posedge clk) begin
    if (~start) begin
        state <= STATE_SETUP;
        done <= 0;
        num_pixels <= 0;
    end
    else if (clear) begin
        state <= STATE_CLEAR_REST;
        edge_addr_read <= 0;
    end
    else if (~done) begin


    case (state)
```

```
            STATE_SETUP: begin
                bram_write <= 3'b000;
                edge_addr_read <= 0;
                edge_addr_write <= 0;
                x_curr <= 0;
                y_curr <= 0;

                next_state <= STATE_GET_FIRST;
                state <= STATE_WAIT;
            end

            STATE_GET_FIRST: begin
                if (bram_read == 3'b011 && x_curr > 35 && y_curr > 35) begin
//                  x_start <= x_curr;
//                  y_start <= y_curr;
                    addr_start <= edge_addr_read;

                    bram_write <= 3'b111;
                    edge_addr_write <= edge_addr_read;

                    addr_curr <= edge_addr_read;

                    state <= STATE_EXPLORE;
                end
                else begin
```

```
        edge_addr_read <= edge_addr_read + 1;
        if (x_curr == WIDTH - 1) begin
            x_curr <= 0;
            y_curr <= y_curr + 1;
        end else begin
            x_curr <= x_curr + 1;
        end

        state <= STATE_WAIT;
        next_state <= STATE_GET_FIRST;

    end
end




STATE_WAIT: begin
    state <= STATE_WAIT_TWO;
end

STATE_WAIT_TWO: begin
    state <= next_state;
end

STATE_EXPLORE: begin
```

```
next_state <= STATE_IS_IT_EDGE;
state <= STATE_WAIT;

case (explore_dir)
   DIR_GET_RIGHT: begin
      if (addr_curr + 1 == addr_prev) begin
         state <= STATE_EXPLORE;
         explore_dir <= explore_dir + 1;
      end else begin
         edge_addr_read <= addr_curr + 1;
      end
   end

   DIR_GET_DOWNRIGHT: begin
      if (addr_curr + 640 + 1 == addr_prev) begin
         state <= STATE_EXPLORE;
         explore_dir <= explore_dir + 1;
      end else begin
         edge_addr_read <= addr_curr + 640 + 1;
      end
   end

   DIR_GET_DOWN: begin
      if (addr_curr + 640 == addr_prev) begin
         state <= STATE_EXPLORE;
```

```
            explore_dir <= explore_dir + 1;
        end else begin
            edge_addr_read <= addr_curr + 640;
        end
    end

    DIR_GET_DOWNLEFT: begin
        if (addr_curr + 640 - 1 == addr_prev) begin
            state <= STATE_EXPLORE;
            explore_dir <= explore_dir + 1;
        end else begin
            edge_addr_read <= addr_curr + 640 - 1;
        end
    end

    DIR_GET_LEFT: begin
        if (addr_curr - 1 == addr_prev) begin
            state <= STATE_EXPLORE;
            explore_dir <= explore_dir + 1;
        end else begin
            edge_addr_read <= addr_curr - 1;
        end
    end

    DIR_GET_UPLEFT: begin
```

```
    if (addr_curr - 640 - 1 == addr_prev) begin
        state <= STATE_EXPLORE;
        explore_dir <= explore_dir + 1;
    end else begin
        edge_addr_read <= addr_curr - 640 - 1;
    end
end

DIR_GET_UP: begin
    if (addr_curr - 640 == addr_prev) begin
        state <= STATE_EXPLORE;
        explore_dir <= explore_dir + 1;
    end else begin
        edge_addr_read <= addr_curr - 640;
    end
end

DIR_GET_UPRIGHT: begin
    if (addr_curr - 640 + 1 == addr_prev) begin
        state <= STATE_EXPLORE;
        explore_dir <= explore_dir + 1;
    end else begin
        edge_addr_read <= addr_curr - 640 + 1;
    end
end
```

```
    endcase
end

STATE_IS_IT_EDGE: begin
    state <= STATE_EXPLORE;

    if (edge_addr_read == addr_start) begin

        edge_addr_read <= 0;
        state <= STATE_WAIT;
        next_state <= STATE_CLEAR_REST;
    end
    else begin
        if (bram_read == 3'b011) begin
            //update our memory of current and past pixels
            addr_prev <= addr_curr;
            addr_curr <= edge_addr_read;

            explore_dir <= 0;

            //write that bin into the bram location
            edge_addr_write <= edge_addr_read;
            bram_write <= 3'b111;
        end
        else begin
```

```
            explore_dir <= explore_dir + 1;
        end
    end
end

STATE_CLEAR_REST: begin
    edge_addr_read <= edge_addr_read + 1;
    if (bram_read == 3'b111) begin
        bram_write <= 3'b001;
        edge_addr_write <= edge_addr_read;
        num_pixels <= num_pixels + 1;

    end
    else begin
        bram_write <= 3'b000;
        edge_addr_write <= edge_addr_read;
    end

    if (edge_addr_read == 307200) begin
        done <= 1;
    end

    state <= STATE_WAIT;
    next_state <= STATE_CLEAR_REST;
```

```
            end



        endcase
        end
    end
endmodule
`timescale 1ns / 1ps



module OV7670_config_rom(
    input wire clk,
    input wire [7:0] addr,
    output reg [15:0] dout
    );
    //FFFF is end of rom, FFF0 is delay
    always @(posedge clk) begin
    case(addr)
    0:  dout <= 16'h12_80; //reset
    1:  dout <= 16'hFF_F0; //delay
    2:  dout <= 16'h12_04; // COM7,     set RGB color output
    3:  dout <= 16'h11_80; // CLKRC     internal PLL matches input clock
    4:  dout <= 16'h0C_00; // COM3,     default settings
    5:  dout <= 16'h3E_00; // COM14,    no scaling, normal pclock
    6:  dout <= 16'h04_00; // COM1,     disable CCIR656
```

```
7:  dout <= 16'h40_d0; //COM15,     RGB565, full output range
8:  dout <= 16'h3a_04; //TSLB       set correct output data sequence (magic)
9:  dout <= 16'h14_18; //COM9       MAX AGC value x4
10: dout <= 16'h4F_B3; //MTX1       all of these are magical matrix coefficients
11: dout <= 16'h50_B3; //MTX2
12: dout <= 16'h51_00; //MTX3
13: dout <= 16'h52_3d; //MTX4
14: dout <= 16'h53_A7; //MTX5
15: dout <= 16'h54_E4; //MTX6
16: dout <= 16'h58_9E; //MTXS
17: dout <= 16'h3D_C0; //COM13      sets gamma enable, does not preserve reserved bits, may be wrong?
18: dout <= 16'h17_14; //HSTART     start high 8 bits
19: dout <= 16'h18_02; //HSTOP      stop high 8 bits //these kill the odd colored line
20: dout <= 16'h32_80; //HREF       edge offset
21: dout <= 16'h19_03; //VSTART     start high 8 bits
22: dout <= 16'h1A_7B; //VSTOP      stop high 8 bits
23: dout <= 16'h03_0A; //VREF       vsync edge offset
24: dout <= 16'h0F_41; //COM6       reset timings
25: dout <= 16'h1E_00; //MVFP       disable mirror / flip //might have magic value of 03
26: dout <= 16'h33_0B; //CHLF       //magic value from the internet
27: dout <= 16'h3C_78; //COM12      no HREF when VSYNC low
28: dout <= 16'h69_00; //GFIX       fix gain control
29: dout <= 16'h74_00; //REG74      Digital gain control
30: dout <= 16'hB0_84; //RSVD       magic value from the internet *required* for good color
```

```
31: dout <= 16'hB1_0c; //ABLC1
32: dout <= 16'hB2_0e; //RSVD       more magic internet values
33: dout <= 16'hB3_80; //THL_ST
//begin mystery scaling numbers
34: dout <= 16'h70_3a;
35: dout <= 16'h71_35;
36: dout <= 16'h72_11;
37: dout <= 16'h73_f0;
38: dout <= 16'ha2_02;
//gamma curve values
39: dout <= 16'h7a_20;
40: dout <= 16'h7b_10;
41: dout <= 16'h7c_1e;
42: dout <= 16'h7d_35;
43: dout <= 16'h7e_5a;
44: dout <= 16'h7f_69;
45: dout <= 16'h80_76;
46: dout <= 16'h81_80;
47: dout <= 16'h82_88;
48: dout <= 16'h83_8f;
49: dout <= 16'h84_96;
50: dout <= 16'h85_a3;
51: dout <= 16'h86_af;
52: dout <= 16'h87_c4;
53: dout <= 16'h88_d7;
```

```
54: dout <= 16'h89_e8;
//AGC and AEC
54: dout <= 16'h13_e0; //COM8, disable AGC / AEC
55: dout <= 16'h00_00; //set gain reg to 0 for AGC
56: dout <= 16'h10_00; //set ARCJ reg to 0
57: dout <= 16'h0d_40; //magic reserved bit for COM4
58: dout <= 16'h14_18; //COM9, 4x gain + magic bit
59: dout <= 16'ha5_05; // BD50MAX
60: dout <= 16'hab_07; //DB60MAX
61: dout <= 16'h24_95; //AGC upper limit
62: dout <= 16'h25_33; //AGC lower limit
63: dout <= 16'h26_e3; //AGC/AEC fast mode op region
64: dout <= 16'h9f_78; //HAECC1
65: dout <= 16'ha0_68; //HAECC2
66: dout <= 16'ha1_03; //magic
67: dout <= 16'ha6_d8; //HAECC3
68: dout <= 16'ha7_d8; //HAECC4
69: dout <= 16'ha8_f0; //HAECC5
70: dout <= 16'ha9_90; //HAECC6
71: dout <= 16'haa_94; //HAECC7
72: dout <= 16'h13_e5; //COM8, enable AGC / AEC
default: dout <= 16'hFF_FF;         //mark end of ROM
endcase

end
```

```
endmodule
`timescale 1ns / 1ps



module OV7670_config
#(
    parameter CLK_FREQ = 25000000
)
(
    input wire clk,
    input wire SCCB_interface_ready,
    input wire [15:0] rom_data,
    input wire start,
    output reg [7:0] rom_addr,
    output reg done,
    output reg [7:0] SCCB_interface_addr,
    output reg [7:0] SCCB_interface_data,
    output reg SCCB_interface_start
    );

    initial begin
        rom_addr = 0;
        done = 0;
        SCCB_interface_addr = 0;
        SCCB_interface_data = 0;
```

```
    SCCB_interface_start = 0;
end

localparam FSM_IDLE = 0;
localparam FSM_SEND_CMD = 1;
localparam FSM_DONE = 2;
localparam FSM_TIMER = 3;

reg [2:0] FSM_state = FSM_IDLE;
reg [2:0] FSM_return_state;
reg [31:0] timer = 0;

always@(posedge clk) begin

    case(FSM_state)

        FSM_IDLE: begin
            FSM_state <= start ? FSM_SEND_CMD : FSM_IDLE;
            rom_addr <= 0;
            done <= start ? 0 : done;
        end

        FSM_SEND_CMD: begin
            case(rom_data)
                16'hFFFF: begin //end of ROM
```

```
            FSM_state <= FSM_DONE;
        end

        16'hFFF0: begin //delay state
            timer <= (CLK_FREQ/100); //10 ms delay
            FSM_state <= FSM_TIMER;
            FSM_return_state <= FSM_SEND_CMD;
            rom_addr <= rom_addr + 1;
        end

        default: begin //normal rom commands
            if (SCCB_interface_ready) begin
                FSM_state <= FSM_TIMER;
                FSM_return_state <= FSM_SEND_CMD;
                timer <= 0; //one cycle delay gives ready chance to deassert
                rom_addr <= rom_addr + 1;
                SCCB_interface_addr <= rom_data[15:8];
                SCCB_interface_data <= rom_data[7:0];
                SCCB_interface_start <= 1;
            end
        end
    endcase
end

FSM_DONE: begin //signal done
```

```
            FSM_state <= FSM_IDLE;
            done <= 1;
        end


        FSM_TIMER: begin //count down and jump to next state
            FSM_state <= (timer == 0) ? FSM_return_state : FSM_TIMER;
            timer <= (timer==0) ? 0 : timer - 1;
            SCCB_interface_start <= 0;
        end
      endcase
   end
endmodule
`timescale 1ns / 1ps


module SCCB_interface
#(
   parameter CLK_FREQ = 25000000,
   parameter SCCB_FREQ = 100000
)
(
   input wire clk,
   input wire start,
   input wire [7:0] address,
```

```
input wire [7:0] data,
output reg ready,
output reg SIOC_oe,
output reg SIOD_oe
);

localparam CAMERA_ADDR = 8'h42;
localparam FSM_IDLE = 0;
localparam FSM_START_SIGNAL = 1;
localparam FSM_LOAD_BYTE = 2;
localparam FSM_TX_BYTE_1 = 3;
localparam FSM_TX_BYTE_2 = 4;
localparam FSM_TX_BYTE_3 = 5;
localparam FSM_TX_BYTE_4 = 6;
localparam FSM_END_SIGNAL_1 = 7;
localparam FSM_END_SIGNAL_2 = 8;
localparam FSM_END_SIGNAL_3 = 9;
localparam FSM_END_SIGNAL_4 = 10;
localparam FSM_DONE = 11;
localparam FSM_TIMER = 12;


initial begin
    SIOC_oe = 0;
    SIOD_oe = 0;
```

```
    ready = 1;
end

reg [3:0] FSM_state = 0;
reg [3:0] FSM_return_state = 0;
reg [31:0] timer = 0;
reg [7:0] latched_address;
reg [7:0] latched_data;
reg [1:0] byte_counter = 0;
reg [7:0] tx_byte = 0;
reg [3:0] byte_index = 0;


always@(posedge clk) begin

    case(FSM_state)

        FSM_IDLE: begin
            byte_index <= 0;
            byte_counter <= 0;
            if (start) begin
                FSM_state <= FSM_START_SIGNAL;
                latched_address <= address;
                latched_data <= data;
                ready <= 0;
```

```
    end
    else begin
        ready <= 1;
    end
end

FSM_START_SIGNAL: begin //comunication interface start signal, bring SIOD low
    FSM_state <= FSM_TIMER;
    FSM_return_state <= FSM_LOAD_BYTE;
    timer <= (CLK_FREQ/(4*SCCB_FREQ));
    SIOC_oe <= 0;
    SIOD_oe <= 1;
end

FSM_LOAD_BYTE: begin //load next byte to be transmitted
    FSM_state <= (byte_counter == 3) ? FSM_END_SIGNAL_1 : FSM_TX_BYTE_1;
    byte_counter <= byte_counter + 1;
    byte_index <= 0; //clear byte index
    case(byte_counter)
        0: tx_byte <= CAMERA_ADDR;
        1: tx_byte <= latched_address;
        2: tx_byte <= latched_data;
        default: tx_byte <= latched_data;
    endcase
end
```

```
        FSM_TX_BYTE_1: begin //bring SIOC low and and delay for next state
            FSM_state <= FSM_TIMER;
            FSM_return_state <= FSM_TX_BYTE_2;
            timer <= (CLK_FREQ/(4*SCCB_FREQ));
            SIOC_oe <= 1;
        end

        FSM_TX_BYTE_2: begin //assign output data,
            FSM_state <= FSM_TIMER;
            FSM_return_state <= FSM_TX_BYTE_3;
            timer <= (CLK_FREQ/(4*SCCB_FREQ)); //delay for SIOD to stabilize
            SIOD_oe <= (byte_index == 8) ? 0 : ~tx_byte[7]; //allow for 9 cycle ack, output enable
signal is inverting
        end

        FSM_TX_BYTE_3: begin // bring SIOC high
            FSM_state <= FSM_TIMER;
            FSM_return_state <= FSM_TX_BYTE_4;
            timer <= (CLK_FREQ/(2*SCCB_FREQ));
            SIOC_oe <= 0; //output enable is an inverting pulldown
        end

        FSM_TX_BYTE_4: begin //check for end of byte, incriment counter
            FSM_state <= (byte_index == 8) ? FSM_LOAD_BYTE : FSM_TX_BYTE_1;
```

```
            tx_byte <= tx_byte<<1; //shift in next data bit
            byte_index <= byte_index + 1;
        end

        FSM_END_SIGNAL_1: begin //state is entered with SIOC high, SIOD high. Start by bringing SIOC low
            FSM_state <= FSM_TIMER;
            FSM_return_state <= FSM_END_SIGNAL_2;
            timer <= (CLK_FREQ/(4*SCCB_FREQ));
            SIOC_oe <= 1;
        end

        FSM_END_SIGNAL_2: begin // while SIOC is low, bring SIOD low
            FSM_state <= FSM_TIMER;
            FSM_return_state <= FSM_END_SIGNAL_3;
            timer <= (CLK_FREQ/(4*SCCB_FREQ));
            SIOD_oe <= 1;
        end

        FSM_END_SIGNAL_3: begin // bring SIOC high
            FSM_state <= FSM_TIMER;
            FSM_return_state <= FSM_END_SIGNAL_4;
            timer <= (CLK_FREQ/(4*SCCB_FREQ));
            SIOC_oe <= 0;
        end
```

```
        FSM_END_SIGNAL_4: begin // bring SIOD high when SIOC is high
            FSM_state <= FSM_TIMER;
            FSM_return_state <= FSM_DONE;
            timer <= (CLK_FREQ/(4*SCCB_FREQ));
            SIOD_oe <= 0;
        end

        FSM_DONE: begin //add delay between transactions
            FSM_state <= FSM_TIMER;
            FSM_return_state <= FSM_IDLE;
            timer <= (2*CLK_FREQ/(SCCB_FREQ));
            byte_counter <= 0;
        end

        FSM_TIMER: begin //count down and jump to next state
            FSM_state <= (timer == 0) ? FSM_return_state : FSM_TIMER;
            timer <= (timer==0) ? 0 : timer - 1;
        end
      endcase
    end


endmodule
/* SD Card controller module. Allows reading from and writing to a microSD card
```

through SPI mode. */

`timescale 1ns / 1ps

module sd_controller(
    output reg cs, // Connect to SD_DAT[3].
    output mosi, // Connect to SD_CMD.
    input miso, // Connect to SD_DAT[0].
    output sclk, // Connect to SD_SCK.
            // For SPI mode, SD_DAT[2] and SD_DAT[1] should be held HIGH.
            // SD_RESET should be held LOW.

    input rd,   // Read-enable. When [ready] is HIGH, asseting [rd] will
            // begin a 512-byte READ operation at [address].
            // [byte_available] will transition HIGH as a new byte has been
            // read from the SD card. The byte is presented on [dout].
    output reg [7:0] dout, // Data output for READ operation.
    output reg byte_available, // A new byte has been presented on [dout].

    input wr,   // Write-enable. When [ready] is HIGH, asserting [wr] will
            // begin a 512-byte WRITE operation at [address].
            // [ready_for_next_byte] will transition HIGH to request that
            // the next byte to be written should be presentaed on [din].
    input [7:0] din, // Data input for WRITE operation.
    output reg ready_for_next_byte, // A new byte should be presented on [din].

```
    input reset, // Resets controller on assertion.
    output ready, // HIGH if the SD card is ready for a read or write operation.
    input [31:0] address,   // Memory address for read/write operation. This MUST
                            // be a multiple of 512 bytes, due to SD sectoring.
    input clk,  // 25 MHz clock.
    output [4:0] status // For debug purposes: Current state of controller.
);

    parameter RST = 0;
    parameter INIT = 1;
    parameter CMD0 = 2;
    parameter CMD55 = 3;
    parameter CMD41 = 4;
    parameter POLL_CMD = 5;

    parameter IDLE = 6;
    parameter READ_BLOCK = 7;
    parameter READ_BLOCK_WAIT = 8;
    parameter READ_BLOCK_DATA = 9;
    parameter READ_BLOCK_CRC = 10;
    parameter SEND_CMD = 11;
    parameter RECEIVE_BYTE_WAIT = 12;
    parameter RECEIVE_BYTE = 13;
    parameter WRITE_BLOCK_CMD = 14;
```

```
parameter WRITE_BLOCK_INIT = 15;
parameter WRITE_BLOCK_DATA = 16;
parameter WRITE_BLOCK_BYTE = 17;
parameter WRITE_BLOCK_WAIT = 18;

parameter WRITE_DATA_SIZE = 515;

reg [4:0] state = RST;
assign status = state;
reg [4:0] return_state;
reg sclk_sig = 0;
reg [55:0] cmd_out;
reg [7:0] recv_data;
reg cmd_mode = 1;
reg [7:0] data_sig = 8'hFF;

reg [9:0] byte_counter;
reg [9:0] bit_counter;

reg [26:0] boot_counter = 27'd100_000_000;
always @(posedge clk) begin
    if(reset == 1) begin
        state <= RST;
        sclk_sig <= 0;
        boot_counter <= 27'd100_000_000;
```

```
end
else begin
    case(state)
        RST: begin
            if(boot_counter == 0) begin
                sclk_sig <= 0;
                cmd_out <= {56{1'b1}};
                byte_counter <= 0;
                byte_available <= 0;
                ready_for_next_byte <= 0;
                cmd_mode <= 1;
                bit_counter <= 160;
                cs <= 1;
                state <= INIT;
            end
            else begin
                boot_counter <= boot_counter - 1;
            end
        end
        INIT: begin
            if(bit_counter == 0) begin
                cs <= 0;
                state <= CMD0;
            end
            else begin
```

```
        bit_counter <= bit_counter - 1;
        sclk_sig <= ~sclk_sig;
    end
end
CMD0: begin
    cmd_out <= 56'hFF_40_00_00_00_00_95;
    bit_counter <= 55;
    return_state <= CMD55;
    state <= SEND_CMD;
end
CMD55: begin
    cmd_out <= 56'hFF_77_00_00_00_00_01;
    bit_counter <= 55;
    return_state <= CMD41;
    state <= SEND_CMD;
end
CMD41: begin
    cmd_out <= 56'hFF_69_00_00_00_00_01;
    bit_counter <= 55;
    return_state <= POLL_CMD;
    state <= SEND_CMD;
end
POLL_CMD: begin
    if(recv_data[0] == 0) begin
        state <= IDLE;
```

```
    end
    else begin
        state <= CMD55;
    end
end
IDLE: begin
    if(rd == 1) begin
        state <= READ_BLOCK;
    end
    else if(wr == 1) begin
        state <= WRITE_BLOCK_CMD;
    end
    else begin
        state <= IDLE;
    end
end
READ_BLOCK: begin
    cmd_out <= {16'hFF_51, address, 8'hFF};
    bit_counter <= 55;
    return_state <= READ_BLOCK_WAIT;
    state <= SEND_CMD;
end
READ_BLOCK_WAIT: begin
    if(sclk_sig == 1 && miso == 0) begin
        byte_counter <= 511;
```

```
            bit_counter <= 7;
            return_state <= READ_BLOCK_DATA;
            state <= RECEIVE_BYTE;
        end
        sclk_sig <= ~sclk_sig;
    end
    READ_BLOCK_DATA: begin
        dout <= recv_data;
        byte_available <= 1;
        if (byte_counter == 0) begin
            bit_counter <= 7;
            return_state <= READ_BLOCK_CRC;
            state <= RECEIVE_BYTE;
        end
        else begin
            byte_counter <= byte_counter - 1;
            return_state <= READ_BLOCK_DATA;
            bit_counter <= 7;
            state <= RECEIVE_BYTE;
        end
    end
    READ_BLOCK_CRC: begin
        bit_counter <= 7;
        return_state <= IDLE;
        state <= RECEIVE_BYTE;
```

```
end
SEND_CMD: begin
    if (sclk_sig == 1) begin
        if (bit_counter == 0) begin
            state <= RECEIVE_BYTE_WAIT;
        end
        else begin
            bit_counter <= bit_counter - 1;
            cmd_out <= {cmd_out[54:0], 1'b1};
        end
    end
    sclk_sig <= ~sclk_sig;
end
RECEIVE_BYTE_WAIT: begin
    if (sclk_sig == 1) begin
        if (miso == 0) begin
            recv_data <= 0;
            bit_counter <= 6;
            state <= RECEIVE_BYTE;
        end
    end
    sclk_sig <= ~sclk_sig;
end
RECEIVE_BYTE: begin
    byte_available <= 0;
```

```
    if (sclk_sig == 1) begin
        recv_data <= {recv_data[6:0], miso};
        if (bit_counter == 0) begin
            state <= return_state;
        end
        else begin
            bit_counter <= bit_counter - 1;
        end
    end
    sclk_sig <= ~sclk_sig;
end
WRITE_BLOCK_CMD: begin
    cmd_out <= {16'hFF_58, address, 8'hFF};
    bit_counter <= 55;
    return_state <= WRITE_BLOCK_INIT;
    state <= SEND_CMD;
          ready_for_next_byte <= 1;
end
WRITE_BLOCK_INIT: begin
    cmd_mode <= 0;
    byte_counter <= WRITE_DATA_SIZE;
    state <= WRITE_BLOCK_DATA;
    ready_for_next_byte <= 0;
end
WRITE_BLOCK_DATA: begin
```

```
    if (byte_counter == 0) begin
        state <= RECEIVE_BYTE_WAIT;
        return_state <= WRITE_BLOCK_WAIT;
    end
    else begin
        if ((byte_counter == 2) || (byte_counter == 1)) begin
            data_sig <= 8'hFF;
        end
        else if (byte_counter == WRITE_DATA_SIZE) begin
            data_sig <= 8'hFE;
        end
        else begin
            data_sig <= din;
            ready_for_next_byte <= 1;
        end
        bit_counter <= 7;
        state <= WRITE_BLOCK_BYTE;
        byte_counter <= byte_counter - 1;
    end
end
WRITE_BLOCK_BYTE: begin
    if (sclk_sig == 1) begin
        if (bit_counter == 0) begin
            state <= WRITE_BLOCK_DATA;
            ready_for_next_byte <= 0;
```

```
                end
                else begin
                    data_sig <= {data_sig[6:0], 1'b1};
                    bit_counter <= bit_counter - 1;
                end;
            end;
            sclk_sig <= ~sclk_sig;
        end
        WRITE_BLOCK_WAIT: begin
            if (sclk_sig == 1) begin
                if (miso == 1) begin
                    state <= IDLE;
                    cmd_mode <= 1;
                end
            end
            sclk_sig = ~sclk_sig;
        end
    endcase
  end
end

assign sclk = sclk_sig;
assign mosi = cmd_mode ? cmd_out[55] : data_sig[7];
assign ready = (state == IDLE);
endmodule
```

```
`timescale 1ns / 1ps



module sobel(
    input wire clk,
    input wire start,
    output reg done,
    input wire [5:0] SW,  //lower 6 switches

    //from the BRAM of the original picture
    input wire [11:0] pixel_data,
    output reg [18:0] pic_memory_addr,

    //writing to the BRAM of the edges
    output reg [3:0] is_edge,
    output reg [18:0] edge_memory_addr
    );

    reg [3:0] bw_pixel0, bw_pixel1, bw_pixel2, bw_pixel3, bw_pixel4, bw_pixel5, bw_pixel6,
bw_pixel7, bw_pixel8;
    reg [3:0] bw_pixel_load0, bw_pixel_load1, bw_pixel_load2;

    parameter STATE_SETUP = 0;
    parameter STATE_THRESHOLD = 2;
    parameter STATE_WAIT = 3;
```

```
parameter STATE_WAIT2 = 4;
parameter STATE_GET_9 = 5;
parameter STATE_CALCULATE_G = 6;
parameter STATE_SQUARE = 7;
parameter STATE_SHIFTWINDOW = 8;
parameter STATE_SQUARE2 = 9;
reg [4:0] state = STATE_SETUP;
reg [4:0] next_state;

parameter WIDTH = 640;
parameter HEIGHT = 480;

reg [9:0] x;
reg [8:0] y;
reg [9:0] x_buffer = 25;
reg [8:0] y_buffer = 25;

reg [6:0] GX;
reg [6:0] GY;

reg [11:0] GX2; //11
reg [11:0] GY2; //11
reg [11:0] threshold; // = 12'd3000;
```

```
reg [3:0] i = 0;
reg [5:0] old_SW = 16'hFFFF;

always @(posedge clk) begin
    if (~start) begin
            threshold <= 0;
            state <= STATE_SETUP;
            done <= 0;
        end
    if (~done) begin
    case (state)
        STATE_SETUP: begin
            pic_memory_addr <= 0;
            i <= 0;
            x <= 1;
            y <= 1;
            edge_memory_addr <= 641;
            done <= 0;

            if (start) begin
                state <= STATE_WAIT;
                next_state <= STATE_GET_9;
            end
        end
```

```
        STATE_WAIT: state <= STATE_WAIT2;


        STATE_WAIT2: state <= next_state;


        STATE_GET_9: begin
            i <= i + 1;
            state <= STATE_WAIT;
            next_state <= STATE_GET_9;


            case (i)
                0: begin
                    bw_pixel0 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) + (pixel_data[7:4] >>
3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr + 1;
                end
                1: begin
                    bw_pixel1 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) + (pixel_data[7:4] >>
3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr + 1;
                end
                2: begin
                    bw_pixel2 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) + (pixel_data[7:4] >>
3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr + WIDTH - 2;
                end
```

```
                3: begin
                    bw_pixel3 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) + (pixel_data[7:4] >>
3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr + 1;
                end
                4: begin
                    bw_pixel4 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) + (pixel_data[7:4] >>
3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr + 1;
                end
                5: begin
                    bw_pixel5 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) + (pixel_data[7:4] >>
3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr + WIDTH - 2;
                end
                6: begin
                    bw_pixel6 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) + (pixel_data[7:4] >>
3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr + 1;
                end
                7: begin
                    bw_pixel7 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) + (pixel_data[7:4] >>
3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr + 1;
                end
                8: begin
```

```
                bw_pixel8 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) + (pixel_data[7:4] >>
3) + (pixel_data[3:0] >> 3);

                pic_memory_addr <= pic_memory_addr - WIDTH - WIDTH + 1;

                state <= STATE_CALCULATE_G;

            end

        endcase

    end


    STATE_CALCULATE_G: begin

        GX <= bw_pixel0 - bw_pixel2 + (bw_pixel3 << 1) - (bw_pixel5 << 1) + bw_pixel6 -
bw_pixel8;

        GY <= bw_pixel0 + (bw_pixel1 << 1) + bw_pixel2 - bw_pixel6 - (bw_pixel7 << 1) -
bw_pixel8;

        state <= STATE_SQUARE;

    end


    STATE_SQUARE: begin

        GX2 <= GX * GX;

        state <= STATE_SQUARE2;

    end


    STATE_SQUARE2: begin

        GY2 <= GY * GY;


        i <= 0;
```

```
            state <= STATE_SHIFTWINDOW;
        end

        STATE_SHIFTWINDOW: begin
            i <= i + 1;
            state <= STATE_WAIT;
            next_state <= STATE_SHIFTWINDOW;
            case(i)
                0: begin
                    bw_pixel_load0 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) +
(pixel_data[7:4] >> 3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr + WIDTH;
                end

                1: begin
                    bw_pixel_load1 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) +
(pixel_data[7:4] >> 3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr + WIDTH;
                end

                2: begin
                    bw_pixel_load2 = (pixel_data[11:8] >> 2) + (pixel_data[7:4] >> 1) +
(pixel_data[7:4] >> 3) + (pixel_data[3:0] >> 3);
                    pic_memory_addr <= pic_memory_addr - WIDTH - WIDTH + 1;
                end
```

```
        3: begin
            bw_pixel0 <= bw_pixel1;
            bw_pixel1 <= bw_pixel2;
            bw_pixel2 <= bw_pixel_load0;
            bw_pixel3 <= bw_pixel4;
            bw_pixel4 <= bw_pixel5;
            bw_pixel5 <= bw_pixel_load1;
            bw_pixel6 <= bw_pixel7;
            bw_pixel7 <= bw_pixel8;
            bw_pixel8 <= bw_pixel_load2;

            if (x == WIDTH - 1) begin
                x <= 0;
                y <= y + 1;
            end
            else begin
                x <= x + 1;
            end

            state <= STATE_THRESHOLD;
        end
    endcase
end

STATE_THRESHOLD: begin
```

```
            edge_memory_addr <= edge_memory_addr + 1;
            is_edge <= (GX2 + GY2) > threshold ? 3'b001: 3'b000;
            state <= STATE_CALCULATE_G;

            i <= 0;


            if ((x < x_buffer)  || (x > (WIDTH - x_buffer)) || (y < y_buffer) || ( y >( HEIGHT - y_buffer
))) begin
                state <= STATE_SHIFTWINDOW;
                is_edge <= 0;
            end
            else begin
                state <= STATE_CALCULATE_G;
            end

            if (( x == WIDTH - 1) && (y == HEIGHT - 2)) begin
                done <= 1;
            end
          end
       endcase
       end
   end
endmodule
`timescale 1ns / 1ps
```

```
module audio_processing(
    input clock,
    input clk_25mhz,
    input reset,
    input wings_done,
    input julian_done,
    input [7:0] sd_data,
    input sd_ready,
    input sd_read_available,
    input [1:0] filter_control,
    output AUD_PWM,
    output reg [31:0] sd_addr,
    output AUD_SD,
    output reg sd_rd,
    output fft_bram_out_write_enablea,
    output reg [9:0] fft_bram_out_addra,
    output [8:0] fft_bram_out_dina,
    output [2:0] bin,
    output [59:0] bin_addr,
    output done,
    output reg [2:0] state,
    output reg filter_flag,
    output filter_done,
```

```
    output filter_ready,
    output wire flag,
    output wire last_flag,
    output test

    );

    //fsm signals
    reg fft_first_round;
    reg [12:0] fft_send_count;
    reg make_fft_valid;
    reg [2:0] state;
    reg first_sample_flag;
    reg filter_flag;

    //32khz clk for audio
    reg last_clk_32khz;
    wire clk_32khz; // audio sample rate clock
    clock_divider clk_32khz_module(.clk_in(clock), .clk_out(clk_32khz), .divider(32'd1563),
.reset(reset)); // 100_000_000 / (32_000*2) = 1563


    //SD to fifo
    reg [8:0] sd_count;
    reg last_sd_read_available;
```

```
    wire overflow;

    reg fwrite;

    reg fread;

    wire [7:0] fifo_to_filter;

    fifo #(.LOGSIZE(9), .WIDTH(8))
myfifo(.clk(clock),.reset(reset),.rd(fread),.wr(fwrite),.din(sd_data), .full(ffull),

            .empty(fempty), .overflow(overflow), .dout(fifo_to_filter));


    //fifo to filter

    wire [7:0] audio_out;

    //wire filter_done;

    reg filter_ready;

    wire filt_ready;

    assign filt_ready = filter_ready;

    wire signed [7:0] filter_to_fft;

    filter_control filter_controller(.clock(clock), .reset(reset), .switch(filter_control),
.ready(filt_ready),

        .audio_in(fifo_to_filter), .audio_out(audio_out), .done(filter_done), .flag(flag),
.last_flag(last_flag));




    //filter to bram

    wire [10:0] filter_bram_addra;

    reg [10:0] filter_bram_addra_filter_reg;

    reg [10:0] filter_bram_addra_fft_reg;

    wire [7:0] fft_input;
```

```
    reg internal_bram_we;
    //assign internal_bram_we = filter_done;
    filter_input_buffer filter_bram (
      .clka(clock),    // input wire clka
      .wea(internal_bram_we),      // input wire [0 : 0] wea
      .addra(filter_bram_addra),  // input wire [10 : 0] addra
      .dina(audio_out),    // input wire [7 : 0] dina
      .douta(fft_input)  // output wire [7 : 0] douta
    );

    //filter to audio PWM
    assign AUD_SD = 1;
    audio_PWM audio_at32khz(.clk(clock), .reset(reset),.music_data(audio_out),
.PWM_out(AUD_PWM));


    //internal bram to FFT

    //internal bram address multiplexing
    assign filter_bram_addra = (state == 3'b100 ) ? filter_bram_addra_fft_reg :
filter_bram_addra_filter_reg;

    //FFT signals related to input data
    assign zero_extended_fft_input_data = {8'b0, fft_input};
    wire signed [15:0] fft_input_data = {1'b0, zero_extended_fft_input_data} - (1 << 15);
```

```
    wire fft_data_valid; //data is ready for fft

    assign fft_data_valid = (julian_done && fft_send_count <=1023 && fft_ready &&
make_fft_valid) ? 1 : 0;

    wire last_sample;

    assign last_sample = (fft_send_count == 1024) ? 1 :0;


    //FFT signals related to output data

    wire [15:0] fft_output_data;

    wire fft_out_data_valid;

    wire fft_out_data_last; //last sample in frame

    wire [9:0] tuser;


    fft_mag my_fft(.clk(clock),
.event_tlast_missing(tlast_missing),.frame_tdata(fft_input_data),.frame_tlast(last_sample),

            .frame_tready(fft_ready), .frame_tvalid(fft_data_valid),
.magnitude_tdata(fft_output_data), .magnitude_tlast(fft_out_data_last),

            .magnitude_tuser(tuser), .magnitude_tvalid(fft_out_data_valid));


    //FFT to external bram

    assign fft_bram_out_write_enablea = (state == 3'b100) && julian_done && fft_out_data_valid;

    assign fft_bram_out_dina = fft_output_data;


    reg done_reg;

    initial done_reg = 0;

    assign done = done_reg;

    //assign done = (state != 3'b100) ? !julian_done : (!julian_done || fft_out_data_last);
```

```
//FFT to Julian
assign bin = 3'd4;
//assign bin_addr = {30'd0, 10'd300, 10'd700, 10'd1023};
assign bin_addr = {20'd0,10'd1023, 10'd691, 10'd541, 10'd521};



reg test;
reg [10:0] av_counter;
initial av_counter = 0;
always @ (posedge clock)
begin
    last_clk_32khz <= clk_32khz;
    if(done)
    begin
        test <= 1;
    end
    if (reset)
    begin
        sd_addr <= 32'h0;
        filter_bram_addra_fft_reg <= 11'b0;
        filter_bram_addra_filter_reg <= 11'b0;
        fwrite <= 0;
        fread <= 0;
        sd_rd <= 0;
```

```
    state <= 3'b000;
    make_fft_valid <= 0;
    filter_ready <= 0;
    first_sample_flag <= 1;
    filter_flag <= 0;
end
else
begin
    if(wings_done)
    begin
        last_sd_read_available <= sd_read_available;
        case(state)
            //SD --> FIFO --> Filter --> audioPWM
            //                ||
            //                --> BRAM1 ---> FFT --> BRAM2
            3'b000:
            begin
                //whenever julian is done, i am not done
                if (julian_done)
                begin
                    done_reg <=0;
                end
                // when filtering is done write it to the bram and read a new value into the filter
                // as a 32khz rate to match the audio requirements
                if (filter_done)
```

```
begin
   filter_flag <= 1;
end
else if (filter_flag && last_clk_32khz == 0 && clk_32khz == 1)
begin
   fread <= 1;
   filter_ready <= 1;
   internal_bram_we <= 1;
   filter_flag <=0;
   state <= 3'b011;
end
else
begin
   state <= 3'b001;
end
//start filling up the fifo if it is empty
if (fempty)
begin
   sd_rd <=1;
end
else
begin
   sd_rd<=0;
end
```

```
                // fifo has room and sd card byte is avaliable so write to fifo, increment
sd_count(reading)
                // and change states to see if filter is ready to read from fifo if this is the first value
                if (last_sd_read_available == 0 && sd_read_available == 1 )
                begin
                   fwrite <= 1;
                   sd_count <= sd_count + 1;

                end
                // otherwise don't write to fifo
                else
                begin
                   fwrite <= 0;
                end
                // if 512 bytes have been read from SD card then increment address by 200 hex
and reset sd_count since
                // the fifo will be full
                if (sd_count >= 511)
                begin
                   sd_addr <= sd_addr + 32'h200;
                   sd_count <= 0;
                end



            end
```

```
                3'b001:
                begin
                    if (julian_done)
                    begin
                        done_reg <=0;
                    end

                     // fifo has room and sd card byte is avaliable so write to fifo, increment
sd_count(reading)
                    if (last_sd_read_available == 0 && sd_read_available == 1 )
                    begin
                        fwrite <= 1;
                        sd_count <= sd_count + 1;
                    end
                    // otherwise don't write to fifo
                    else
                    begin
                        fwrite <= 0;
                    end
                    // if 512 bytes have been read from SD card then increment address by 200 hex
and reset sd_count since
                    // the fifo will be full
                    if (sd_count >= 511)
                    begin
                        sd_addr <= sd_addr + 32'h200;
                        sd_count <= 0;
```

```
end

//only write one value to fifo
//if this is the first value being written to fifo from sd card simply shift into filter and
// go into state waiting for filter to finish

 if(first_sample_flag)
 begin
    first_sample_flag <=0;
    fread <= 1;
    filter_ready <= 1;
    state <= 3'b010;
    fwrite <=0;

 end
 else
 begin
    // when filtering is done write it to the bram and read a new value into the filter
    // as a 32khz rate to match the audio requirements
    if (filter_done)
    begin
       filter_flag <= 1;
       fread <= 0;
    end
```

```
                //otherwise go straight to waiting for the filter to finish processing previous
sample
                else if (filter_flag && last_clk_32khz == 0 && clk_32khz == 1)
                begin
                   fread <= 1;
                   filter_ready <= 1;
                   internal_bram_we <= 1;
                   filter_flag <=0;
                   state <= 3'b011;
                end
                else if(!filter_flag)
                begin
                   fread <= 0;
                   filter_ready <= 0;
                   state <= 3'b010;
                end
                else
                begin
                   filter_ready <= 0;
                end

             end
          end
          3'b010:
          begin
```

```
                // filtering is done so write it to the bram and read a new value into the filter
                if (filter_done)
                begin
                    filter_flag <= 1;
                    fread <= 0;
                end
                else if (filter_flag && last_clk_32khz == 0 && clk_32khz == 1)
                begin
                    fread <= 1;
                    filter_ready <= 1;
                    internal_bram_we <= 1;
                    filter_flag <= 0;
                    state <= 3'b011;
                end
                //go back and try to add another value to the fifo  while waiting for the filter
                else
                begin
                    //only read out one value from fifo
                    fread <= 0;
                    //filter is not ready for new values
                    filter_ready <= 0;
                end
                // fifo has room and sd card byte is avaliable so write to fifo, increment
sd_count(reading)
                if (last_sd_read_available == 0 && sd_read_available == 1 )
```

```
                begin
                    fwrite <= 1;
                    sd_count <= sd_count + 1;
                end
                // otherwise don't write to fifo
                else
                begin
                    fwrite <= 0;
                end
                // if 512 bytes have been read from SD card then increment address by 200 hex and reset sd_count since
                // the fifo will be full
                if (sd_count >= 511)
                begin
                    sd_addr <= sd_addr + 32'h200;
                    sd_count <= 0;
                end
                if (julian_done)
                begin
                    done_reg <=0;
                end
            end
            3'b011:
            begin
                //if coming from state 1 fread will need to be set to 0 so that multiple values are not read
```

```
                // from the fifo
                fread <= 0;
                //if coming from state 0,1, or 2 and a byte was avaliable for mthe sd card then fwrite needs to de
                // asserted
                fwrite <= 0;
                //a value was just written to the bram so increment the address
                filter_bram_addra_filter_reg <= filter_bram_addra_filter_reg + 1;
                //if 1024 of 1535 values have been filtered and written to the bram
                //then send the first fft window over and also set first fft flag
                // also disable internal bram writing
                if (filter_bram_addra_filter_reg == 1024  || filter_bram_addra_filter_reg == 1536)
                begin
                   state <= 3'b100;
                   internal_bram_we <= 0;
                   fft_first_round <= 1;
                end
                // 512 and 0 also trigger an fft window after the first window has been sent
                // also disable internal bram writing
                else if ((filter_bram_addra_filter_reg == 512 || filter_bram_addra_filter_reg == 0) && fft_first_round == 1)
                begin
                   state <= 3'b100;
                   internal_bram_we <= 0;
                end
                // go back to state 0 to push more data from the sd card into the pipeline if
```

```
                // there is not enough data for the fft to run yet
                else
                begin
                    state <= 3'b000;
                end
                // fifo has room and sd card byte is avaliable so write to fifo, increment
sd_count(reading)
                if (last_sd_read_available == 0 && sd_read_available == 1 )
                begin
                    fwrite <= 1;
                    sd_count <= sd_count + 1;
                end
                // otherwise don't write to fifo
                else
                begin
                    fwrite <= 0;
                end
                // if 512 bytes have been read from SD card then increment address by 200 hex
and reset sd_count since
                // the fifo will be full
                if (sd_count >= 511)
                begin
                    sd_addr <= sd_addr + 32'h200;
                    sd_count <= 0;
                end
                if (julian_done)
```

```
            begin
                done_reg <=0;
            end
        end
        3'b100:
        begin
            // once julian is done using external bram check if fft is ready to accept data
            if(julian_done)
            begin
                if (fft_ready && fft_send_count <= 1023)
                begin
                    //account for one clock cycle bram read delay by reseting the data to invalid
after
                    //each value is output from fft module
                    if (!fft_data_valid)
                    begin
                        make_fft_valid <= 1;
                    end
                    //increment fft send count so that only 1024 values are output
                    if (fft_out_data_valid)
                    begin
                        fft_send_count <= fft_send_count +1;
                        filter_bram_addra_fft_reg <= filter_bram_addra_fft_reg + 1;
                        if(av_counter ==0)
                        begin
```

```
                fft_bram_out_addra <= fft_bram_out_addra +1;
            end
            else
            begin
                fft_bram_out_addra <= fft_bram_out_addra;
            end
            make_fft_valid <= 0;
        end
    end
    //reset the internal bram reading address used by the fft to 512 back so that
windows
    //overlap by half the width
    if (fft_out_data_last)
    begin
        if(av_counter == 4)
        begin
            av_counter <= 0;
        end
        else
        begin
            av_counter <= av_counter + 1;
        end
        fft_send_count <= 0;
        filter_bram_addra_fft_reg <= filter_bram_addra_fft_reg - 512;
        fft_bram_out_addra <= 0;
```

```
                done_reg <= 1;
                state <= 3'b000;
            end
            else
            begin
                done_reg <= 0;
            end
        end
        // fifo has room and sd card byte is avaliable so write to fifo, increment
sd_count(reading)
        if (last_sd_read_available == 0 && sd_read_available == 1 )
        begin
            fwrite <= 1;
            sd_count <= sd_count + 1;
        end
        // otherwise don't write to fifo
        else
        begin
            fwrite <= 0;
        end
        // if 512 bytes have been read from SD card then increment address by 200 hex
and reset sd_count since
        // the fifo will be full
        if (sd_count >= 511)
        begin
            sd_addr <= sd_addr + 32'h200;
```

```
                    sd_count <= 0;
                end
            end
        endcase
    end
  end
end
endmodule
`timescale 1ns / 1ps


module video_playback(
//    input wire [11:0] pixel_data,
    input wire [2:0] pixel_data,
    input wire [12:0] rgb_data,
    input wire video_clk,
    output wire [18:0] memory_addr,
    output reg vsync,
    output reg hsync,
    output wire [11:0] video_out,
    input wire contour
    );


    // horizontal: 800 pixels total
```

```
// display 640 pixels per line
reg hblank,vblank;
wire hsyncon,hsyncoff,hreset,hblankon;
reg [11:0] hcount = 0;
reg [11:0] vcount = 0;
reg blank;
//kludges to fix frame alignment due to memory access time
reg blank_delay;
reg blank_delay_2;
reg hsync_pre_delay;
reg hsync_pre_delay_2;
reg vsync_pre_delay;
reg vsync_pre_delay_2;

reg at_display_area;
reg [3:0] bw_value;
reg [11:0] bw_total;

assign video_out = ~at_display_area ? 0:
                   ~contour ? rgb_data :
                   pixel_data == 3'b001 ? 12'hF00 :
                   pixel_data == 3'b010 ? 12'h0F0 :
                   pixel_data == 3'b011 ? 12'h00F :
                   pixel_data == 3'b100 ? 12'hF0F :
                   pixel_data == 3'b101 ? 12'h0FF :
```

```
                pixel_data == 3'b110 ? 12'h4F4 :
                pixel_data == 3'b111 ? 12'hFF0 : rgb_data;

assign hblankon = (hcount == 639);   //blank after display width
assign hsyncon = (hcount == 655);  // active video + front porch
assign hsyncoff = (hcount == 751); //active video + front portch + sync
assign hreset = (hcount == 799); //plus back porch

// vertical:  525 lines total
// display 480 lines
wire vsyncon,vsyncoff,vreset,vblankon;
assign vblankon = hreset & (vcount == 479);
assign vsyncon = hreset & (vcount == 489);
assign vsyncoff = hreset & (vcount == 491);
assign vreset = (hreset & (vcount == 524));

// sync and blanking
wire next_hblank,next_vblank;
assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;

assign memory_addr = hcount+vcount*640;

always @(posedge video_clk) begin
```

```
        blank_delay <= blank;
        blank_delay_2 <= blank_delay;
        hsync_pre_delay_2 <= hsync_pre_delay;
        vsync_pre_delay_2 <= vsync_pre_delay;
        vsync <= vsync_pre_delay_2;
        hsync <= hsync_pre_delay_2;
         //hcount
         hcount <= hreset ? 0 : hcount + 1;
         hblank <= next_hblank;
         hsync_pre_delay <= hsyncon ? 0 : hsyncoff ? 1 : hsync_pre_delay;  // active low

         vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
         vblank <= next_vblank;
         vsync_pre_delay <= vsyncon ? 0 : vsyncoff ? 1 : vsync_pre_delay;  // active low

         blank <= next_vblank | (next_hblank & ~hreset);

        at_display_area <= ((hcount >= 0) && (hcount < 640) && (vcount >= 0) && (vcount < 480));

        bw_value <= (rgb_data[11:8] >> 2) + (rgb_data[7:4] >> 1) + (rgb_data[7:4] >> 3) +
(rgb_data[3:0] >> 3);
        bw_total <= {bw_value, bw_value, bw_value};
```

```
    end

endmodule
```